μKummer: efficient hyperelliptic signatures and key exchange on microcontrollers

Joost Renes[1], Peter Schwabe[1], Benjamin Smith[2], and Lejla Batina[1] *

[1] Digital Security Group, Radboud University, The Netherlands
lejla@cs.ru.nl,j.renes@cs.ru.nl,peter@cryptojedi.org
[2] INRIA *and* Laboratoire d'Informatique de l'École polytechnique (LIX), France
smith@lix.polytechnique.fr

Abstract. We describe the design and implementation of efficient signature and key-exchange schemes for the AVR ATmega and ARM Cortex M0 microcontrollers, targeting the 128-bit security level. Our algorithms are based on an efficient Montgomery ladder scalar multiplication on the Kummer surface of Gaudry and Schost's genus-2 hyperelliptic curve, combined with the Jacobian point recovery technique of Chung, Costello, and Smith. Our results are the first to show the feasibility of software-only hyperelliptic cryptography on constrained platforms, and represent a significant improvement on the elliptic-curve state-of-the-art for both key exchange and signatures on these architectures. Notably, our key-exchange scalar-multiplication software runs in under 9520k cycles on the ATmega and under 2640k cycles on the Cortex M0, improving on the current speed records by 32% and 75% respectively.

Keywords. Hyperelliptic curve cryptography, Kummer surface, AVR ATmega, ARM Cortex M0.

1 Introduction

The current state of the art in asymmetric cryptography, not only on microcontrollers, is elliptic-curve cryptography; the most widely accepted reasonable security is the 128-bit security level. All current speed records for 128-bit secure key exchange and signatures on microcontrollers are held—until now—by elliptic-curve-based schemes. Outside the world of microcontrollers, it is well known that genus-2 hyperelliptic curves and their Kummer surfaces present an attractive alternative to elliptic curves [1, 2]. For example, at Asiacrypt 2014 Bernstein, Chuengsatiansup, Lange and Schwabe [3] presented speed records for timing-attack-protected 128-bit-secure scalar multiplication on a range of architectures with Kummer-based software. These speed records are currently only being surpassed by the elliptic-curve-based Fourℚ software by Costello and

* This work has been supported by the Netherlands Organisation for Scientific Research (NWO) through Veni 2013 project 13114 and by the Technology Foundation STW (project 13499 - TYPHOON & ASPASIA), from the Dutch government. Permanent ID of this document: b230ab9b9c664ec4aad0cea0bd6a6732. Date: 2016-07-19

Longa [4] presented at Asiacrypt 2015, which makes heavy use of efficiently computable endomorphisms (i.e., of additional structure of the underlying elliptic curve). The Kummer-based speed records in [3] were achieved by exploiting the computational power of vector units of recent "large" processors such as Intel Sandy Bridge, Ivy Bridge, and Haswell, or the ARM Cortex-A8. Surprisingly, very little attention has been given to Kummer surfaces on embedded processors. Indeed, this is the first work showing the feasibility of software-only implementations of hyperelliptic-curve based crypto on constrained platforms. There have been some investigations of binary hyperelliptic curves targeting the much lower 80-bit security level, but those are actually examples of software-hardware co-design showing that using hardware acceleration for field operations was necessary to get reasonable performance figures (see eg. [5] and [6]).

In this paper we investigate the potential of genus-2 hyperelliptic curves for both key exchange and signatures on the "classical" 8-bit AVR ATmega architecture, and the more modern 32-bit ARM Cortex-M0 processor. The former has the most previous results to compare to, while ARM is becoming more relevant in real-world applications. We show that not only are hyperelliptic curves competitive, they clearly outperform state-of-the art elliptic-curve schemes in terms of speed and size. For example, our variable-basepoint scalar multiplication on a 127-bit Kummer surface is 31% faster on AVR and 26% faster on the M0 than the recently presented speed records for Curve25519 software by Düll, Haase, Hinterwälder, Hutter, Paar, Sánchez, and Schwabe [7]; our implementation is also smaller, and requires less RAM.

We use a recent result by Chung, Costello, and Smith [8] to also set new speed records for 128-bit secure signatures. Specifically, we present a new signature scheme based on fast Kummer surface arithmetic. It is inspired by the EdDSA construction by Bernstein, Duif, Lange, Schwabe, and Yang [9]. On the ATmega, it produces shorter signatures, achieves higher speeds and needs less RAM than the Ed25519 implementation presented in [10].

	ATmega		Cortex M0	
	Cycles	Stack bytes	Cycles	Stack bytes
`keygen`	10 206 181	812	2 774 087	1 056
`sign`	10 404 033	926	2 865 351	1 360
`verify`	16 240 510	992	4 453 978	1 432
`dh_exchange`	9 739 059	429	2 644 604	584

Table 1. Cycle counts and stack usage in bytes of all functions related to the signature and key exchange schemes, for the AVR ATmega and ARM Cortex M0 microcontrollers.

Our routines handling secret data are constant-time, and are thus naturally resistant to timing attacks. These algorithms are built around the Montgomery ladder, which improves resistance against simple-power-analysis (SPA) attacks. Resistance to DPA attacks can easily be added to the implementation by randomizing the scalar and/or Jacobian points. Re-randomizing the latter after each ladder step would also guarantee resistance against horizontal types of attacks.

Source code. We place all of the software described in this paper into the public domain, to maximize the reuseability of our results. The software is available at `http://www.cs.ru.nl/~jrenes/`.

2 High-level overview

We begin by describing the details of our signature and Diffie–Hellman schemes, explaining the choices we made in their design. Concrete implementation details appear in §3 and §4 below. Experimental results and comparisons follow in §5.

2.1 Signatures

Our signature scheme, defined at the end of this section, adheres closely to the proposal of [8, §8], which in turn is a type of Schnorr signature [11]. There are however some differences and trade-offs, which we discuss below.

Group structure. We build the signature scheme on top of the group structure from the Jacobian $\mathcal{J}_{\mathcal{C}}(\mathbb{F}_q)$ of a genus-2 hyperelliptic curve $\mathcal{C}$. More specifically, $\mathcal{C}$ is the Gaudry–Schost curve over the prime field $\mathbb{F}_q$ with $q = 2^{127} - 1$ (cf. §3.2). The Jacobian is a group of order $\#\mathcal{J}_{\mathcal{C}}(\mathbb{F}_q) = 2^4 N$, where

$$N = 2^{250} - \texttt{0x334D69820C75294D2C27FC9F9A154FF47730B4B840C05BD}$$

is a 250-bit prime. For more details on the Jacobian and its elements, see §3.3.

Hash function. We may use any hash function H with a 128-bit security level. For our purposes, $H(M) = \texttt{SHAKE128}(M, 512)$ suffices [12]. While `SHAKE128` has variable-length output, we only use the 512-bit output implementation.

Encoding. At the highest level, we operate on points Q in $\mathcal{J}_{\mathcal{C}}(\mathbb{F}_q)$. To minimize communication costs, we compress the usual 508-bit representation of Q into a 256-bit encoding $\underline{Q}$ (see §3.3). (This notation is the same as in [9].)

Public generator. The public generator can be any element P of $\mathcal{J}_{\mathcal{C}}(\mathbb{F}_q)$ such that $[N]P = 0$. In our implementation we have made the arbitrary choice $P = (X^2 + u_1 X + u_0, v_1 X + v_0)$, where

$$u_1 = \texttt{0x7D5D9C3307E959BF27B8C76211D35E8A}, \quad u_0 = \texttt{0x2703150F9C594E0CA7E8302F93079CE8},$$
$$v_1 = \texttt{0x444569AF177A9C1C721736D8F288C942}, \quad v_0 = \texttt{0x7F26CFB225F42417316836CFF8AEFB11}.$$

This is the point which we use the most for scalar multiplication. Since it remains fixed, we assume we have its decompressed representation precomputed, so as to avoid having to perform the relatively expensive decompression operation whenever we need a scalar multiplication; this gives a low-cost speed gain. We further assume we have a "wrapped" representation of the projection of P to the Kummer surface, which is used to speed up the `xDBLADD` function. See §4.1 for more details on the `xWRAP` function.

Public keys. In contrast to the public generator, we assume public keys are compressed: they are communicated much more frequently, and we therefore benefit much more from smaller keys. Moreover, we include the public key in one of the hashes during the `sign` operation [13,14], computing $h = H(\underline{R}||\underline{Q}||M)$ instead of the $h = H(\underline{R}||M)$ originally suggested by Schnorr [11]. This protects against adversaries attacking multiple public keys simultaneously.

Compressed signatures. Schnorr [11] mentions the option of compressing signatures by hashing one of their two components: the hash size only needs to be $b/2$ bits, where b is the key length. Following this suggestion, our signatures are 384-bit values of the form $(h_{128}||s)$, where h_{128} means the lowest 128 bits of $h = H(\underline{R}||\underline{Q}||M)$, and s is a 256-bit scalar. The most obvious upside is that signatures are smaller, reducing communication overhead. Another big advantage is that we can exploit the half-size scalar to speed up signature verification. On the other hand, we lose the possibility of efficient batch verification.

Verification efficiency. The most costly operation in signature verification is the two-dimensional scalar multiplication $T = [s]P \oplus [h_{128}]Q$. In [8], the authors propose an algorithm relying on the differential addition chains presented in [15]. However, since we are using compressed signatures, we have a small scalar h_{128}. Unfortunately the two-dimensional algorithm in [8] cannot directly exploit this fact, therefore not obtaining much benefit from the compressed signature. On the other hand, we can simply compute $[s]P$ and $[h_{128}]Q$ separately using the fast scalar multiplication on the Kummer surface and finally add them together on the Jacobian. Here $[s]P$ is a 256-bit scalar multiplication, whereas $[h_{128}]Q$ is only a 128-bit scalar multiplication. Not only do we need fewer cycles compared to the two-dimensional routine, but we also reduce code size by reusing the one-dimensional scalar multiplication routine.

The scheme. We now define our signature scheme, taking the above into account.

Key generation (`keygen`). Let d be a 256-bit secret key, and P the public generator. Compute $(d'||d'') \leftarrow H(d)$ (with d' and d'' both 256 bits), then $Q \leftarrow [16d']P$. The public key is $\underline{Q}$.

Signing (`sign`). Let M be a message, d a 256-bit secret key, P the public generator, and $\underline{Q}$ a compressed public key. Compute $(d'||d'') \leftarrow H(d)$ (with d' and d'' both 256 bits), then $r \leftarrow H(d''||M)$, then $R \leftarrow [r]P$, then $h \leftarrow H(\underline{R}||\underline{Q}||M)$, and finally $s \leftarrow (r - 16h_{128}d') \bmod N$. The signature is $(h_{128}||s)$.

Verification (`verify`). Let M be a message with a signature $(h_{128}||s)$ corresponding to a public key $\underline{Q}$, and let P be the public generator. Compute $T \leftarrow [s]P \oplus [h_{128}]Q$, then $g \leftarrow H(\underline{T}||\underline{Q}||M)$. The signature is correct if $g_{128} = h_{128}$, and incorrect otherwise.

Remark 1. We note that there may be faster algorithms to compute the "one-and-a-half-dimensional" scalar multiplication in `verify`, especially since we do not have to worry about being constant-time. One option might be to adapt

Montgomery's PRAC [16, §3.3.1] to make use of the half-size scalar. But while this may lead to a speed-up, it would also cause an increase in code size compared to simply re-using the one-dimensional scalar multiplication. We have chosen not to pursue this line, preferring the solid benefits of reduced code size instead.

2.2 Diffie-Hellman key exchange.

For key exchange it is not necessary to have a group structure; it is enough to have a pseudo-multiplication. We can therefore carry out our the key exchange directly on the Kummer surface $\mathcal{K}_{\mathcal{C}} = \mathcal{J}_{\mathcal{C}}/\langle\pm\rangle$, gaining efficiency by not projecting from and recovering to the Jacobian $\mathcal{J}_{\mathcal{C}}$. If Q is a point on $\mathcal{J}_{\mathcal{C}}$, then its image in $\mathcal{K}_{\mathcal{C}}$ is $\pm Q$. The common representation for points in $\mathcal{K}_{\mathcal{C}}(\mathbb{F}_q)$ is a 512-bit 4-tuple of field elements. For input points (i. e. the generator or public keys), we prefer the 384-bit "wrapped" representation (see §3.5). This not only reduces key size, but it also allows a speed-up in the core `xDBLADD` subroutine. The wrapped representation of a point $\pm Q$ on $\mathcal{K}_{\mathcal{C}}$ is denoted by $\underline{\pm Q}$.

Key exchange (`dh_exchange`). Let d be a 256-bit secret key, and $\underline{\pm P}$ the public generator (respectively public key). Compute $\pm Q \leftarrow \pm[d]P$. The generated public key (respectively shared secret) is $\underline{\pm Q}$.

Remark 2. While it might be possible to reduce the key size even further to 256 bits, we would then have to pay the cost of compressing and decompressing, and also wrapping for `xDBLADD` (see the discussion in [8, App. A]). We therefore choose to keep the 384-bit representation, which is consistent with [3].

3 Building blocks: algorithms and their implementation

We begin by presenting the finite field $\mathbb{F}_{2^{127}-1}$ in §3.1. We then define the curve $\mathcal{C}$ in §3.2, before giving basic methods for the elements of $\mathcal{J}_{\mathcal{C}}$ in §3.3. We then present the fast Kummer $\mathcal{K}_{\mathcal{C}}$ and its differential addition operations in §3.4.

3.1 The field $\mathbb{F}_q$

We work over the prime finite field $\mathbb{F}_q$, where q is the Mersenne prime

$$q := 2^{127} - 1 \ .$$

We let **M**, **S**, **a**, **s**, **neg**, and **I** denote the costs of multiplication, squaring, addition, subtraction, negation, and inversion in $\mathbb{F}_q$. Later, we will define a special operation for multiplying by small constants: its cost is denoted by $\mathbf{m_c}$.

For complete field arithmetic we implement modular reduction, addition, subtraction, multiplication, and inversion. We comment on some important aspects here, giving cycle counts in Table 2.

We can represent elements of $\mathbb{F}_q$ as 127-bit values; but since the ATmega and Cortex M0 work with 8- and 32-bit words, respectively, the obvious choice is to represent field elements with 128 bits. That is, an element $g \in \mathbb{F}_q$ is represented as $g = \sum_{i=0}^{15} g_i 2^{8i}$ on the AVR ATmega platform and as $g = \sum_{i=0}^{3} g'_i 2^{32i}$ on the Cortex M0, where $g_i \in \{0, \ldots, 2^8 - 1\}$, $g'_i \in \{0, \ldots, 2^{32} - 1\}$.

Working with the prime field $\mathbb{F}_q$, we need integer reduction modulo q; this is implemented as `bigint_red`. Reduction is very efficient because $2^{128} \equiv 2 \bmod q$, which enables us to reduce using only shifts and integer additions. Given this reduction, we implement addition and subtraction operations for $\mathbb{F}_q$ (as `gfe_add` and `gfe_sub`, respectively) in the obvious way.

The most costly operations in $\mathbb{F}_q$ are multiplication (`gfe_mul`) and squaring (`gfe_sqr`), which are implemented as 128×128-bit bit integer operations (`bigint_mul` and `bigint_sqr`) followed by a call to `bigint_red`. Since we are working on the same platforms as [7] in which both of these operations are already highly optimized, we took the necessary code from those implementations:

– On the AVR ATmega: The authors of [17] implement a 3-level Karatsuba multiplication of two 256-bit integers, representing elements f of $\mathbb{F}_{2^{255}-19}$ as $f = \sum_{i=0}^{31} f_i 2^{8i}$ with $f_i \in \{0, \ldots, 2^8 - 1\}$. Since the first level of Karatsuba relies on a 128×128-bit integer multiplication routine named `MUL128`, we simply lift this function out to form a 2-level 128×128-bit Karatsuba multiplication. Similarly, their 256×256-bit squaring relies on a 128×128-bit routine `SQR128`, which we can (almost) directly use. Since the 256×256-bit squaring is 2-level Karatsuba, the 128×128-bit squaring is 1-level Karatsuba.

– On the ARM Cortex M0: The authors of [7] use optimized Karatsuba multiplication and squaring. Their assembly code does not use subroutines, but fully inlines 128×128-bit multiplication and squaring. The 256×256-bit multiplication and squaring are both 3-level Karatsuba implementations. Hence, using these, we end up with 2-level 128×128-bit Karatsuba multiplication and squaring.

The function `gfe_invert` computes inversions in $\mathbb{F}_q$ as exponentiations, using the fact that $g^{-1} = g^{q-2}$ for all g in $\mathbb{F}_q^{\times}$. To do this efficiently we use an addition chain for $q - 2$, doing the exponentiation in $10\mathbf{M} + 126\mathbf{S}$.

Finally, to speed up our Jacobian point decompression algorithms, we define a function `gfe_powminhalf` which computes $g \mapsto g^{-1/2}$ for g in $\mathbb{F}_q$ (up to a choice of sign). To do this, we note that $g^{-1/2} = \pm g^{-(q+1)/4} = \pm g^{(3q-5)/4}$ in $\mathbb{F}_q$; this exponentiation can be done with an addition chain of length 136, using $11\mathbf{M} + 125\mathbf{S}$. We can then define a function `gfe_sqrtinv`, which given (x, y) and a bit b, computes $(\sqrt{x}, 1/y)$ as $(\pm xyz, xyz^2)$ where $z = \texttt{gfe\_powminhalf}(xy^2)$, choosing the sign so that the square root has least significant bit b. Including the `gfe_powminhalf` call, this costs $15\mathbf{M} + 126\mathbf{S} + 1\mathbf{neg}$.

3.2 The curve $\mathcal{C}$ and its theta constants

We define the curve $\mathcal{C}$ "backwards", starting from its (squared) theta constants

$$a := -11 , \quad b := 22 , \quad c := 19 , \quad \text{and} \quad d := 3 \quad \text{in } \mathbb{F}_q .$$

	AVR ATmega	ARM Cortex M0	Symbolic cost
`bigint_mul`	1 654	410	
`bigint_sqr`	1 171	260	
`bigint_red`	438	71	
`gfe_mul`	1 952	502	$\mathbf{M}$
`gfe_sqr`	1469	353	$\mathbf{S}$
`gfe_mulconst`	569	83	$\mathbf{m_c}$
`gfe_add`	400	62	$\mathbf{a}$
`gfe_sub`	401	66	$\mathbf{s}$
`gfe_invert`	169 881	46 091	$\mathbf{I}$
`gfe_powminhalf`	169 881	46 294	$11\mathbf{M} + 125\mathbf{S}$
`gfe_sqrtinv`	178 041	48 593	$15\mathbf{M} + 126\mathbf{S} + 1\mathbf{neg}$

Table 2. Cycle counts for our field implementation (including function-call overhead).

From these, we define the dual theta constants

$$A := a + b + c + d = 33 \ , \qquad B := a + b - c - d = -11 \ ,$$
$$C := a - b + c - d = -17 \ , \qquad D := a - b - c + d = -49 \ .$$

Observe that projectively,

$$(1/a : 1/b : 1/c : 1/d) = (114 : -57 : -66 : -418) \ ,$$
$$(1/A : 1/B : 1/C : 1/D) = (-833 : 2499 : 1617 : 561) \ .$$

Crucially, all of these constants can be represented using just 16 bits each. Since Kummer arithmetic involves many multiplications by these constants, we implement a separate 16×128-bit multiplication function `gfe_mulconst`. For the AVR ATmega, we store the constants in two 8-bit registers. For the Cortex M0, the values fit into a halfword; this works well with the 16×16-bit multiplication. Multiplication by any of these 16-bit constants costs $\mathbf{m_c}$.

Continuing, we define $e/f := (1 + \alpha)/(1 - \alpha)$, where $\alpha^2 = CD/AB$ (we take the square root with least significant bit 0), and thus

$$\lambda := ac/bd = \texttt{0x155555555555555555555555555555552} \ ,$$
$$\mu := ce/df = \texttt{0x73E334FBB315130E05A505C31919A746} \ ,$$
$$\nu := ae/bf = \texttt{0x552AB1B63BF799716B5806482D2D21F3} \ .$$

These are the *Rosenhain invariants* of the curve $\mathcal{C}$, found by Gaudry and Schost [18], which we are (finally!) ready to define as

$$\mathcal{C} : Y^2 = f_{\mathcal{C}}(X) := X(X - 1)(X - \lambda)(X - \mu)(X - \nu) \ .$$

The curve constants are the coefficients of $f_{\mathcal{C}}(X) = \sum_{i=0}^{5} f_i X^i$: so $f_0 = 0$, $f_5 = 1$,

$$f_1 = \texttt{0x1EDD6EE48E0C2F16F537CD791E4A8D6E} \ , \qquad f_2 = \texttt{0x73E799E36D9FCC210C9CD1B164C39A35} \ ,$$
$$f_3 = \texttt{0x4B9E333F48B6069CC47DC236188DF6E8} \ , \qquad f_4 = \texttt{0x219CC3F8BB9DFE2B39AD9E9F6463E172} \ .$$

We store the squared theta constants $(a : b : c : d)$, along with $(1/a : 1/b : 1/c : 1/d)$, and $(1/A : 1/B : 1/C : 1/D)$; the Rosenhain invariants λ, μ, and ν, together with $\lambda\mu$ and $\lambda\nu$; and the curve constants f_1, f_2, f_3, and f_4, for use in our Kummer and Jacobian arithmetic functions. Obviously, none of the Rosenhain or curve constants are small; multiplying by these costs a full **M**.

3.3 Elements of $\mathcal{J}_{\mathcal{C}}$, compressed and decompressed.

Our algorithms use the usual Mumford representation for elements of $\mathcal{J}_{\mathcal{C}}(\mathbb{F}_q)$: they correspond to pairs $\langle u(X), v(X)\rangle$, where u and v are polynomials over $\mathbb{F}_q$ with u monic, $\deg v < \deg u \leq 2$, and $v(X)^2 \equiv f_{\mathcal{C}}(X) \pmod{u(X)}$. We compute the group operation $\oplus$ in $\mathcal{J}_{\mathcal{C}}(\mathbb{F}_q)$ using a function `ADD`, which implements the algorithm found in [19] (after a change of coordinates to meet their Assumption 1)[3] at a cost of 28**M** + 2**S** + 11**a** + 24**s** + 1**I**.

For transmission, we compress the 508-bit Mumford representation to a 256-bit form. Our functions `compress` (Algorithm 1) and `decompress` (Algorithm 2) implement Stahlke's compression technique (see [20] and [8, App. A] for details).

Algorithm 1: `compress`: compresses points on $\mathcal{J}_{\mathcal{C}}$ to 256-bit strings. Symbolic cost: 3**M** + 1**S** + 2**a** + 2**s**. ATmega: 8 016 cycles. Cortex M0: 2 186 cycles.

```
Input: $\langle X^2 + u_1X + u_0, v_1X + v_0\rangle = P \in \mathcal{J}_{\mathcal{C}}$.
Output: A string $b_0 \cdots b_{255}$ of 256 bits.
w $\leftarrow 4((u_1 \cdot v_0 - u_0 \cdot v_1) \cdot v_1 - v_0^2)$      // 3M + 1S + 2a + 2s
$\mathsf{b}_0 \leftarrow$ LeastSignificantBit($v_1$)
$\mathsf{b}_{128} \leftarrow$ LeastSignificantBit(w)
return $\mathsf{b}_0 || u_0 || \mathsf{b}_{128} || u_1$
```

3.4 The Kummer surface $\mathcal{K}_{\mathcal{C}}$

The Kummer surface of $\mathcal{C}$ is the quotient $\mathcal{K}_{\mathcal{C}} := \mathcal{J}_{\mathcal{C}}/\langle\pm 1\rangle$; points on $\mathcal{K}_{\mathcal{C}}$ correspond to points on $\mathcal{J}_{\mathcal{C}}$ taken up to sign. If P is a point in $\mathcal{J}_{\mathcal{C}}$, then we write

$$(x_P : y_P : z_P : t_P) = \pm P$$

for its image in $\mathcal{K}_{\mathcal{C}}$. To avoid subscript explosion, we make the following convention: when points P and Q on $\mathcal{J}_{\mathcal{C}}$ are clear from the context, we write

$$(x_\oplus : y_\oplus : z_\oplus : t_\oplus) = \pm(P \oplus Q) \quad \text{and} \quad (x_\ominus : y_\ominus : z_\ominus : t_\ominus) = \pm(P \ominus Q) \ .$$

[3] We only call `ADD` once in our algorithms, so for lack of space we omit its description.

Algorithm 2: `decompress`: decompresses 256-bit string to a point on $\mathcal{J}_{\mathcal{C}}$. Symbolic cost: $46\mathbf{M} + 255\mathbf{S} + 17\mathbf{a} + 12\mathbf{s} + 6\mathbf{neg}$. ATmega: 386 524 cycles Cortex M0: 106 013 cycles

```
Input: A string $b_0 \cdots b_{255}$ of 256 bits.
Output: $\langle X^2 + u_1 X + u_0, v_1 X + v_0 \rangle = P \in \mathcal{J}_{\mathcal{C}}$.
$\mathsf{U}_1 = b_{129} \cdots b_{256}$ as an element of $\mathbb{F}_q$
$\mathsf{U}_0 = b_1 \cdots b_{127}$ as an element of $\mathbb{F}_q$
$\mathsf{T}_1 \leftarrow \mathsf{U}_1^2$                                                    // 1S
$\mathsf{T}_2 \leftarrow \mathsf{U}_0 - \mathsf{T}_1$                                                 // 1s
$\mathsf{T}_3 \leftarrow \mathsf{U}_0 + \mathsf{T}_2$                                                 // 1a
$\mathsf{T}_4 \leftarrow \mathsf{U}_0 \cdot (\mathsf{T}_3 \cdot f_4 + (\mathsf{U}_1 \cdot f_3 - 2f_2))$                       // 3M + 1a + 2s
$\mathsf{T}_3 \leftarrow -\mathsf{T}_3$                                                    // 1neg
$\mathsf{T}_1 \leftarrow \mathsf{T}_3 - \mathsf{U}_0$                                                 // 1s
$\mathsf{T}_4 \leftarrow 2(\mathsf{T}_4 + (\mathsf{T}_1 \cdot \mathsf{U}_0 + f_1) \cdot \mathsf{U}_1)$                          // 2M + 3a
$\mathsf{T}_1 \leftarrow 2(\mathsf{T}_1 - \mathsf{U}_0))$                                             // 1a + 1s
$\mathsf{T}_5 \leftarrow ((\mathsf{U}_0 - (f_3 + \mathsf{U}_1 \cdot (\mathsf{U}_1 - f_4))) \cdot \mathsf{U}_0 + f_1)^2$         // 2M + 1S + 2a + 2s
$\mathsf{T}_5 \leftarrow \mathsf{T}_4^2 - 2\mathsf{T}_5 \cdot \mathsf{T}_1$                                     // 1M + 1S + 1a + 1s
$(\mathsf{T}_6, \mathsf{T}_5) \leftarrow$ gfe_sqrtinv$(\mathsf{T}_5, \mathsf{T}_1, b_1)$                   // 19M + 127S + 2neg
$\mathsf{T}_4 \leftarrow (\mathsf{T}_5 - \mathsf{T}_4) \cdot \mathsf{T}_6$                                         // 1M + 1s
$\mathsf{T}_5 \leftarrow -f_4 \cdot \mathsf{T}_2 - ((\mathsf{T}_3 - f_3) \cdot \mathsf{U}_1) + f_2 + \mathsf{T}_4$          // 2M + 2s + 2a + 1neg
$\mathsf{T}_6 =$ gfe_powminhalf$(4\mathsf{T}_6)$                  // $= 1/(2v_1)$. 11M + 125S + 2a
$\mathsf{V}_1 \leftarrow 2\mathsf{T}_5 \cdot \mathsf{T}_6$                                              // 1M + 1a
if $b_0 \neq$ LeastSignificantBit$(\mathsf{V}_1)$ then $(\mathsf{V}_1, \mathsf{T}_6) \leftarrow (-\mathsf{V}_1, -\mathsf{T}_6)$      // 2neg
$\mathsf{T}_5 \leftarrow (\mathsf{U}_1 \cdot f_4 + (\mathsf{T}_2 - f_3)) \cdot \mathsf{U}_0$                            // 2M + 1a + 1s
$\mathsf{V}_0 \leftarrow (\mathsf{U}_1 \cdot \mathsf{T}_4 + \mathsf{T}_5 + f_1) \cdot \mathsf{T}_6$                              // 2M + 2a
return $\langle X^2 + \mathsf{U}_1 X + \mathsf{U}_0, \mathsf{V}_1 X + \mathsf{V}_0 \rangle$
```

The Kummer surface of this $\mathcal{C}$ has a “fast” model in $\mathbb{P}^3$ defined by

$$\mathcal{K}_{\mathcal{C}} : E \cdot xyzt = \left(\begin{array}{c} (x^2 + y^2 + z^2 + t^2) \\ -F \cdot (xt + yz) - G \cdot (xz + yt) - H \cdot (xy + zt) \end{array} \right)^2$$

where

$$F = \frac{a^2 - b^2 - c^2 + d^2}{ad - bc} \ , \quad G = \frac{a^2 - b^2 + c^2 - d^2}{ac - bd} \ , \quad H = \frac{a^2 + b^2 - c^2 - d^2}{ab - cd} \ ,$$

and $E = 4abcd\left(ABCD/((ad - bc)(ac - bd)(ab - cd))\right)^2$ (see eg. [21], [22], and [23]). The identity point $\langle 1, 0 \rangle$ of $\mathcal{J}_{\mathcal{C}}$ maps to

$$\pm 0_{\mathcal{J}_{\mathcal{C}}} = (a : b : c : d) \ .$$

Algorithm 3 (`Project`) maps general points from $\mathcal{J}_{\mathcal{C}}(\mathbb{F}_q)$ into $\mathcal{K}_{\mathcal{C}}$. The “special” case where u is linear is treated in [8, §7.2]; this is not implemented, since `Project` only operates on public generators and keys, none of which are special.

Algorithm 3: `Project`: $\mathcal{J}_{\mathcal{C}} \to \mathcal{K}_{\mathcal{C}}$. Symbolic cost: $8\mathbf{M} + 1\mathbf{S} + 4\mathbf{m_c} + 7\mathbf{a} + 4\mathbf{s}$. ATmega: 20 205 cycles. Cortex M0: 5 667 cycles.

```
Input: $\langle X^2 + u_1X + u_0, v_1X + v_0 \rangle = P \in \mathcal{J}_{\mathcal{C}}$.
Output: $(x_P : y_P : z_P : t_P) = \pm P \in \mathcal{K}_{\mathcal{C}}$.
$(\mathtt{T}_1, \mathtt{T}_2, \mathtt{T}_3, \mathtt{T}_4) \leftarrow (\mu - u_0, \lambda\nu - u_0, \nu - u_0, \lambda\mu - u_0)$                 // 4s
$\mathtt{T}_5 \leftarrow \lambda + u_1$                                                                  // 1a
$\mathtt{T}_7 \leftarrow u_0 \cdot ((\mathtt{T}_5 + \mu) \cdot \mathtt{T}_3)$                              // 2M + 1a
$\mathtt{T}_5 \leftarrow u_0 \cdot ((\mathtt{T}_5 + \nu) \cdot \mathtt{T}_1)$                              // 2M + 1a
$(\mathtt{T}_6, \mathtt{T}_8) \leftarrow (u_0 \cdot ((\mu + u_1) \cdot \mathtt{T}_2 + \mathtt{T}_2), u_0 \cdot ((\nu + u_1) \cdot \mathtt{T}_4 + \mathtt{T}_4))$   // 4M + 4a
$\mathtt{T}_1 \leftarrow v_0^2$                                                                          // 1S
$(\mathtt{T}_5, \mathtt{T}_6, \mathtt{T}_7, \mathtt{T}_8) \leftarrow (\mathtt{T}_5 - \mathtt{T}_1, \mathtt{T}_6 - \mathtt{T}_1, \mathtt{T}_7 - \mathtt{T}_1, \mathtt{T}_8 - \mathtt{T}_1)$   // 4s
return $(a \cdot \mathtt{T}_5 : b \cdot \mathtt{T}_6 : c \cdot \mathtt{T}_7 : d \cdot \mathtt{T}_8)$                    // 4m_c
```

3.5 Pseudo-addition on $\mathcal{K}_{\mathcal{C}}$.

While the points of $\mathcal{K}_{\mathcal{C}}$ do not form a group, we have a pseudo-addition operation (differential addition), which computes $\pm(P \oplus Q)$ from $\pm P$, $\pm Q$, and $\pm(P \ominus Q)$. The function `xADD` (Algorithm 4) implements the standard differential addition. The special case where $P = Q$ yields a pseudo-doubling operation.

To simplify the presentation of our algorithms, we define three operations on points in $\mathbb{P}^3$. First, $\mathcal{M} : \mathbb{P}^3 \times \mathbb{P}^3 \to \mathbb{P}^3$ multiplies corresponding coordinates:

$$\mathcal{M} : ((x_1 : y_1 : z_1 : t_1), (x_2 : y_2 : z_2 : t_2)) \longmapsto (x_1x_2 : y_1y_2 : z_1z_2 : t_1t_2) \ .$$

The special case $(x_1 : y_1 : z_1 : t_1) = (x_2 : y_2 : z_2 : t_2)$ is denoted by

$$\mathcal{S} : (x : y : z : t) \longmapsto (x^2 : y^2 : z^2 : t^2) \ .$$

Finally, the Hadamard transform[4] is defined by

$$\mathcal{H} : (x : y : z : t) \longmapsto (x' : y' : z' : t') \quad \text{where} \quad \begin{cases} x' = x + y + z + t \ , \\ y' = x + y - z - t \ , \\ z' = x - y + z - t \ , \\ t' = x - y - z + t \ . \end{cases}$$

Clearly $\mathcal{M}$ and $\mathcal{S}$ cost $4\mathbf{M}$ and $4\mathbf{S}$, respectively. The Hadamard transform can easily be implemented with $4\mathbf{a} + 4\mathbf{s}$. However, the additions and subtractions are relatively cheap, making function call overhead a large factor. To minimize this we inline the Hadamard transform, trading a bit of code size for efficiency.

Algorithm 4: `xADD`: Differential addition on $\mathcal{K}_{\mathcal{C}}$. Symbolic cost: $14\mathbf{M} + 4\mathbf{S} + 4\mathbf{m_c} + 12\mathbf{a} + 12\mathbf{s}$. ATmega: 34 774 cycles. Cortex M0: 9 598 cycles.

```
  Input: $(\pm P, \pm Q, \pm(P \ominus Q)) \in \mathcal{K}_{\mathcal{C}}^3$ for some $P$ and $Q$ on $\mathcal{J}_{\mathcal{C}}$.
  Output: $\pm(P \oplus Q) \in \mathcal{K}_{\mathcal{C}}$.
1 $(\mathsf{V}_1, \mathsf{V}_2) \leftarrow (\mathcal{H}(\pm P), \mathcal{H}(\pm Q))$                 // 8a + 8s
2 $\mathsf{V}_1 \leftarrow \mathcal{M}(\mathsf{V}_1, \mathsf{V}_2)$                                  // 4M
3 $\mathsf{V}_1 \leftarrow \mathcal{M}(\mathsf{V}_1, (1/A : 1/B : 1/C : 1/D))$              // $4\mathbf{m_c}$
4 $\mathsf{V}_1 \leftarrow \mathcal{H}(\mathsf{V}_1)$                                         // 4a + 4s
5 $\mathsf{V}_1 \leftarrow \mathcal{S}(\mathsf{V}_1)$                                             // 4S
6 $(\mathsf{C}_1, \mathsf{C}_2) \leftarrow (z_\ominus \cdot t_\ominus, x_\ominus \cdot y_\ominus)$                          // 2M
7 $\mathsf{V}_2 \leftarrow \mathcal{M}((\mathsf{C}_1 : \mathsf{C}_1 : \mathsf{C}_2 : \mathsf{C}_2), (y_\ominus : x_\ominus : t_\ominus : z_\ominus))$   // 4M
8 return $\mathcal{M}(\mathsf{V}_1, \mathsf{V}_2)$                                        // 4M
```

Lines 5 and 6 of Algorithm 4 only involve the third argument, $\pm(P \ominus Q)$; essentially, they compute the point $(y_\ominus z_\ominus t_\ominus : x_\ominus z_\ominus t_\ominus : x_\ominus y_\ominus t_\ominus : x_\ominus y_\ominus z_\ominus)$ (which is projectively equivalent to $(1/x_\ominus : 1/y_\ominus : 1/z_\ominus : 1/t_\ominus)$, but requires no inversions; note that this is generally *not* a point on $\mathcal{K}_{\mathcal{C}}$). In practice, the pseudoadditions used in our scalar multiplication all use a fixed third argument, so it makes sense to precompute this "inverted" point and to scale it by $x_\ominus$ so that the first coordinate is 1, thus saving $7\mathbf{M}$ in each subsequent differential addition for a one-off cost of $1\mathbf{I}$. The resulting data can be stored as the 3-tuple $(x_\ominus/y_\ominus, x_\ominus/z_\ominus, x_\ominus/t_\ominus)$, ignoring the trivial first coordinate: this is the *wrapped* form of $\pm(P \ominus Q)$. The function `xWRAP` (Algorithm 5) applies this transformation.

Algorithm 6 combines the pseudo-doubling with the differential addition, sharing intermediate operands, to define a differential double-and-add `xDBLADD`. This is the fundamental building block of the Montgomery ladder.

[4] Note that $(A : B : C : D) = \mathcal{H}((a : b : c : d))$ and $(a : b : c : d) = \mathcal{H}((A : B : C : D))$.

Algorithm 5: `xWRAP`: $(x : y : z : t) \mapsto (x/y, x/z, x/t)$. Symbolic cost: $7\mathbf{M} + 1\mathbf{I}$ ATmega: 182 251 cycles. Cortex M0: 49 609 cycles.

```
**Input**: $(x : y : z : t) \in \mathbb{P}^3$
**Output**: $(x/y, x/z, x/t) \in \mathbb{F}_q^3$.
$\mathsf{V}_1 \leftarrow y \cdot z$                                        // 1M
$\mathsf{V}_2 \leftarrow x/(\mathsf{V}_1 \cdot t)$                                 // 2M + 1I
$\mathsf{V}_3 \leftarrow \mathsf{V}_2 \cdot t$                                       // 1M
**return** $(\mathsf{V}_3 \cdot z, \mathsf{V}_3 \cdot y, \mathsf{V}_1 \cdot \mathsf{V}_2)$                      // 3M
```

Algorithm 6: `xDBLADD`: Combined differential double-and-add. The difference point is wrapped. Symbolic cost: $7\mathbf{M} + 12\mathbf{S} + 12\mathbf{m_c} + 16\mathbf{a} + 16\mathbf{s}$. ATmega: 36 706 cycles. Cortex M0: 9 861 cycles.

```
**Input**: $(\pm P, \pm Q, (x_\ominus/y_\ominus, x_\ominus/z_\ominus, x_\ominus/t_\ominus)) \in \mathcal{K}_\mathcal{C}^2 \times \mathbb{F}_q$.
**Output**: $(\pm[2]P, \pm(P \oplus Q)) \in \mathcal{K}_\mathcal{C}^2$.
$(\mathsf{V}_1, \mathsf{V}_2) \leftarrow (\mathcal{S}(\pm P), \mathcal{S}(\pm Q))$                          // 8S
$(\mathsf{V}_1, \mathsf{V}_2) \leftarrow (\mathcal{H}(\mathsf{V}_1), \mathcal{H}(\mathsf{V}_2))$                        // 8a + 8s
$(\mathsf{V}_1, \mathsf{V}_2) \leftarrow (\mathcal{S}(\mathsf{V}_1), \mathcal{M}(\mathsf{V}_1, \mathsf{V}_2))$                     // 4M+4S
$(\mathsf{V}_1, \mathsf{V}_2) \leftarrow \big(\mathcal{M}(\mathsf{V}_1, (\frac{1}{A} : \frac{1}{B} : \frac{1}{C} : \frac{1}{D})), \mathcal{M}(\mathsf{V}_2, (\frac{1}{A} : \frac{1}{B} : \frac{1}{C} : \frac{1}{D}))\big)$   // $8\mathbf{m_c}$
$(\mathsf{V}_1, \mathsf{V}_2) \leftarrow (\mathcal{H}(\mathsf{V}_1), \mathcal{H}(\mathsf{V}_2))$                        // 8a + 8s
**return** $(\mathcal{M}(\mathsf{V}_1, (\frac{1}{a} : \frac{1}{b} : \frac{1}{c} : \frac{1}{d})), \mathcal{M}(\mathsf{V}_2, (1 : \frac{x_\ominus}{y_\ominus} : \frac{x_\ominus}{y_\ominus} : \frac{x_\ominus}{t_\ominus})))$   // 3M + $4\mathbf{m}_c$
```

	$\mathbf{M}$	$\mathbf{S}$	$\mathbf{m_c}$	$\mathbf{a}$	$\mathbf{s}$	**neg**	$\mathbf{I}$	ATmega	Cortex M0
`ADD`	28	2	0	11	24	0	1	228 552	62 886
`Project`	8	1	4	7	8	0	0	20 205	5 667
`xWRAP`	7	0	0	0	0	0	1	182 251	49 609
`xUNWRAP`	4	0	0	0	0	0	0	7 297	2 027
`xADD`	14	4	4	12	12	0	0	34 774	9 598
`xDBLADD`	7	12	12	16	16	0	0	36 706	9 861
`recoverGeneral`	77	8	0	19	10	3	1	318 910	88 414
`fast2genPartial`	11	0	0	9	0	0	0	21 339	6 110
`fast2genFull`	15	0	0	12	0	0	0	29 011	8 333
`recoverFast`	139	12	4	70	22	5	1	447 176	124 936
`compress`	3	1	0	2	2	0	0	8 016	2 186
`decompress`	46	255	0	17	12	6	0	386 524	106 013

Table 3. Operation and cycle counts of basic functions on the Kummer and Jacobian.

4 Scalar multiplication

All of our cryptographic routines are built around scalar multiplication in $\mathcal{J}_\mathcal{C}$ and pseudo-scalar multiplication in $\mathcal{K}_\mathcal{C}$. We implement pseudo-scalar multiplication using the classic Montgomery ladder in §4.1. In §4.2, we extend this to full scalar multiplication on $\mathcal{J}_\mathcal{C}$ using the point recovery technique proposed in [8].

4.1 Pseudomultiplication on $\mathcal{K}_{\mathcal{C}}$

Since $[m](\ominus P) = \ominus[m]P$ for all m and P, we have a pseudo-scalar multiplication operation $(m, \pm P) \longmapsto \pm[m]P$ on $\mathcal{K}_{\mathcal{C}}$, which we compute using Algorithm 7 (the Montgomery ladder), implemented as `crypto_scalarmult`. The loop of Algorithm 7 maintains the following invariant: at the end of iteration i we have

$$(V_1, V_2) = (\pm[k]P, \pm[k+1]P) \quad \text{where} \quad k = \textstyle\sum_{j=i}^{\beta-1} m_j 2^{\beta-1-i} \,.$$

Hence, at the end we return $\pm[m]P$, and also $\pm[m+1]P$ as a (free) byproduct. We suppose we have a constant-time conditional swap routine $\texttt{CSWAP}(b, (V_1, V_2))$, which returns (V_1, V_2) if $b = 0$ and (V_2, V_1) if $b = 1$. This makes the execution of Algorithm 7 uniform and constant-time, and thus suitable for use with secret m.

Algorithm 7: `crypto_scalarmult`: Montgomery ladder on $\mathcal{K}_{\mathcal{C}}$. Uniform and constant-time: may be used for secret scalars. The point is wrapped. Symbolic cost: $(4+7\beta)\mathbf{M} + 12\beta\mathbf{S} + 12\beta\mathbf{m_c} + 16\beta\mathbf{a} + 16\beta\mathbf{s}$, where β = scalar bitlength. ATmega: 9 513 536 cycles. Cortex: 2 633 662 cycles.

```
Input: $(m = \sum_{i=0}^{\beta-1} m_i 2^i, (x_P/y_P, x_P/z_P, x_P/t_P)) \in [0, 2^\beta) \times \mathbb{F}_q^3$ for $\pm P$ in $\mathcal{K}_{\mathcal{C}}$.
Output: $(\pm[m]P, \pm[m+1]P) \in \mathcal{K}_{\mathcal{C}}^2$.
$\mathsf{V}_1 \leftarrow (a : b : c : d)$
$\mathsf{V}_2 \leftarrow$ xUNWRAP$(x_P/y_P, x_P/z_P, x_P/t_P)$                 // $= \pm P$. 4M
for $i = 250$ down to $0$ do          // $7\beta$M + $12\beta$S + $12\beta m_c$ + $16\beta$a + $16\beta$s
    $(\mathsf{V}_1, \mathsf{V}_2) \leftarrow$ CSWAP$(m_i, (\mathsf{V}_1, \mathsf{V}_2))$
    $(\mathsf{V}_1, \mathsf{V}_2) \leftarrow$ xDBLADD$(\mathsf{V}_1, \mathsf{V}_2, (x_P/y_P, x_P/z_P, x_P/t_P))$
    $(\mathsf{V}_1, \mathsf{V}_2) \leftarrow$ CSWAP$(m_i, (\mathsf{V}_1, \mathsf{V}_2))$
return $(\mathsf{V}_1, \mathsf{V}_2)$
```

Our implementation of `crypto_scalarmult` assumes that its input Kummer point $\pm P$ is wrapped. This follows the approach of [3]. Indeed, many calls to `crypto_scalarmult` involve Kummer points that are stored or transmitted in wrapped form. However, `crypto_scalarmult` does require the unwrapped point internally—if only to initialize one variable. We therefore define a function `xUNWRAP` (Algorithm 8) to invert the `xWRAP` transformation at a cost of only 4**M**.

Algorithm 8: `xUNWRAP`: $(x/y, x/z, x/t) \mapsto (x : y : z : t)$. Symbolic cost: 4**M**. ATmega: 7 297 cycles. Cortex: 2 027 cycles.

```
Input: $(u, v, w) \in \mathbb{F}_q^3$ s.t. $u = x_P/y_P, v = x_P/z_P, w = x_P/t_P$ for $\pm P \in \mathcal{K}_{\mathcal{C}}$
Output: $(x_P : y_P : z_P : t_P) \in \mathbb{P}^3$
$(\mathsf{T}_1, \mathsf{T}_2, \mathsf{T}_3) \leftarrow (v \cdot w, u \cdot w, u \cdot v)$          // 3M
return $(\mathsf{T}_3 \cdot w : \mathsf{T}_1 : \mathsf{T}_2 : \mathsf{T}_3)$          // 1M
```

4.2 Point recovery from $\mathcal{K}_\mathcal{C}$ to $\mathcal{J}_\mathcal{C}$

Point recovery means efficiently computing $[m]P$ on $\mathcal{J}_\mathcal{C}$ given $\pm[m]P$ on $\mathcal{K}_\mathcal{C}$ and some additional information. In our case, the additional information is the base point P and the second output of the Montgomery ladder, $\pm[m+1]P$. Algorithm 9 (`Recover`) implements the point recovery described in [8]. This is the genus-2 analogue of the elliptic-curve methods in [24], [25], and [26].

Algorithm 9: `Recover`: From $\mathcal{K}_\mathcal{C}$ to $\mathcal{J}_\mathcal{C}$. Symbolic cost: $139\mathbf{M} + 12\mathbf{S} + 4\mathbf{m_c} + 70\mathbf{a} + 22\mathbf{s} + 3\mathbf{neg} + 1\mathbf{I}$. ATmega: 447 176 cycles. Cortex: 124 936 cycles.

```
Input: $(P, \pm P, \pm Q, \pm(P \oplus Q)) \in \mathcal{J}_\mathcal{C} \times \mathcal{K}_\mathcal{C}^3$ for some $P, Q$ in $\mathcal{J}_\mathcal{C}$.
Output: $Q \in \mathcal{J}_\mathcal{C}$.
gP ← fast2genPartial($\pm P$)                                // 11M + 9a
gQ ← fast2genFull($\pm Q$)                                   // 15M + 12a
gS ← fast2genPartial($\pm(P \oplus Q)$)                      // 11M + 9a
xD ← xADD($\pm P, \pm Q, \pm(P \oplus Q)$)     // 14M + 4S + 4m_c + 12a + 12s
gD ← fast2genPartial(xD)                                     // 11M + 9a
return recoverGeneral($P$, gP, gQ, gS, gD)     // 77M+8S+19a+10s+3neg+1I
```

We refer the reader to [8] for technical details on this method, but there is one important mathematical detail that we should mention (since it is reflected in the structure of our code): point recovery is more natural starting from the general Flynn model $\widetilde{\mathcal{K}}_\mathcal{C}$ of the Kummer, because it is more closely related to the Mumford model for $\mathcal{J}_\mathcal{C}$. Algorithm 9 therefore proceeds in two steps: first Algorithms 10 (`fast2genFull`) and 11 (`fast2genPartial`) map the problem into $\widetilde{\mathcal{K}}_\mathcal{C}$, and then we recover from $\widetilde{\mathcal{K}}_\mathcal{C}$ to $\mathcal{J}_\mathcal{C}$ using Algorithm 12 (`recoverGeneral`).

Since the general Kummer $\widetilde{\mathcal{K}}_\mathcal{C}$ only appears briefly in our recovery procedure (we never use its relatively slow arithmetic operations), we will not investigate it in detail here—but the curious reader may refer to [27] for the general theory. For our purposes, it suffices to recall that $\widetilde{\mathcal{K}}_\mathcal{C}$ is, like $\mathcal{K}_\mathcal{C}$, embedded in $\mathbb{P}^3$; and the isomorphism $\mathcal{K}_\mathcal{C} \to \widetilde{\mathcal{K}}_\mathcal{C}$ is defined (in eg. [8, §7.4]) by the linear transformation

$$(x_P : y_P : z_P : t_P) \longmapsto (\tilde{x}_P : \tilde{y}_P : \tilde{z}_P : \tilde{t}_P) := (x_P : y_P : z_P : t_P)L \ ,$$

where L is (any scalar multiple of) the matrix

$$\begin{pmatrix} a^{-1}(\nu-\lambda) & a^{-1}(\mu\nu-\lambda) & a^{-1}\lambda\nu(\mu-1) & a^{-1}\lambda\nu(\mu\nu-\lambda) \\ b^{-1}(\mu-1) & b^{-1}(\mu\nu-\lambda) & b^{-1}\mu(\nu-\lambda) & b^{-1}\mu(\mu\nu-\lambda) \\ c^{-1}(\lambda-\mu) & c^{-1}(\lambda-\mu\nu) & c^{-1}\lambda\mu(1-\nu) & c^{-1}\lambda\mu(\lambda-\mu\nu) \\ d^{-1}(1-\nu) & d^{-1}(\lambda-\mu\nu) & d^{-1}\nu(\lambda-\mu) & d^{-1}\nu(\lambda-\mu\nu) \end{pmatrix},$$

which we precompute and store. If $\pm P$ is a point on $\mathcal{K}_\mathcal{C}$, then $\widetilde{\pm P}$ denotes its image on $\widetilde{\mathcal{K}}_\mathcal{C}$; we compute $\widetilde{\pm P}$ using Algorithm 10 (`fast2genFull`).

Algorithm 10: `fast2genFull`: The map $\mathcal{K}_{\mathcal{C}} \to \widetilde{\mathcal{K}}_{\mathcal{C}}$. Symbolic cost: $15\mathbf{M}+12\mathbf{a}$. ATmega: 29 011 cycles. Cortex: 8 333 cycles.

```
Input: $\pm P \in \mathcal{K}_{\mathcal{C}}$.
Output: $\widetilde{\pm P} \in \widetilde{\mathcal{K}}_{\mathcal{C}}$.
$\tilde{x}_P \leftarrow x_P + (L_{12}/L_{11})y_P + (L_{13}/L_{11})z_P + (L_{14}/L_{11})t_P$                      // $3\mathbf{M}+3\mathbf{a}$
$\tilde{y}_P \leftarrow (L_{21}/L_{11})x_P + (L_{22}/L_{11})y_P + (L_{23}/L_{11})z_P + (L_{24}/L_{11})t_P$    // $4\mathbf{M}+3\mathbf{a}$
$\tilde{z}_P \leftarrow (L_{31}/L_{11})x_P + (L_{32}/L_{11})y_P + (L_{33}/L_{11})z_P + (L_{34}/L_{11})t_P$    // $4\mathbf{M}+3\mathbf{a}$
$\tilde{t}_P \leftarrow (L_{41}/L_{11})x_P + (L_{42}/L_{11})y_P + (L_{43}/L_{11})z_P + (L_{44}/L_{11})t_P$    // $4\mathbf{M}+3\mathbf{a}$
return $(\tilde{x}_P : \tilde{y}_P : \tilde{z}_P : \tilde{t}_P)$
```

Sometimes we only require the first three coordinates of $\widetilde{\pm P}$. Algorithm 11 (`fast2genPartial`) saves $4\mathbf{M} + 3\mathbf{a}$ per point by not computing $\tilde{t}_P$.

Algorithm 11: `fast2genPartial`: The map $\mathcal{K}_{\mathcal{C}} \to \mathbb{P}^2$. Symbolic cost: $11\mathbf{M} + 9\mathbf{a}$. ATmega: 21 339 cycles. Cortex: 8 333 cycles.

```
Input: $\pm P \in \mathcal{K}_{\mathcal{C}}$.
Output: $(\tilde{x}_P : \tilde{y}_P : \tilde{z}_P) \in \mathbb{P}^2$
$\tilde{x}_P \leftarrow x_P + (L_{12}/L_{11})y_P + (L_{13}/L_{11})z_P + (L_{14}/L_{11})t_P$                      // $3\mathbf{M}+3\mathbf{a}$
$\tilde{y}_P \leftarrow (L_{21}/L_{11})x_P + (L_{22}/L_{11})y_P + (L_{23}/L_{11})z_P + (L_{24}/L_{11})t_P$    // $4\mathbf{M}+3\mathbf{a}$
$\tilde{z}_P \leftarrow (L_{31}/L_{11})x_P + (L_{32}/L_{11})y_P + (L_{33}/L_{11})z_P + (L_{34}/L_{11})t_P$    // $4\mathbf{M}+3\mathbf{a}$
return $(\tilde{x}_P : \tilde{y}_P : \tilde{z}_P)$
```

4.3 Full scalar multiplication on $\mathcal{J}_{\mathcal{C}}$

We now combine our pseudo-scalar multiplication function `crypto_scalarmult` with the point-recovery function `Recover` to define a full scalar multiplication function `jacobian_scalarmult` (Algorithm 13) on $\mathcal{J}_{\mathcal{C}}$.

Remark 3. `jacobian_scalarmult` takes not only a scalar m and a Jacobian point P in its Mumford representation, but also the wrapped form of $\pm P$ as an auxiliary argument: that is, we assume that `xP` ← `Project`(P) and `xWRAP`(`xP`) have already been carried out. This saves redundant `Project` and `xWRAP` calls when operating on fixed base points, as is often the case in our protocols. Nevertheless, `jacobian_scalarmult` could easily be converted to a "pure" Jacobian scalar multiplication function (with no auxiliary input) by inserting appropriate `Project` and `xWRAP` calls at the start, and removing the `xUNWRAP` call at Line 2, increasing the total cost by $11\mathbf{M} + 1\mathbf{S} + 4\mathbf{m_c} + 7\mathbf{a} + 8\mathbf{s} + 1\mathbf{I}$.

5 Results and comparison

The high-level cryptographic functions for our signature scheme are named `keygen`, `sign` and `verify`. Their implementations contain no surprises: they

Algorithm 12: `recoverGeneral`: From $\widetilde{\mathcal{K}}_{\mathcal{C}}$ to $\mathcal{J}_{\mathcal{C}}$. Symbolic cost: $77\mathbf{M} + 8\mathbf{S} + 19\mathbf{a} + 10\mathbf{s} + 3\mathbf{neg} + 1\mathbf{I}$. ATmega: 318 910 cycles. Cortex: 88 414 cycles.

```
Input: $(P, \widetilde{\pm P}, \widetilde{\pm Q}, \widetilde{\pm(P\oplus Q)}, \widetilde{\pm(P\ominus Q)}) \in \mathcal{J}_{\mathcal{C}} \times \widetilde{\mathcal{K}}_{\mathcal{C}}^4$ for some $P$ and $Q$ in $\mathcal{J}_{\mathcal{C}}$.
    The values of $\tilde{t}_P$, $\tilde{t}_\oplus$, and $\tilde{t}_\ominus$ are not required.
Output: $Q \in \mathcal{J}_{\mathcal{C}}$.
$(\mathsf{Z1}, \mathsf{Z2}) \leftarrow (\tilde{y}_P \cdot \tilde{x}_Q - \tilde{x}_Q \cdot \tilde{y}_P, \tilde{x}_P \cdot \tilde{z}_Q - \tilde{z}_P \cdot \tilde{x}_Q)$    // 4M+2s
$\mathsf{T1} \leftarrow \mathsf{Z1} \cdot \tilde{z}_P$    // 1M
$\mathsf{mZ3} \leftarrow \mathsf{Z2} \cdot \tilde{y}_P + \mathsf{T1}$    // 1M + 1a
$\mathsf{D} \leftarrow \mathsf{Z2}^2 \cdot \tilde{x}_P + \mathsf{mZ3} \cdot \mathsf{Z1}$    // 2M + 1S + 1a
$\mathsf{T2} \leftarrow \mathsf{Z1} \cdot \mathsf{Z2}$    // 1M
$\mathsf{T3} \leftarrow \tilde{x}_P \cdot \tilde{x}_Q$    // 1M
$\mathsf{E} \leftarrow \mathsf{T3} \cdot (\mathsf{T3} \cdot (f_2 \cdot \mathsf{Z2}^2 - f_1 \cdot \mathsf{T2}) + \tilde{t}_Q \cdot \mathsf{D})$    // 5M + 1S + 1a + 1s
$\mathsf{E} \leftarrow \mathsf{E} + \mathsf{mZ3} \cdot \tilde{x}_Q^2 \cdot (f_3 \cdot \mathsf{Z2} \cdot \tilde{x}_P + f_4 \cdot \mathsf{mZ3})$    // 5M + 1S + 2a
$\mathsf{E} \leftarrow \mathsf{E} + \mathsf{mZ3} \cdot \tilde{x}_Q \cdot (\mathsf{mZ3} \cdot \tilde{y}_Q - \mathsf{Z2} \cdot \tilde{x}_P \cdot \tilde{z}_Q)$    // 5M + 1a + 1s
$\mathsf{X1} \leftarrow \tilde{x}_P \cdot (\mathsf{Z2} \cdot v_1(P) - \mathsf{Z1} \cdot v_0(P))$    // 3M + 1s
$\mathsf{T4} \leftarrow \mathsf{Z1} \cdot \tilde{y}_P + \mathsf{Z2} \cdot \tilde{x}_P$    // 2M + 1a
$\mathsf{X2} \leftarrow \mathsf{T1} \cdot v_1(P) + \mathsf{T4} \cdot v_0(P)$    // 2M + 1a
$\mathsf{C5} \leftarrow \mathsf{Z1}^2 - \mathsf{T4} \cdot \tilde{x}_Q$    // 1M + 1S + 1s
$\mathsf{C6} \leftarrow \mathsf{T1} \cdot \tilde{x}_Q + \mathsf{T2}$    // 1M + 1a
$\mathsf{T5} \leftarrow \tilde{z}_\oplus \cdot \tilde{x}_\ominus - \tilde{x}_\oplus \cdot \tilde{z}_\ominus$    // 2M + 1s
$\mathsf{X3} \leftarrow \mathsf{X1} \cdot \mathsf{T5} - \mathsf{X2} \cdot (\tilde{x}_\oplus \cdot \tilde{y}_\ominus - \tilde{y}_\oplus \cdot \tilde{x}_\ominus)$    // 4M + 2s
$(\mathsf{X5}, \mathsf{X6}) \leftarrow (\mathsf{X3} \cdot \mathsf{C5}, \mathsf{X3} \cdot \mathsf{C6})$    // 2M
$\mathsf{X4} \leftarrow \mathsf{T3} \cdot (\mathsf{X1} \cdot (\tilde{z}_\oplus \cdot \tilde{y}_\ominus - \tilde{y}_\oplus \cdot \tilde{z}_\ominus) + \mathsf{T5} \cdot \mathsf{X2})$    // 5M + 1a + 1s
$(\mathsf{X7}, \mathsf{X8}) \leftarrow (\mathsf{X5} + \mathsf{Z1} \cdot \mathsf{X4}, \mathsf{X6} + \mathsf{Z2} \cdot \mathsf{Z4})$    // 2M + 2a
$\mathsf{T6} \leftarrow \tilde{x}_\oplus \cdot \tilde{x}_\ominus$    // 1M
$\mathsf{E} \leftarrow -\mathsf{T6} \cdot \mathsf{T3} \cdot (\mathsf{E} \cdot \tilde{x}_P^2 + (\mathsf{X1} \cdot \mathsf{T3})^2)$    // 5M + 2S + 1a + 1neg
$(\mathsf{X9}, \mathsf{X10}) \leftarrow (\mathsf{E} \cdot \mathsf{X7}, \mathsf{E} \cdot \mathsf{X8})$    // 2M
$\mathsf{F} \leftarrow \mathsf{X2} \cdot (\tilde{x}_\oplus \cdot \tilde{y}_\ominus + \tilde{y}_\oplus \cdot \tilde{x}_\ominus) + \mathsf{X1} \cdot (\tilde{z}_\oplus \cdot \tilde{x}_\ominus + \tilde{x}_\oplus \cdot \tilde{z}_\ominus)$    // 6M + 3a
$\mathsf{F} \leftarrow \mathsf{X1} \cdot \mathsf{F} + 2(\mathsf{X2}^2 \cdot \mathsf{T6})$    // 2M + 1S + 2a
$\mathsf{F} \leftarrow -2(\mathsf{F} \cdot \mathsf{D} \cdot \mathsf{T6} \cdot \mathsf{T3} \cdot \mathsf{T3}^2 \cdot \tilde{x}_P)$    // 5M + 1S + 1a + 1neg
$(\mathsf{U1}, \mathsf{U0}) \leftarrow (-\mathsf{F} \cdot \tilde{y}_Q, \mathsf{F} \cdot \tilde{z}_Q)$    // 2M + 1neg
$\mathsf{Fi} \leftarrow 1/(\mathsf{F} \cdot \tilde{x}_Q)$    // 1M + 1I
$(u_1', u_0', v_1', v_0') \leftarrow (\mathsf{Fi} \cdot \mathsf{U1}, \mathsf{Fi} \cdot \mathsf{U0}, \mathsf{Fi} \cdot \mathsf{X9}, \mathsf{Fi} \cdot \mathsf{X10})$    // 4M
return $\langle X^2 + u_1' X + u_0', v_1' X + v_0' \rangle$
```

Algorithm 13: `jacobian_scalarmult`: Scalar multiplication on $\mathcal{J}_{\mathcal{C}}$, using the Montgomery ladder on $\mathcal{K}_{\mathcal{C}}$ and recovery to $\mathcal{J}_{\mathcal{C}}$. Assumes wrapped projected point as auxiliary input. Symbolic cost: $(7\beta+143)\mathbf{M} + (12\beta+12)\mathbf{S} + (12\beta+4)\mathbf{m_c} + (70+16\beta)\mathbf{a} + (22+16\beta)\mathbf{s} + 3\mathbf{neg} + \mathbf{I}$. ATmega: 9 968 127 cycles. Cortex: 2 709 401 cycles.

```
Input: $(m, P, (x_P/y_P, x_P/z_P, x_P/t_P)) \in [0, 2^\beta) \times \mathcal{J}_{\mathcal{C}}$
Output: $[m]P \in \mathcal{J}_{\mathcal{C}}$
$(\mathsf{X}_0, \mathsf{X}_1) \leftarrow$ crypto_scalarmult$(m, (x_P/y_P, x_P/z_P, x_P/t_P))$
                                        // $(7\beta+4)$M+12$\beta$S+12$\beta m_c$+16$\beta$a+16$\beta$s
$\mathsf{xP} \leftarrow$ xUNWRAP$((x_P/y_P, x_P/z_P, x_P/t_P))$    // 4M
return Recover$(P, \mathsf{xP}, \mathsf{X}_0, \mathsf{X}_1)$    // 139M+12S+4$m_c$+70a+22s+3neg+1I
```

do exactly what was specified in §2.1, calling the lower-level functions described in §3 and §4 as required. Our Diffie-Hellman key generation and key exchange use only the function `dh_exchange`, which implements exactly what we specified in §2.2: one call to `crypto_scalarmult` plus a call to `xWRAP` to convert to the correct 384-bit representation. Table 1 (in the introduction) presents the cycle counts and stack usage for all of our high-level functions.

Code and compilation. For our experiments, we compiled our AVR ATmega code with `avr-gcc -O2`, and our ARM Cortex M0 code with `clang -O2` (the optimization levels `-O3`, `-O1`, and `-Os` gave fairly similar results). The total program size is 20 242 bytes for the AVR ATmega, and 19 606 bytes for the ARM Cortex M0. This consists of the full signature and key-exchange code, including the reference implementation of the hash function `SHAKE128` with 512-bit output.[5]

Basis for comparison. As we believe ours to be the first genus-2 hyperelliptic curve implementation on both the AVR ATmega and the ARM Cortex M0 architectures, we can only compare with elliptic curve-based alternatives at the same 128-bit security level: notably [29], [30], [31], and [7]. This comparison is not superficial: the key exchange in [29], [30], and [7] uses the highly efficient x-only arithmetic on Montgomery elliptic curves, while [31] uses similar techniques for Weierstrass elliptic curves, and x-only arithmetic is the exact elliptic-curve analogue of Kummer surface arithmetic. To provide full scalar multiplication in a group, [31] appends y-coordinate recovery to its x-only arithmetic (using the approach of [26]); again, this is the elliptic-curve analogue of our methods.

Results for ARM Cortex M0. As we see in Table 4, genus-2 techniques give great results for Diffie–Hellman key exchange on the ARM Cortex M0 architecture. Compared with the current fastest implementation [7], we reduce the number of clock cycles by about 27%, while roughly halving code size and stack space. For signatures, the state-of-the-art is [31]: here we reduce the cycle count for the underlying scalar multiplications by a very impressive 75%, at the cost of an increase in code size and stack usage.

Results for AVR ATmega. Looking at Table 5, on the AVR ATmega architecture we reduce the cycle count for Diffie–Hellman by about 32% compared with the current record [7], again roughly halving the code size, and reducing stack usage by about 80%. The cycle count for Jacobian scalar multiplication (for signatures) is reduced by 71% compared with [31], while increasing the stack usage by 25%.

Finally we can compare to the current fastest full signature implementation [10], shown in Table 6. We almost halve the number of cycles, while reducing stack usage by a decent margin (code size is not reported in [10]).

[5] We used the reference C implementation for the Cortex M0, and the assembly implementation for AVR; both are available from [28]. The only change required is to the padding, which must take domain separation into account according to [12, p.28].

	Implementation	**Object**	**Clock cycles**	**Code size**	**Stack**
S,**DH**	Wenger et al. [31]	NIST P-256	≈ 10 730 000	7 168 bytes	540 bytes
DH	Düll et al. [7]	Curve25519	3 589 850	7 900 bytes	548 bytes
DH	**This work**	$\mathcal{K}_\mathcal{C}$	2 633 662	≈ 4 328 bytes	248 bytes
S	**This work**	$\mathcal{J}_\mathcal{C}$	2 709 401	≈ 9 874 bytes	968 bytes

Table 4. Comparison of scalar multiplication routines on the ARM Cortex M0 architecture at the 128-bit security level. **S** denotes signature-compatible full scalar multiplication; **DH** denotes Diffie–Hellman pseudo-scalar multiplication.

	Implementation	**Object**	**Cycles**	**Code size**	**Stack**
DH	Liu et al. [29]	256-bit curve	≈ 21 078 200	14 700 bytes*	556 bytes
S,**DH**	Wenger et al. [31]	NIST P-256	≈ 34 930 000	16 112 bytes	590 bytes
DH	Hutter, Schwabe [30]	Curve25519	22 791 579	n/a†	677 bytes
DH	Düll et al. [7]	Curve25519	13 900 397	17 710 bytes	494 bytes
DH	**This work**	$\mathcal{K}_\mathcal{C}$	9 513 536	≈ 9 490 bytes	99 bytes
S	**This work**	$\mathcal{J}_\mathcal{C}$	9 968 127	≈ 16 516 bytes	735 bytes

Table 5. Comparison of scalar multiplication routines on the AVR ATmega architecture at the 128-bit security level. **S** denotes signature-compatible full scalar multiplication; **DH** denotes Diffie–Hellman pseudo-scalar multiplication. The implementation marked * also contains a fixed-basepoint scalar multiplication routine, whereas the implementation marked † does not report code size for the separated scalar multiplication.

Implementation	**Object**	**Function**	**Cycles**	**Stack**
Nascimento et al. [10]	Ed25519	sig. gen.	19 047 706	1 473 bytes
Nascimento et al. [10]	Ed25519	sig. ver.	30 776 942	1 226 bytes
This work	$\mathcal{J}_\mathcal{C}$	`sign`	10 404 033	926 bytes
This work	$\mathcal{J}_\mathcal{C}$	`verify`	16 240 510	992 bytes

Table 6. Comparison of signature schemes on the AVR ATmega architecture at the 128-bit security level.

References

1. Bernstein, D.J.: Elliptic vs. hyperelliptic, part 1 (2006) `http://cr.yp.to/talks/2006.09.20/slides.pdf`. 1
2. Bos, J.W., Costello, C., Hisil, H., Lauter, K.E.: Fast cryptography in genus 2. In Johansson, T., Nguyen, P.Q., eds.: Advances in Cryptology – EUROCRYPT 2013. Volume 7881 of LNCS., Springer (2013) 194–210 `https://eprint.iacr.org/2012/670.pdf`. 1
3. Bernstein, D.J., Chuengsatiansup, C., Lange, T., Schwabe, P.: Kummer strikes back: New DH speed records. In Sarkar, P., Iwata, T., eds.: Advances in Cryptology – ASIACRYPT 2014. Volume 8873 of LNCS., Springer (2014) 317–337 `https://cryptojedi.org/papers/#kummer`. 1, 2, 5, 13

4. Costello, C., Longa, P.: FourQ: Four-dimensional decompositions on a Q-curve over the mersenne prime. In Iwata, T., Cheon, J.H., eds.: Advances in Cryptology – ASIACRYPT 2015. Volume 9452 of LNCS., Springer (2015) 214–235 https://eprint.iacr.org/2015/565. 2
5. Batina, L., Hwang, D., Hodjat, A., Preneel, B., Verbauwhede, I.: Hardware/software co-design for hyperelliptic curve cryptography (HECC) on the 8051μP. In Rao, J.R., Sunar, B., eds.: Cryptographic Hardware and Embedded Systems – CHES 2005. Volume 3659 of LNCS., Springer (2005) 106–118 https://www.iacr.org/archive/ches2005/008.pdf. 2
6. Hodjat, A., Batina, L., Hwang, D., Verbauwhede, I.: HW/SW co-design of a hyperelliptic curve cryptosystem using a microcode instruction set coprocessor. Integration, the VLSI Journal **40** (2007) 45–51 https://www.cosic.esat.kuleuven.be/publications/article-622.pdf. 2
7. Düll, M., Haase, B., Hinterwälder, G., Hutter, M., Paar, C., Sánchez, A.H., Schwabe, P.: High-speed curve25519 on 8-bit, 16-bit and 32-bit microcontrollers. Design, Codes and Cryptography **77** (2015) http://cryptojedi.org/papers/#mu25519. 2, 6, 17, 18
8. Costello, C., Chung, P.N., Smith, B.: Fast, uniform, and compact scalar multiplication for elliptic curves and genus 2 jacobians with applications to signature schemes. Cryptology ePrint Archive, Report 2015/983 (2015) https://eprint.iacr.org/2015/983. 2, 3, 4, 5, 8, 10, 12, 14
9. Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.Y.: High-speed high-security signatures. J. Cryptographic Engineering **2** (2012) 77–89 https://cryptojedi.org/papers/#ed25519. 2, 3
10. Nascimento, E., López, J., Dahab, R.: Efficient and secure elliptic curve cryptography for 8-bit avr microcontrollers. In Chakraborty, R.S., Schwabe, P., Solworth, J., eds.: Security, Privacy, and Applied Cryptography Engineering. Volume 9354 of LNCS., Springer (2015) 289–309 2, 17, 18
11. Schnorr, C.P.: Efficient identification and signatures for smart cards. In Brassard, G., ed.: Advances in Cryptology - CRYPTO '89. Volume 435 of LNCS., Springer (1989) 239–252 3, 4
12. Dworkin, M.J.: SHA-3 standard: Permutation-based hash and extendable-output functions. Technical report, National Institute of Standards and Technology (NIST) (2015) http://www.nist.gov/manuscript-publication-search.cfm?pub_id=919061. 3, 17
13. Katz, J., Wang, N.: Efficiency improvements for signature schemes with tight security reductions. In: Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS 2003, ACM (2003) 155–164 https://www.cs.umd.edu/~jkatz/papers/CCCS03_sigs.pdf. 4
14. M'Raïhi, D., Naccache, D., Pointcheval, D., Vaudenay, S.: Computational alternatives to random number generators. In Tavares, S.E., Meijer, H., eds.: Selected Areas in Cryptography. Volume 1556 of LNCS., Springer (1998) 72–80 https://www.di.ens.fr/~pointche/Documents/Papers/1998_sac.pdf. 4
15. Bernstein, D.J.: Differential addition chains. http://cr.yp.to/ecdh/diffchain-20060219.pdf (2006) 4
16. Stam, M.: Speeding up subgroup cryptosystems. PhD thesis, Technische Universiteit Eindhoven (2003) http://alexandria.tue.nl/extra2/200311829.pdf?q=subgroup. 5
17. Hutter, M., Schwabe, P.: Multiprecision multiplication on AVR revisited. Journal of Cryptographic Engineering **5** (2015) 201–214 http://cryptojedi.org/papers/#avrmul. 6

18. Gaudry, P., Schost, E.: Genus 2 point counting over prime fields. J. Symb. Comput. **47** (2012) 368–400 https://cs.uwaterloo.ca/~eschost/publications/countg2.pdf. 7
19. Hisil, H., Costello, C.: Jacobian coordinates on genus 2 curves. In Sarkar, P., Iwata, T., eds.: Advances in Cryptology – ASIACRYPT 2014. Volume 8873 of LNCS., Springer (2014) 338–357 https://eprint.iacr.org/2014/385.pdf. 8
20. Stahlke, C.: Point compression on jacobians of hyperelliptic curves over $\mathbb{F}_q$. Cryptology ePrint Archive, Report 2004/030 (2004) https://eprint.iacr.org/2004/030. 8
21. Chudnovsky, D.V., Chudnovsky, G.V.: Sequences of numbers generated by addition in formal groups and new primality and factorization tests. Adv. in Appl. Math. **7** (1986) 385–434 10
22. Cosset, R.: Applications of theta functions for hyperelliptic curve cryptography. PhD thesis, Université Henri Poincaré - Nancy I (2011) https://tel.archives-ouvertes.fr/tel-00642951/file/main.pdf. 10
23. Gaudry, P.: Fast genus 2 arithmetic based on Theta functions. J. Mathematical Cryptology **1** (2007) 243–265 https://eprint.iacr.org/2005/314/. 10
24. López, J., Dahab, R.: Fast multiplication on elliptic curves over GF(2^m) without precomputation. In Koç, Ç.K., Paar, C., eds.: Cryptographic Hardware and Embedded Systems. Volume 1717 of LNCS., Springer (1999) 316–327 14
25. Okeya, K., Sakurai, K.: Efficient elliptic curve cryptosystems from a scalar multiplication algorithm with recovery of the y-coordinate on a Montgomery-form elliptic curve. In Koç, Ç.K., Naccache, D., Paar, C., eds.: Cryptographic Hardware and Embedded Systems – CHES 2001. Volume 2162 of LNCS., Springer (2001) 126–141 14
26. Brier, É., Joye, M.: Weierstraß elliptic curves and side-channel attacks. In Naccache, D., Paillier, P., eds.: Public Key Cryptography. Volume 2274 of LNCS., Springer (2002) 335–345 http://link.springer.com/content/pdf/10.1007%2F3-540-45664-3_24.pdf. 14, 17
27. Cassels, J.W.S., Flynn, E.V.: Prolegomena to a middlebrow arithmetic of curves of genus 2. Volume 230. Cambridge University Press (1996) 14
28. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: The KECCAK sponge function family (2016) http://keccak.noekeon.org/. 17
29. Zhe Liu, Erich Wenger, J.G.: MoTE-ECC: Energy-scalable elliptic curve cryptography for wireless sensor networks. In Boureanui, I., Owesarski, P., Vaudenay, S., eds.: Applied Cryptography and Network Security. Volume 8479 of LNCS., Springer (2014) 361–379 https://online.tugraz.at/tug_online/voe_main2.getvolltext?pCurrPk=77985. 17, 18
30. Hutter, M., Schwabe, P.: NaCl on 8-bit AVR microcontrollers. In Youssef, A., Nitaj, A., eds.: Progress in Cryptology – AFRICACRYPT 2013. Volume 7918 of LNCS., Springer (2013) 156–172 http://cryptojedi.org/papers/#avrnacl. 17, 18
31. Wenger, E., Unterluggauer, T., Werner, M.: 8/16/32 shades of elliptic curve cryptography on embedded processors. In Paul, G., Vaudenay, S., eds.: Progress in Cryptology – INDOCRYPT 2013. Volume 8250 of LNCS., Springer (2013) 244–261 https://online.tugraz.at/tug_online/voe_main2.getvolltext?pCurrPk=72486. 17, 18