qDSA: Small and Secure Digital Signatures with Curve-Based Diffie–Hellman Key Pairs

Joost Renes[1](✉) and Benjamin Smith[2]

[1] Digital Security Group, Radboud University, Nijmegen, The Netherlands
j.renes@cs.ru.nl
[2] INRIA and Laboratoire d'Informatique de l'École polytechnique (LIX),
Université Paris–Saclay, Palaiseau, France
smith@lix.polytechnique.fr

Abstract. qDSA is a high-speed, high-security signature scheme that facilitates implementations with a very small memory footprint, a crucial requirement for embedded systems and IoT devices, and that uses the same public keys as modern Diffie–Hellman schemes based on Montgomery curves (such as Curve25519) or Kummer surfaces. qDSA resembles an adaptation of EdDSA to the world of Kummer varieties, which are quotients of algebraic groups by ±1. Interestingly, qDSA does not require any full group operations or point recovery: all computations, including signature verification, occur on the quotient where there is no group law. We include details on four implementations of qDSA, using Montgomery and fast Kummer surface arithmetic on the 8-bit AVR ATmega and 32-bit ARM Cortex M0 platforms. We find that qDSA significantly outperforms state-of-the-art signature implementations in terms of stack usage and code size. We also include an efficient compression algorithm for points on fast Kummer surfaces, reducing them to the same size as compressed elliptic curve points for the same security level.

Keywords: Signatures · Kummer · Curve25519 · Diffie–Hellman · Elliptic curve · Hyperelliptic curve

1 Introduction

Modern asymmetric cryptography based on elliptic and hyperelliptic curves [29, 31] achieves two important goals. The first is efficient key exchange using the Diffie–Hellman protocol [16], using the fact that the (Jacobian of the) curve carries the structure of an abelian group. But in fact, as Miller observed [31], we do not need the full group structure for Diffie–Hellman: the associated *Kummer variety* (the quotient by ±1) suffices, which permits more efficiently-computable arithmetic [21,32]. Perhaps the most well-known example is Curve25519 [5], which offers fast scalar multiplications based on x-only arithmetic.

J. Renes—This work has been supported by the Technology Foundation STW (project 13499 - TYPHOON & ASPASIA), from the Dutch government.

T. Takagi and T. Peyrin (Eds.): ASIACRYPT 2017, Part II, LNCS 10625, pp. 273–302, 2017.
https://doi.org/10.1007/978-3-319-70697-9_10

The second objective is efficient digital signatures, which are critical for authentication. There are several group-based signature schemes, the most important of which are ECDSA [1], Schnorr [40], and now EdDSA [8] signatures. In contrast to the Diffie–Hellman protocol, all of these signature schemes explicitly require the group structure of the (Jacobian of the) curve. An unfortunate side-effect of this is that users essentially need two public keys to support both curve-based protocols. Further, basic cryptographic libraries need to provide implementations for arithmetic on both the Jacobian and the Kummer variety, thus complicating and increasing the size of the trusted code base. For example, the NaCl library [9] uses Ed25519 [8] for signatures, and Curve25519 [5] for key exchange. This problem is worse for genus-2 hyperelliptic curves, where the Jacobian is significantly harder to use safely than its Kummer surface.

There have been several partial solutions to this problem. By observing that elements of the Kummer variety are elements of the Jacobian *up to sign*, one can build scalar multiplication on the Jacobian based on the fast Kummer arithmetic [14,35]. This avoids the need for a separate scalar multiplication on the Jacobian, but does not avoid the need for its group law; it also introduces the need for projecting to and recovering from the Kummer. In any case, it does not solve the problem of having different public key types.

Another proposal is XEdDSA [36], which uses the public key on the Kummer variety to construct EdDSA signatures. In essence, it creates a key pair on the Jacobian by appending a sign bit to the public key on the Kummer variety, which can then be used for signatures. In [23] Hamburg shows that one can actually verify signatures using only the x-coordinates of points on an elliptic curve, which is applied in the recent STROBE framework [24]. We generalize this approach to allow Kummer varieties of curves of higher genera, and naturally adapt the scheme by only allowing challenges up to sign. This allows us to provide a proof of security, which has thus far not been attempted (in [23] Hamburg remarks that verifying up to sign does "probably not impact security at all"). Similar techniques have been applied for batch verification of ECDSA signatures [28], using the theory of summation polynomials [41].

In this paper we show that there is no intrinsic reason why Kummer varieties cannot be used for signatures. We present `qDSA`, a signature scheme relying only on Kummer arithmetic, and prove it secure in the random oracle model. It should not be surprising that the reduction in our proof is slightly weaker than the standard proof of security of Schnorr signatures [37], but not by more than we should expect. There is no difference between public keys for `qDSA` and Diffie–Hellman. After an abstract presentation in Sect. 2, we give a detailed description of elliptic-curve `qDSA` instances in Sect. 3. We then move on to genus-2 instances based on fast Kummer surfaces, which give better performance. The necessary arithmetic appears in Sect. 4, before Sect. 5 describes the new verification algorithm.

We also provide an efficient compression method for points on fast Kummer surfaces in Sect. 6, solving a long-standing open problem [6]. Our technique means that `qDSA` public keys for $g = 2$ can be efficiently compressed to 32 bytes, and that qDSA signatures fit into 64 bytes; it also finally reduces the size of Kummer-based Diffie–Hellman public keys from 48 to 32 bytes.

Finally, we provide constant-time software implementations of genus-1 and genus-2 qDSA instances for the AVR ATmega and ARM Cortex M0 platforms. The performance of all four qDSA implementations, reported in Sect. 7, comfortably beats earlier implementations in terms of stack usage and code size.

Source code. We place all of the software described here into the public domain, to maximize the reusability of our results. The software is available at http://www.cs.ru.nl/jrenes/.

2 The qDSA Signature Scheme

In this section we define qDSA, the *quotient Digital Signature Algorithm*. We start by recalling the basics of Kummer varieties in Sect. 2.1 and defining key operations in Sect. 2.2. The rest of the section is dedicated to the definition of the qDSA signature scheme, which is presented in full in Algorithm 1, and its proof of security, which follows Pointcheval and Stern [37,38]. qDSA closely resembles the Schnorr signature scheme [40], as it results from applying the Fiat–Shamir heuristic [19] to an altered Schnorr identification protocol, together with a few standard changes as in EdDSA [8]. We comment on some special properties of qDSA in Sect. 2.5.

Throughout, we work over finite fields $\mathbb{F}_p$ with $p > 3$. Our low-level algorithms include costs in terms of basic $\mathbb{F}_p$-operations: $\mathbf{M}$, $\mathbf{S}$, $\mathbf{C}$, $\mathbf{a}$, $\mathbf{s}$, $\mathbf{I}$, and $\mathbf{E}$ denote the unit costs of computing a single multiplication, squaring, multiplication by a small constant, addition, subtraction, inverse, and square root, respectively.

2.1 The Kummer Variety Setting

Let $\mathcal{C}$ be a (hyper)elliptic curve and $\mathcal{J}$ its Jacobian[1]. The Jacobian is a commutative algebraic group with group operation $+$, inverse $-$, and identity 0. We assume $\mathcal{J}$ has a subgroup of large prime order N. The associated *Kummer variety* $\mathcal{K}$ is the quotient $\mathcal{K} = \mathcal{J}/\pm$. By definition, working with $\mathcal{K}$ corresponds to working on $\mathcal{J}$ *up to sign*. If P is an element of $\mathcal{J}$, we denote its image in $\mathcal{K}$ by $\pm P$. In this paper we take $\log_2 N \approx 256$, and consider two important cases.

Genus 1. Here $\mathcal{J} = \mathcal{C}/\mathbb{F}_p$ is an elliptic curve with $\log_2 p \approx 256$, while $\mathcal{K} = \mathbb{P}^1$ is the x-line. We choose $\mathcal{C}$ to be Curve25519 [5], which is the topic of Sect. 3.

Genus 2. Here $\mathcal{J}$ is the Jacobian of a genus-2 curve $\mathcal{C}/\mathbb{F}_p$, where $\log_2 p \approx 128$, and $\mathcal{K}$ is a *Kummer surface*. We use the Gaudry–Schost parameters [22] for our implementations. Kummer arithmetic, including some new constructions we need for signature verification and compression, is described in Sects. 4, 5 and 6.

[1] In what follows, we could replace $\mathcal{J}$ by an arbitrary abelian group and all the proofs would be completely analogous. For simplicity we restrict to the cryptographically most interesting case of a Jacobian.

A point $\pm P$ in $\mathcal{K}(\mathbb{F}_p)$ is the image of a pair of points $\{P, -P\}$ on $\mathcal{J}$. It is important to note that P and $-P$ are not necessarily in $\mathcal{J}(\mathbb{F}_p)$; if not, then they are conjugate points in $\mathcal{J}(\mathbb{F}_{p^2})$, and correspond to points in $\mathcal{J}'(\mathbb{F}_p)$, where $\mathcal{J}'$ is the *quadratic twist* of $\mathcal{J}$. Both $\mathcal{J}$ and $\mathcal{J}'$ always have the same Kummer variety; we return to this fact, and its implications for our scheme, in Sect. 2.5 below.

2.2 Basic Operations

While a Kummer variety $\mathcal{K}$ has no group law, the operation

$$\{\pm P, \pm Q\} \mapsto \{\pm(P+Q), \pm(P-Q)\} \tag{1}$$

is well-defined. We can therefore define a *pseudo-addition* operation by $\texttt{xADD} : (\pm P, \pm Q, \pm(P-Q)) \mapsto \pm(P+Q)$. The special case where $\pm(P-Q) = \pm 0$ is the *pseudo-doubling* $\texttt{xDBL} : \pm P \mapsto \pm[2]P$. In our applications we can often improve efficiency by combining two of these operations in a single function

$$\texttt{xDBLADD} : (\pm P, \pm Q, \pm(P-Q)) \longmapsto (\pm[2]P, \pm(P+Q)) \,.$$

For any integer m, the scalar multiplication $[m]$ on $\mathcal{J}$ induces the key cryptographic operation of *pseudomultiplication* on $\mathcal{K}$, defined by

$$\texttt{Ladder} : (m, \pm P) \longmapsto \pm[m]P \,.$$

As its name suggests, we compute `Ladder` using Montgomery's famous ladder algorithm [32], which is a uniform sequence of `xDBLADD`s and constant-time conditional swaps.[2] This constant-time nature will be important for signing.

Our signature verification requires a function `Check` on $\mathcal{K}^3$ defined by

$$\texttt{Check} : (\pm P, \pm Q, \pm R) \longmapsto \begin{cases} \textbf{True} & \text{if } \pm R \in \{\pm(P+Q), \pm(P-Q)\} \\ \textbf{False} & \text{otherwise} \end{cases}$$

Since we are working with projective points, we need a way to uniquely represent them. Moreover, we want this representation to be as small as possible, to minimize communication overhead. For this purpose we define the functions

$$\texttt{Compress} : \mathcal{K}(\mathbb{F}_p) \longrightarrow \{0,1\}^{256} \,,$$

writing $\overline{\pm P} := \texttt{Compress}(\pm P)$, and

$$\texttt{Decompress} : \{0,1\}^{256} \longrightarrow \mathcal{K}(\mathbb{F}_p) \cup \{\perp\}$$

such that $\texttt{Decompress}(\overline{\pm P}) = \pm P$ for $\pm P$ in $\mathcal{K}(\mathbb{F}_p)$ and $\texttt{Decompress}(X) = \perp$ for $X \in \{0,1\}^{256} \setminus \mathrm{Im}(\texttt{Compress})$.

For the remainder of this section we assume that `Ladder`, `Check`, `Compress`, and `Decompress` are defined. Their implementation depends on whether we are in the genus 1 or 2 setting; we return to this in later sections.

[2] In contemporary implementations such as NaCl, the `Ladder` function is sometimes named `crypto_scalarmult`.

2.3 The qID Identification Protocol

Let P be a generator of a prime order subgroup of $\mathcal{J}$, of order N, and let $\pm P$ be its image in $\mathcal{K}$. Let $\mathbb{Z}_N^+$ denote the subset of $\mathbb{Z}_N$ with zero least significant bit (where we identify elements of $\mathbb{Z}_N$ with their representatives in $[0, N-1]$). Note that since N is odd, $\mathtt{LSB}(-x) = 1 - \mathtt{LSB}(x)$ for all $x \in \mathbb{Z}_N^*$. The private key is an element $d \in \mathbb{Z}_N$. Let $Q = [d]P$ and let the public key be $\pm Q$. Now consider the following Schnorr-style identification protocol, which we call qID:

(1) The prover sets $r \leftarrow_R \mathbb{Z}_N^*$, $\pm R \leftarrow \pm[r]P$ and sends $\pm R$ to the verifier;
(2) The verifier sets $c \leftarrow_R \mathbb{Z}_N^+$ and sends c to the prover;
(3) The prover sets $s \leftarrow (r - cd) \mod N$ and sends s to the verifier;
(4) The verifier accepts if and only if $\pm R \in \{\pm([s]P + [c]Q), \pm([s]P - [c]Q)\}$.

There are some important differences between qID and the basic Schnorr identification protocol in [40].

Scalar multiplications on $\mathcal{K}$. It is well-known that one can use $\mathcal{K}$ to perform the scalar multiplication [14,35] within a Schnorr identification or signature scheme, but with this approach one must always lift back to an element of a group. In contrast, in our scheme this recovery step is not necessary.

Verification on $\mathcal{K}$. The original verification [40] requires checking that $R = [s]P + [c]Q$ for some $R, [s]P, [c]Q \in \mathcal{J}$. Working on $\mathcal{K}$, we only have these values up to sign (i. e. $\pm R$, $\pm[s]P$ and $\pm[c]Q$), which is not enough to check that $R = [s]P + [c]Q$. Instead, we only verify that $\pm R = \pm([s]P \pm [c]Q)$.

Challenge from $\mathbb{Z}_N^+$. A Schnorr protocol using the weaker verification above would not satisfy the special soundness property: the transcripts $(\pm R, c, s)$ and $(\pm R, -c, s)$ are both valid, and do not allow us to extract a witness. Choosing c from $\mathbb{Z}_N^+$ instead of $\mathbb{Z}$ eliminates this possibility, and allows a security proof. (This is the main difference with Hamburg's STROBE [24].)

Proposition 1. *The qID identification protocol is a sigma protocol.*

Proof. We prove the required properties (see [25, Sect. 6]).

Completeness: If the protocol is followed, then $r = s + cd$, and therefore $[r]P = [s]P + [c]Q$ on $\mathcal{J}$. Mapping to $\mathcal{K}$, it follows that $\pm R = \pm([s]P + [c]Q)$.

Special soundness: Let $(\pm R, c_0, s_0)$ and $(\pm R, c_1, s_1)$ be two valid transcripts such that $c_0 \neq c_1$. By verification, each $s_i \equiv \pm r \pm c_i d \pmod N$, so $s_0 \pm s_1 \equiv (c_0 \pm c_1)\, d \pmod N$, where the signs are chosen to cancel r. Now $c_0 \pm c_1 \not\equiv 0 \pmod N$ because c_0 and c_1 are both in $\mathbb{Z}_N^+$, so we can extract a witness $d \equiv (s_0 \pm s_1)\,(c_0 \pm c_1)^{-1} \pmod N$.

Honest-verifier zero-knowledge: A simulator $\mathcal{S}$ generates $c \leftarrow_R \mathbb{Z}_N^+$ and sets $s \leftarrow_R \mathbb{Z}_N$ and $R \leftarrow [s]P + [c]Q$.[3] If $R = \mathcal{O}$, it restarts. It outputs $(\pm R, c, s)$. As in [38, Lemma 5], we let

$$\delta = \left\{(\pm R, c, s) : c \in_R \mathbb{Z}_N^+\,, r \in_R \mathbb{Z}_N^*\,, \pm R = \pm[r]P\,, s = r - cd\right\}\,,$$
$$\delta' = \left\{(\pm R, c, s) : c \in_R \mathbb{Z}_N^+\,, s \in_R \mathbb{Z}_N\,, R = [s]P + [c]Q\,, R \neq \mathcal{O}\right\}$$

[3] As we only know Q up to sign, we may need two attempts to construct $\mathcal{S}$.

be the distributions of honest and simulated signatures, respectively. The elements of δ and δ' are the same. First, consider δ. There are exactly $N-1$ choices for r, and exactly $(N+1)/2$ for c; all of them lead to distinct tuples. There are thus $(N^2-1)/2$ possible tuples, all of which have probability $2/(N^2-1)$ of occurring. Now consider δ'. Again, there are $(N+1)/2$ choices for c. We have N choices for s, exactly one of which leads to $R = \mathcal{O}$. Thus, given c, there are $N-1$ choices for s. We conclude that δ' also contains $(N^2-1)/2$ possible tuples, which all have probability $2/(N^2-1)$ of occurring. □

2.4 Applying Fiat–Shamir

Applying the Fiat–Shamir transform [19] to `qID` yields a signature scheme `qSIG`. We will need a hash function $\overline{H} : \{0,1\}^* \to \mathbb{Z}_N^+$, which we define by taking a hash function $H : \{0,1\}^* \to \mathbb{Z}_N$ and then setting $\overline{H}$ by

$$\overline{H}(M) \longmapsto \begin{cases} H(M) & \text{if } \mathtt{LSB}(H(M)) = 0 \\ -H(M) & \text{if } \mathtt{LSB}(H(M)) = 1 \end{cases}.$$

The `qSIG` signature scheme is defined as follows:

(1) To sign a message $M \in \{0,1\}^*$ with private key $d \in \mathbb{Z}_N$ and public key $\pm Q \in \mathcal{K}$, the `prover` sets $r \leftarrow_R \mathbb{Z}_N^*$, $\pm R \leftarrow \pm[r]R$, $h \leftarrow \overline{H}(\pm R \mid\mid M)$, and $s \leftarrow (r - hd) \bmod N$, and sends $(\pm R \mid\mid s)$ to the `verifier`.

(2) To verify a signature $(\pm R \mid\mid s) \in \mathcal{K} \times \mathbb{Z}_N$ on a message $M \in \{0,1\}^*$ with public key $\pm Q \in \mathcal{K}$, the `verifier` sets $h \leftarrow \overline{H}(\pm R \mid\mid M)$, $\pm\mathcal{T}_0 \leftarrow \pm[s]P$, and $\pm\mathcal{T}_1 \leftarrow \pm[h]Q$, and accepts if and only if $\pm R \in \{\pm(\mathcal{T}_0 + \mathcal{T}_1), \pm(\mathcal{T}_0 - \mathcal{T}_1)\}$.

Proposition 2 asserts that the security properties of `qID` carry over to `qSIG`.

Proposition 2. *In the random oracle model, if an existential forgery of the `qSIG` signature scheme under an adaptive chosen message attack has non-negligible probability of success, then the DLP in $\mathcal{J}$ can be solved in polynomial time.*

Proof. This is the standard proof of applying the Fiat–Shamir transform to a sigma protocol: see [37, Theorem 13] or [38, Sect. 3.2]. □

2.5 The `qDSA` Signature Scheme

Moving towards the real world, we slightly alter the `qSIG` protocol with some pragmatic choices, following Bernstein et al. [8]:

(1) We replace the randomness r by the output of a pseudo-random function, which makes the signatures deterministic.

(2) We include the public key $\pm Q$ in the generation of the challenge, to prevent attackers from attacking multiple public keys at the same time.

(3) We compress and decompress points on $\mathcal{K}$ where necessary.

The resulting signature scheme, `qDSA`, is summarized in Algorithm 1.

Algorithm 1. The qDSA signature scheme

```
function keypair
    Input: ()
    Output: $(\overline{\pm Q} \mid\mid (d' \mid\mid d''))$: a compressed public key $\overline{\pm Q} \in \{0,1\}^{256}$
            where $\pm Q \in \mathcal{K}$, and a private key $(d' \mid\mid d'') \in \left(\{0,1\}^{256}\right)^2$
    $d \leftarrow \texttt{Random}(\{0,1\}^{256})$
    $(d' \mid\mid d'') \leftarrow H(d)$
    $\pm Q \leftarrow \texttt{Ladder}(d', \pm P)$                      // $\pm Q = \pm[d']P$
    $\overline{\pm Q} \leftarrow \texttt{Compress}(\pm Q)$
    return $(\overline{\pm Q} \mid\mid (d' \mid\mid d''))$
function sign
    Input: $d', d'' \in \{0,1\}^{256}$, $\overline{\pm Q} \in \{0,1\}^{256}$, $M \in \{0,1\}^*$
    Output: $(\overline{\pm R} \mid\mid s) \in \left(\{0,1\}^{256}\right)^2$
    $r \leftarrow H(d'' \mid\mid M)$
    $\pm R \leftarrow \texttt{Ladder}(r, \pm P)$                       // $\pm R = \pm[r]P$
    $\overline{\pm R} \leftarrow \texttt{Compress}(\pm R)$
    $h \leftarrow \overline{H}(\overline{\pm R} \mid\mid \overline{\pm Q} \mid\mid M)$
    $s \leftarrow (r - hd') \bmod N$
    return $(\overline{\pm R} \mid\mid s)$
function verify
    Input: $M \in \{0,1\}^*$, the compressed public key $\overline{\pm Q} \in \{0,1\}^{256}$, and a
           putative signature $(\overline{\pm R} \mid\mid s) \in \left(\{0,1\}^{256}\right)^2$
    Output: True if $(\overline{\pm R} \mid\mid s)$ is a valid signature on $M$ under $\overline{\pm Q}$,
            False otherwise
    $\pm Q \leftarrow \texttt{Decompress}(\overline{\pm Q})$
    if $\pm Q = \perp$ then
        return False
    $h \leftarrow \overline{H}(\overline{\pm R} \mid\mid \overline{\pm Q} \mid\mid M)$
    $\pm \mathcal{T}_0 \leftarrow \texttt{Ladder}(s, \pm P)$                  // $\pm \mathcal{T}_0 = \pm[s]P$
    $\pm \mathcal{T}_1 \leftarrow \texttt{Ladder}(h, \pm Q)$                  // $\pm \mathcal{T}_1 = \pm[h]Q$
    $\pm R \leftarrow \texttt{Decompress}(\overline{\pm R})$
    if $\pm R = \perp$ then
        return False
    $v \leftarrow \texttt{Check}(\pm \mathcal{T}_0, \pm \mathcal{T}_1, \pm R)$         // is $\pm R = \pm(\mathcal{T}_0 \pm \mathcal{T}_1)$?
    return $v$
```

Unified keys. Signatures are entirely computed and verified on $\mathcal{K}$, which is also the natural setting for Diffie–Hellman key exchange. We can therefore use identical key pairs for Diffie–Hellman and for qDSA signatures. This significantly simplifies the implementation of cryptographic libraries, as we no longer need arithmetic for the two distinct objects $\mathcal{J}$ and $\mathcal{K}$. Technically, there is no reason not to use a single key pair for both key exchange and signing; but one should be very careful in doing so, as using one key across multiple protocols could potentially lead to attacks. The primary interest of this aspect of qDSA is not necessarily in reducing the number of keys, but in unifying key formats and reducing the size of the trusted code base.

Security level. The security reduction to the discrete logarithm problem is almost identical to the case of Schnorr signatures [37]. Notably, the challenge space has about half the size ($\mathbb{Z}_N^+$ versus $\mathbb{Z}_N$) while the proof of soundness computes either $s_0 + s_1$ or $s_0 - s_1$. This results in a slightly weaker reduction, as should be expected by moving from $\mathcal{J}$ to $\mathcal{K}$ and by weakening verification. By choosing $\log_2 N \approx 256$ we obtain a scheme with about the same security level as state-of-the-art schemes (eg. EdDSA combined with Ed25519). This could be made more precise (cf. [38]), but we do not provide this analysis here.

Key and signature sizes. Public keys fit into 32 bytes in both the genus 1 and genus 2 settings. This is standard for Montgomery curves; for Kummer surfaces it requires a new compression technique, which we present in Sect. 6. In both cases $\log_2 N < 256$, which means that signatures $(\pm R \mid\mid s)$ fit in 64 bytes.

Twist security. Rational points on $\mathcal{K}$ correspond to pairs of points on either $\mathcal{J}$ or its quadratic twist. As opposed to Diffie–Hellman, in qDSA scalar multiplications with secret scalars are *only* performed on the public parameter $\pm P$, which is chosen as the image of large prime order element of $\mathcal{J}$. Therefore $\mathcal{J}$ is not technically required to have a secure twist, unlike in the modern Diffie–Hellman setting. But if $\mathcal{K}$ is also used for key exchange (which is the whole point!), then twist security is crucial. We therefore strongly recommend twist-secure parameters for qDSA implementations.

Hash function. The function H can be any hash function with at least a $\log_2 \sqrt{N}$-bit security level and at least $2 \log_2 N$-bit output. Throughout this paper we take H to be the extendable output function SHAKE128 [18] with fixed 512-bit output. This enables us to implicitly use H as a function mapping into either $\mathbb{Z}_N \times \{0,1\}^{256}$ (eg. Line 3 of Algorithm 1), $\mathbb{Z}_N$ (eg. Line 8 of Algorithm 1), or $\mathbb{Z}_N^+$ (eg. Line 11 of Algorithm 1, by combining it with a conditional negation) by appropriately reducing (part of) the output modulo N.

Signature compression. Schnorr mentions in [40] that signatures $(R \mid\mid s)$ may be compressed to $(H(R \mid\mid Q \mid\mid M) \mid\mid s)$, taking only the first 128 bits of the hash, thus reducing signature size from 64 to 48 bytes. This is possible because we can recompute R from P, Q, s, and $H(R \mid\mid Q \mid\mid M)$. However, on $\mathcal{K}$ we cannot recover $\pm R$ from $\pm P$, $\pm Q$, s, and $H(\pm R \mid\mid \pm Q \mid\mid M)$, so Schnorr's compression technique is not an option for us.

Batching. Proposals for batch signature verification typically rely on the group structure, verifying random linear combinations of points [8,33]. Since $\mathcal{K}$ has no group structure, these batching algorithms are not possible.

Scalar multiplication for verification. Instead of computing the full point $[s]P+[c]Q$ with a two-dimensional multiscalar multiplication operation, we have to compute $\pm[s]P$ and $\pm[c]Q$ separately. As a result we are unable to use standard tricks for speeding up two-dimensional scalar multiplications (eg. [20]), resulting in increased run-time. On the other hand, it has the benefit of relying on the already implemented `Ladder` function, mitigating the need for a separate algorithm, and is more memory-friendly. Our implementations show a significant decrease in stack usage, at the cost of a small loss of speed (see Sect. 7).

3 Implementing `qDSA` with Elliptic Curves

Our first concrete instantiation of `qDSA` uses the Kummer variety of an elliptic curve, which is just the x-line $\mathbb{P}^1$.

3.1 Montgomery Curves

Consider the elliptic curve in Montgomery form

$$E_{AB}/\mathbb{F}_p : By^2 = x(x^2 + Ax + 1)\,,$$

where $A^2 \neq 4$ and $B \neq 0$. The map $E_{AB} \to \mathcal{K} = \mathbb{P}^1$ defined by

$$P = (X : Y : Z) \longmapsto \pm P = \begin{cases} (X : Z) & \text{if } Z \neq 0 \\ (1 : 0) & \text{if } Z = 0 \end{cases}$$

gives rise to efficient x-only arithmetic on $\mathbb{P}^1$ (see [32]). We use the `Ladder` specified in [17, Algorithm 1]. Compression uses Bernstein's map

$$\texttt{Compress} : (X : Z) \in \mathbb{P}^1(\mathbb{F}_p) \longmapsto XZ^{p-2} \in \mathbb{F}_p,$$

while decompression is the near-trivial

$$\texttt{Decompress} : x \in \mathbb{F}_p \longmapsto (x : 1) \in \mathbb{P}^1(\mathbb{F}_p).$$

Note that `Decompress` never returns $\perp$, and that $\texttt{Decompress}(\texttt{Compress}((X : Z))) = (X : Z)$ whenever $Z \neq 0$ (however, the points $(0 : 1)$ and $(1 : 0)$ should never appear as public keys or signatures).

3.2 Signature Verification

It remains to define the `Check` operation for Montgomery curves. In the final step of verification we are given $\pm R$, $\pm P$, and $\pm Q$ in $\mathbb{P}^1$, and we need to check whether $\pm R \in \{\pm(P+Q), \pm(P-Q)\}$. Proposition 3 reduces this to checking a quadratic relation in the coordinates of $\pm R$, $\pm P$, and $\pm Q$.

Proposition 3. *Writing $(X^P : Z^P) = \pm P$ for P in E_{AB}, etc.: If P, Q, and R are points on E_{AB}, then $\pm R \in \big\{\pm(P+Q), \pm(P-Q)\big\}$ if and only if*

$$B_{ZZ}(X^R)^2 - 2B_{XZ}X^R Z^R + B_{XX}(Z^R)^2 = 0 \tag{2}$$

where

$$B_{XX} = \big(X^P X^Q - Z^P Z^Q\big)^2, \tag{3}$$
$$B_{XZ} = \big(X^P X^Q + Z^P Z^Q\big)\big(X^P Z^Q + Z^P X^Q\big) + 2AX^P Z^P X^Q Z^Q, \tag{4}$$
$$B_{ZZ} = \big(X^P Z^Q - Z^P X^Q\big)^2. \tag{5}$$

Proof. Let $S = (X^S : Z^S) = \pm(P+Q)$ and $D = (X^D : Z^D) = \pm(P-Q)$. If we temporarily assume $\pm 0 \neq \pm P \neq \pm Q \neq \pm 0$ and put $x_P = X^P/Z^P$, etc., then the group law on E_{AB} gives us $x_S x_D = (x_P x_Q - 1)^2/(x_P - x_Q)^2$ and $x_S + x_D = 2((x_P x_Q + 1)(x_P + x_Q) + 2Ax_P x_Q)$. Homogenizing, we obtain

$$\big(X^S X^D : X^S Z^D + Z^S X^D : Z^S Z^D\big) = (\lambda B_{XX} : \lambda 2B_{XZ} : \lambda B_{ZZ})\,. \tag{6}$$

One readily verifies that Eq. (6) still holds even when the temporary assumption does not (that is, when $\pm P = \pm Q$ or $\pm P = \pm 0$ or $\pm Q = \pm 0$). Having degree 2, the homogeneous polynomial $B_{ZZ}X^2 - B_{XZ}XZ + B_{XX}Z^2$ cuts out two points in $\mathbb{P}^1$ (which may coincide); by Eq. (6), they are $\pm(P+Q)$ and $\pm(P-Q)$, so if $(X^R : Z^R)$ satisfies Eq. (2) then it must be one of them. □

3.3 Using Cryptographic Parameters

We use the elliptic curve $E/\mathbb{F}_p : y^2 = x^3 + 486662x^2 + x$ where $p = 2^{255} - 19$, which is commonly referred to as `Curve25519` [5]. Let $P \in E(\mathbb{F}_p)$ be such that $\pm P = (9 : 1)$. Then P has order $8N$, where

$$N = 2^{252} + 27742317777372353535851937790883648493$$

is prime. The `xDBLADD` operation requires us to store $(A+2)/4 = 121666$, and we implement optimized multiplication by this constant. In [5, Sect. 3] Bernstein sets and clears some bits of the private key, also referred to as "clamping". This is not necessary in `qDSA`, but we do it anyway in `keypair` for compatibility.

Algorithm 2. Checking the verification relation for $\mathbb{P}^1$

```
function Check
    Input: $\pm P$, $\pm Q$, $\pm R = (x : 1)$ in $\mathbb{P}^1$ images of points of $E_{AB}(\mathbb{F}_p)$
    Output: True if $\pm R \in \{\pm(P+Q), \pm(P-Q)\}$, False otherwise
    Cost: $8\mathbf{M} + 3\mathbf{S} + 1\mathbf{C} + 8\mathbf{a} + 4\mathbf{s}$
    $(B_{XX}, B_{XZ}, B_{ZZ}) \leftarrow$ BValues$(\pm P, \pm Q)$
    if $B_{XX}x^2 - B_{XZ}x + B_{ZZ} = 0$ then return True
    else return False

function BValues
    Input: $\pm P = (X^P : Z^P)$, $\pm Q = (X^Q : Z^Q)$ in $\mathcal{K}(\mathbb{F}_p)$
    Output: $(B_{XX}(\pm P, \pm Q), B_{XZ}(\pm P, \pm Q), B_{ZZ}(\pm P, \pm Q))$ in $\mathbb{F}_p^3$
    Cost: $6\mathbf{M} + 2\mathbf{S} + 1\mathbf{C} + 7\mathbf{a} + 3\mathbf{s}$
    // Use Eq. (3), (4) and (5) in Proposition 3
```

4 Implementing qDSA with Kummer Surfaces

A number of cryptographic protocols that have been successfully implemented with Montgomery curves have seen substantial practical improvements when the curves are replaced with *Kummer surfaces*. From a general point of view, a Kummer surface is the quotient of some genus-2 Jacobian $\mathcal{J}$ by ± 1; geometrically it is a surface in $\mathbb{P}^3$ with sixteen point singularities, called *nodes*, which are the images in $\mathcal{K}$ of the 2-torsion points of $\mathcal{J}$ (since these are precisely the points fixed by -1). From a cryptographic point of view, a Kummer surface is just a 2-dimensional analogue of the x-coordinate used in Montgomery curve arithmetic.

The algorithmic and software aspects of efficient Kummer surface arithmetic have already been covered in great detail elsewhere (see eg. [7,21,39]). Indeed, the Kummer scalar multiplication algorithms and software that we use in our signature implementation are identical to those described in [39], and use the cryptographic parameters proposed by Gaudry and Schost [22].

This work includes two entirely new Kummer algorithms that are essential for our signature scheme: verification relation testing (`Check`, Algorithm 3) and compression/decompression (`Compress` and `Decompress`, Algorithms 4 and 5). Both of these new techniques require a fair amount of technical development, which we begin in this section by recalling the basic Kummer equation and constants, and deconstructing the pseudo-doubling operation into a sequence of surfaces and maps that will play important roles later. Once the scene has been set, we will describe our signature verification algorithm in Sect. 5 and our point compression scheme in Sect. 6. The reader primarily interested in the resulting performance improvements may wish to skip directly to Sect. 7 on first reading.

The `Check`, `Compress`, and `Decompress` algorithms defined below require the following subroutines:

- `Had` implements a Hadamard transform. Given a vector (x_1, x_2, x_3, x_4) in $\mathbb{F}_p^4$, it returns $(x_1+x_2+x_3+x_4, x_1+x_2-x_3-x_4, x_1-x_2+x_3-x_4, x_1-x_2-x_3+x_4)$.

- `Dot` computes the sum of a 4-way multiplication. Given a pair of vectors (x_1, x_2, x_3, x_4) and (y_1, y_2, y_3, y_4) in $\mathbb{F}_p^4$, it returns $x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$.

4.1 Constants

Our Kummer surfaces are defined by four fundamental constants α_1, α_2, α_3, α_4 and four dual constants $\widehat{\alpha}_1$, $\widehat{\alpha}_2$, $\widehat{\alpha}_3$, and $\widehat{\alpha}_4$, which are related by

$$\begin{aligned} 2\widehat{\alpha}_1^2 &= \alpha_1^2 + \alpha_2^2 + \alpha_3^2 + \alpha_4^2\,, \\ 2\widehat{\alpha}_2^2 &= \alpha_1^2 + \alpha_2^2 - \alpha_3^2 - \alpha_4^2\,, \\ 2\widehat{\alpha}_3^2 &= \alpha_1^2 - \alpha_2^2 + \alpha_3^2 - \alpha_4^2\,, \\ 2\widehat{\alpha}_4^2 &= \alpha_1^2 - \alpha_2^2 - \alpha_3^2 + \alpha_4^2\,. \end{aligned}$$

We require all of the α_i and $\widehat{\alpha}_i$ to be nonzero. The fundamental constants determine the dual constants up to sign, and vice versa. These relations remain true when we exchange the α_i with the $\widehat{\alpha}_i$; we call this "swapping x with $\widehat{x}$" operation "dualizing". To make the symmetry in what follows clear, we define

$$\begin{aligned} \mu_1 &:= \alpha_1^2\,, & \epsilon_1 &:= \mu_2\mu_3\mu_4\,, & \kappa_1 &:= \epsilon_1 + \epsilon_2 + \epsilon_3 + \epsilon_4\,, \\ \mu_2 &:= \alpha_2^2\,, & \epsilon_2 &:= \mu_1\mu_3\mu_4\,, & \kappa_2 &:= \epsilon_1 + \epsilon_2 - \epsilon_3 - \epsilon_4\,, \\ \mu_3 &:= \alpha_3^2\,, & \epsilon_3 &:= \mu_1\mu_2\mu_4\,, & \kappa_3 &:= \epsilon_1 - \epsilon_2 + \epsilon_3 - \epsilon_4\,, \\ \mu_4 &:= \alpha_4^2\,, & \epsilon_4 &:= \mu_1\mu_2\mu_3\,, & \kappa_4 &:= \epsilon_1 - \epsilon_2 - \epsilon_3 + \epsilon_4\,, \end{aligned}$$

along with their respective duals $\widehat{\mu}_i$, $\widehat{\epsilon}_i$, and $\widehat{\kappa}_i$. Note that

$$(\epsilon_1 : \epsilon_2 : \epsilon_3 : \epsilon_4) = (1/\mu_1 : 1/\mu_2 : 1/\mu_3 : 1/\mu_4)$$

and $\mu_i\mu_j - \mu_k\mu_l = \widehat{\mu}_i\widehat{\mu}_j - \widehat{\mu}_k\widehat{\mu}_l$ for $\{i, j, k, l\} = \{1, 2, 3, 4\}$. There are many clashing notational conventions for theta constants in the cryptographic Kummer literature; Table 1 provides a dictionary for converting between them.

Our applications use only the squared constants μ_i and $\widehat{\mu}_i$, so only they need be in $\mathbb{F}_p$. In practice we want them to be as "small" as possible, both to reduce the cost of multiplying by them and to reduce the cost of storing them. In fact, it follows from their definition that it is much easier to find simultaneously small μ_i and $\widehat{\mu}_i$ than it is to find simultaneously small α_i and $\widehat{\alpha}_i$ (or a mixture of the two); this is ultimately why we prefer the squared surface for scalar multiplication. We note that if the μ_i are very small, then the ϵ_i and κ_i are also small, and the same goes for their duals. While we will never actually compute with the unsquared constants, we need them to explain what is happening in the background below.

Finally, the Kummer surface equations involve some derived constants

$$E := \frac{16\alpha_1\alpha_2\alpha_3\alpha_4\widehat{\mu}_1\widehat{\mu}_2\widehat{\mu}_3\widehat{\mu}_4}{(\mu_1\mu_4 - \mu_2\mu_3)(\mu_1\mu_3 - \mu_2\mu_4)(\mu_1\mu_2 - \mu_3\mu_4)}\,,$$

$$F := 2\frac{\mu_1\mu_4 + \mu_2\mu_3}{\mu_1\mu_4 - \mu_2\mu_3}\,, \qquad G := 2\frac{\mu_1\mu_3 + \mu_2\mu_4}{\mu_1\mu_3 - \mu_2\mu_4}\,, \qquad H := 2\frac{\mu_1\mu_2 + \mu_3\mu_4}{\mu_1\mu_2 - \mu_3\mu_4}\,,$$

and their duals $\widehat{E}$, $\widehat{F}$, $\widehat{G}$, $\widehat{H}$. We observe that $E^2 = F^2 + G^2 + H^2 + FGH - 4$ and $\widehat{E}^2 = \widehat{F}^2 + \widehat{G}^2 + \widehat{H}^2 + \widehat{F}\widehat{G}\widehat{H} - 4$.

Table 1. Relations between our theta constants and others in selected related work

Source	Fundamental constants	Dual constants
[21] and [7]	$(a : b : c : d) = (\alpha_1 : \alpha_2 : \alpha_3 : \alpha_4)$	$(A : B : C : D) = (\widehat{\alpha}_1 : \widehat{\alpha}_2 : \widehat{\alpha}_3 : \widehat{\alpha}_4)$
[11]	$(a : b : c : d) = (\alpha_1 : \alpha_2 : \alpha_3 : \alpha_4)$	$(A : B : C : D) = (\widehat{\mu}_1 : \widehat{\mu}_2 : \widehat{\mu}_3 : \widehat{\mu}_4)$
[39]	$(a : b : c : d) = (\mu_1 : \mu_2 : \mu_3 : \mu_4)$	$(A : B : C : D) = (\widehat{\mu}_1 : \widehat{\mu}_2 : \widehat{\mu}_3 : \widehat{\mu}_4)$
[15]	$(\alpha : \beta : \gamma : \delta) = (\mu_1 : \mu_2 : \mu_3 : \mu_4)$	$(A : B : C : D) = (\widehat{\mu}_1 : \widehat{\mu}_2 : \widehat{\mu}_3 : \widehat{\mu}_4)$

4.2 Fast Kummer Surfaces

We compute all of the pseudoscalar multiplications in `qDSA` on the so-called **squared Kummer surface**.

$$\mathcal{K}^{\mathrm{Sqr}} : 4E^2 \cdot X_1X_2X_3X_4 = \begin{pmatrix} X_1^2 + X_2^2 + X_3^2 + X_4^2 - F(X_1X_4 + X_2X_3) \\ - G(X_1X_3 + X_2X_4) - H(X_1X_2 + X_3X_4) \end{pmatrix}^2 ,$$

which was proposed for factorization algorithms by the Chudnovskys [13], then later for Diffie–Hellman by Bernstein [6]. Since E only appears as a square, $\mathcal{K}^{\mathrm{Sqr}}$ is defined over $\mathbb{F}_p$. The zero point on $\mathcal{K}^{\mathrm{Sqr}}$ is $\pm 0 = (\mu_1 : \mu_2 : \mu_3 : \mu_4)$. In our implementations we used the `xDBLADD` and Montgomery ladder exactly as they were presented in [39, Algorithms 6–7] The pseudo-doubling `xDBL` on $\mathcal{K}^{\mathrm{Sqr}}$ is

$$\pm P = \left(X_1^P : X_2^P : X_3^P : X_4^P\right) \longmapsto \left(X_1^{[2]P} : X_2^{[2]P} : X_3^{[2]P} : X_4^{[2]P}\right) = \pm[2]P$$

where

$$X_1^{[2]P} = \epsilon_1(U_1 + U_2 + U_3 + U_4)^2, \qquad U_1 = \widehat{\epsilon}_1(X_1^P + X_2^P + X_3^P + X_4^P)^2, \tag{7}$$

$$X_2^{[2]P} = \epsilon_2(U_1 + U_2 - U_3 - U_4)^2, \qquad U_2 = \widehat{\epsilon}_2(X_1^P + X_2^P - X_3^P - X_4^P)^2, \tag{8}$$

$$X_3^{[2]P} = \epsilon_3(U_1 - U_2 + U_3 - U_4)^2, \qquad U_3 = \widehat{\epsilon}_3(X_1^P - X_2^P + X_3^P - X_4^P)^2, \tag{9}$$

$$X_4^{[2]P} = \epsilon_4(U_1 - U_2 - U_3 + U_4)^2, \qquad U_4 = \widehat{\epsilon}_4(X_1^P - X_2^P - X_3^P + X_4^P)^2 \tag{10}$$

for $\pm P$ with all $X_i^P \neq 0$; more complicated formulæ exist for other $\pm P$ (cf. Sect. 5.1).

4.3 Deconstructing Pseudo-doubling

Figure 1 deconstructs the pseudo-doubling on $\mathcal{K}^{\mathrm{Sqr}}$ from Sect. 4.2 into a cycle of atomic maps between different Kummer surfaces, which form a sort of hexagon. Starting at any one of the Kummers and doing a complete cycle of these maps carries out pseudo-doubling on that Kummer. Doing a half-cycle from a given Kummer around to its dual computes a $(2, 2)$-isogeny splitting pseudo-doubling.

Six different Kummer surfaces may seem like a lot to keep track of—even if there are really only three, together with their duals. However, the new surfaces are important, because they are crucial in deriving our `Check` routine (of course,

$$\begin{tikzcd}
& \mathcal{K}^{\mathrm{Can}} \arrow[r, "\mathcal{S}", "(2{,}2)"'] & \mathcal{K}^{\mathrm{Sqr}} \arrow[dr, "\mathcal{H}", "\cong"'] & \\
\widehat{\mathcal{K}}^{\mathrm{Int}} \arrow[ur, "\mathcal{C}", "\cong"'] & & & \mathcal{K}^{\mathrm{Int}} \arrow[dl, "\widehat{\mathcal{C}}", "\cong"'] \\
& \widehat{\mathcal{K}}^{\mathrm{Sqr}} \arrow[ul, "\widehat{\mathcal{H}}", "\cong"'] & \widehat{\mathcal{K}}^{\mathrm{Can}} \arrow[l, "(2{,}2)"', "\widehat{\mathcal{S}}"] &
\end{tikzcd}$$

Fig. 1. Decomposition of pseudo-doubling on fast Kummer surfaces into a cycle of morphisms. Here, $\mathcal{K}^{\mathrm{Sqr}}$ is the "squared" surface we mostly compute with; $\mathcal{K}^{\mathrm{Can}}$ is the related "canonical" surface; and $\mathcal{K}^{\mathrm{Int}}$ is a new "intermediate" surface which we use in signature verification. (The surfaces $\widehat{\mathcal{K}}^{\mathrm{Sqr}}$, $\widehat{\mathcal{K}}^{\mathrm{Can}}$, and $\widehat{\mathcal{K}}^{\mathrm{Int}}$ are their duals.)

once the algorithm has been written down, the reader is free to forget about the existence of these other surfaces).

The cycle actually begins one step before $\mathcal{K}^{\mathrm{Sqr}}$, with the **canonical surface**

$$\mathcal{K}^{\mathrm{Can}} : 2E \cdot T_1T_2T_3T_4 = \begin{array}{l} T_1^4 + T_2^4 + T_3^4 + T_4^4 - F(T_1^2T_4^2 + T_2^2T_3^2) \\ - G(T_1^2T_3^2 + T_2^2T_4^2) - H(T_1^2T_2^2 + T_3^2T_4^2)\,. \end{array}$$

This was the model proposed for cryptographic applications by Gaudry in [21]; we call it "canonical" because it is the model arising from a canonical basis of theta functions of level $(2,2)$.

Now we can begin our tour around the hexagon, moving from $\mathcal{K}^{\mathrm{Can}}$ to $\mathcal{K}^{\mathrm{Sqr}}$ via the **squaring** map

$$\mathcal{S} : \left(T_1 : T_2 : T_3 : T_4\right) \longmapsto \left(X_1 : X_2 : X_3 : X_4\right) = \left(T_1^2 : T_2^2 : T_3^2 : T_4^3\right),$$

which corresponds to a $(2,2)$-isogeny of Jacobians. Moving on from $\mathcal{K}^{\mathrm{Sqr}}$, the **Hadamard transform** isomorphism

$$\mathcal{H} : (X_1 : X_2 : X_3 : X_4) \longmapsto (Y_1 : Y_2 : Y_3 : Y_4) = \begin{pmatrix} \ X_1 + X_2 + X_3 + X_4 \\ :X_1 + X_2 - X_3 - X_4 \\ :X_1 - X_2 + X_3 - X_4 \\ :X_1 - X_2 - X_3 + X_4 \end{pmatrix}$$

takes us into a third kind of Kummer, which we call the **intermediate surface**:

$$\mathcal{K}^{\mathrm{Int}} : \frac{2\widehat{E}}{\alpha_1\alpha_2\alpha_3\alpha_4} \cdot Y_1Y_2Y_3Y_4 = \begin{array}{l} \frac{Y_1^4}{\mu_1^2} + \frac{Y_2^4}{\mu_2^2} + \frac{Y_3^4}{\mu_3^2} + \frac{Y_4^4}{\mu_4^2} - \widehat{F}\left(\frac{Y_1^2}{\mu_1}\frac{Y_4^2}{\mu_4} + \frac{Y_2^2}{\mu_2}\frac{Y_3^2}{\mu_3}\right) \\ - \widehat{G}\left(\frac{Y_1^2}{\mu_1}\frac{Y_3^2}{\mu_3} + \frac{Y_2^2}{\mu_2}\frac{Y_4^2}{\mu_4}\right) - \widehat{H}\left(\frac{Y_1^2}{\mu_1}\frac{Y_2^2}{\mu_2} + \frac{Y_3^2}{\mu_3}\frac{Y_4^2}{\mu_4}\right). \end{array}$$

We will use $\mathcal{K}^{\mathrm{Int}}$ for signature verification. Now the **dual scaling** isomorphism

$$\widehat{\mathcal{C}} : \left(Y_1 : Y_2 : Y_3 : Y_4\right) \longmapsto \left(\widehat{T}_1 : \widehat{T}_2 : \widehat{T}_3 : \widehat{T}_4\right) = \left(Y_1/\widehat{\alpha}_1 : Y_2/\widehat{\alpha}_2 : Y_3/\widehat{\alpha}_3 : Y_4/\widehat{\alpha}_4\right)$$

takes us into the **dual canonical surface**

$$\widehat{\mathcal{K}}^{\mathrm{Can}} : 2\widehat{E} \cdot \widehat{T}_1\widehat{T}_2\widehat{T}_3\widehat{T}_4 = \begin{array}{l} \widehat{T}_1^4 + \widehat{T}_2^4 + \widehat{T}_3^4 + \widehat{T}_4^4 - \widehat{F}(\widehat{T}_1^2\widehat{T}_4^2 + \widehat{T}_2^2\widehat{T}_3^2) \\ - \widehat{G}(\widehat{T}_1^2\widehat{T}_3^2 + \widehat{T}_2^2\widehat{T}_4^2) - \widehat{H}(\widehat{T}_1^2\widehat{T}_2^2 + \widehat{T}_3^2\widehat{T}_4^2). \end{array}$$

We are now halfway around the hexagon; the return journey is simply the dual of the outbound trip. The **dual squaring** map

$$\widehat{\mathcal{S}} : \left(\widehat{T}_1 : \widehat{T}_2 : \widehat{T}_3 : \widehat{T}_4\right) \longmapsto \left(\widehat{X}_1 : \widehat{X}_2 : \widehat{X}_3 : \widehat{X}_4\right) = \left(\widehat{T}_1^2 : \widehat{T}_2^2 : \widehat{T}_3^2 : \widehat{T}_4^3\right),$$

another $(2, 2)$-isogeny, carries us into the **dual squared surface**

$$\widehat{\mathcal{K}}^{\mathrm{Sqr}} : 4\widehat{E}^2 \cdot \widehat{X}_1\widehat{X}_2\widehat{X}_3\widehat{X}_4 = \begin{pmatrix} \widehat{X}_1^2 + \widehat{X}_2^2 + \widehat{X}_3^2 + \widehat{X}_4^2 - \widehat{F}(\widehat{X}_1\widehat{X}_4 + \widehat{X}_2\widehat{X}_3) \\ - \widehat{G}(\widehat{X}_1\widehat{X}_3 + \widehat{X}_2\widehat{X}_4) - \widehat{H}(\widehat{X}_1\widehat{X}_2 + \widehat{X}_3\widehat{X}_4) \end{pmatrix}^2,$$

before the **dual Hadamard transform**

$$\widehat{\mathcal{H}} : \left(\widehat{X}_1 : \widehat{X}_2 : \widehat{X}_3 : \widehat{X}_4\right) \longmapsto \left(\widehat{Y}_1 : \widehat{Y}_2 : \widehat{Y}_3 : \widehat{Y}_4\right) = \begin{pmatrix} \widehat{X}_1 + \widehat{X}_2 + \widehat{X}_3 + \widehat{X}_4 \\ : \widehat{X}_1 + \widehat{X}_2 - \widehat{X}_3 - \widehat{X}_4 \\ : \widehat{X}_1 - \widehat{X}_2 + \widehat{X}_3 - \widehat{X}_4 \\ : \widehat{X}_1 - \widehat{X}_2 - \widehat{X}_3 + \widehat{X}_4 \end{pmatrix}$$

takes us into the **dual intermediate surface**

$$\widehat{\mathcal{K}}^{\mathrm{Int}} : \frac{2E}{\alpha_1\alpha_2\alpha_3\alpha_4} \cdot \widehat{Y}_1\widehat{Y}_2\widehat{Y}_3\widehat{Y}_4 = \begin{array}{l} \frac{\widehat{Y}_1^4}{\mu_1^2} + \frac{\widehat{Y}_2^4}{\mu_2^2} + \frac{\widehat{Y}_3^4}{\mu_3^2} + \frac{\widehat{Y}_4^4}{\mu_4^2} - \widehat{F}\left(\frac{\widehat{Y}_1^2}{\mu_1}\frac{\widehat{Y}_4^2}{\mu_4} - \frac{\widehat{Y}_2^2}{\mu_2}\frac{\widehat{Y}_3^2}{\mu_3}\right) \\ - \widehat{G}\left(\frac{\widehat{Y}_1^2}{\mu_1}\frac{\widehat{Y}_3^2}{\mu_3} - \frac{\widehat{Y}_2^2}{\mu_2}\frac{\widehat{Y}_4^2}{\mu_4}\right) - \widehat{H}\left(\frac{\widehat{Y}_1^2}{\mu_1}\frac{\widehat{Y}_2^2}{\mu_2} - \frac{\widehat{Y}_3^2}{\mu_3}\frac{\widehat{Y}_4^2}{\mu_4}\right). \end{array}$$

A final **scaling** isomorphism

$$\mathcal{C} : \left(\widehat{Y}_1 : \widehat{Y}_2 : \widehat{Y}_3 : \widehat{Y}_4\right) \longmapsto \left(T_1 : T_2 : T_3 : T_4\right) = \left(\widehat{Y}_1/\alpha_1 : \widehat{Y}_2/\alpha_2 : \widehat{Y}_3/\alpha_3 : \widehat{Y}_4/\alpha_4\right)$$

takes us from $\widehat{\mathcal{K}}^{\mathrm{Int}}$ back to $\mathcal{K}^{\mathrm{Can}}$, where we started.

The canonical surfaces $\mathcal{K}^{\mathrm{Can}}$ resp. $\widehat{\mathcal{K}}^{\mathrm{Can}}$ are only defined over $\mathbb{F}_p(\alpha_1\alpha_2\alpha_3\alpha_4)$ resp. $\mathbb{F}_p(\widehat{\alpha}_1\widehat{\alpha}_2\widehat{\alpha}_3\widehat{\alpha}_4)$, while the scaling isomorphisms $\widehat{\mathcal{C}}$ resp. $\mathcal{C}$ are defined over $\mathbb{F}_p(\widehat{\alpha}_1, \widehat{\alpha}_2, \widehat{\alpha}_3, \widehat{\alpha}_4)$ resp. $\mathbb{F}_p(\alpha_1, \alpha_2, \alpha_3, \alpha_4)$. Everything else is defined over $\mathbb{F}_p$.

We confirm that one cycle around the hexagon, starting and ending on $\mathcal{K}^{\mathrm{Sqr}}$, computes the pseudo-doubling of Eqs. (7), (8), (9), and (10). Similarly, one cycle around the hexagon starting and ending on $\mathcal{K}^{\mathrm{Can}}$ computes Gaudry's pseudo-doubling from [21, Sect. 3.2].

5 Signature Verification on Kummer Surfaces

To verify signatures in the Kummer surface implementation, we need to supply a `Check` algorithm which, given $\pm P$, $\pm Q$, and $\pm R$ on $\mathcal{K}^{\mathrm{Sqr}}$, decides whether $\pm R \in \{\pm(P+Q), \pm(P-Q)\}$. For the elliptic version of `qDSA` described in Sect. 3, we saw that this came down to checking that $\pm R$ satisfied one quadratic relation whose three coefficients were biquadratic forms in $\pm P$ and $\pm Q$. The same principle extends to Kummer surfaces, where the pseudo-group law is similarly defined by biquadratic forms; but since Kummer surfaces are defined in terms of four coordinates (as opposed to the two coordinates of the x-line), this time there are six simple quadratic relations to verify, with a total of ten coefficient forms.

5.1 Biquadratic Forms and Pseudo-addition

Let $\mathcal{K}$ be a Kummer surface. If $\pm P$ is a point on $\mathcal{K}$, then we write $(Z_1^P : Z_2^P : Z_3^P : Z_4^P)$ for its projective coordinates. The classical theory of abelian varieties tells us that there exist biquadratic forms B_{ij} for $1 \le i, j \le 4$ such that for all $\pm P$ and $\pm Q$, if $\pm S = \pm(P+Q)$ and $\pm D = \pm(P-Q)$ then

$$\left(Z_i^S Z_j^D + Z_j^S Z_i^D\right)_{i,j=1}^4 = \lambda \left(B_{ij}(Z_1^P, Z_2^P, Z_3^P, Z_4^P, Z_1^Q, Z_2^Q, Z_3^Q, Z_4^Q)\right)_{i,j=1}^4 \tag{11}$$

where $\lambda \in \Bbbk^\times$ is some common projective factor depending only on the affine representatives chosen for $\pm P$, $\pm Q$, $\pm(P+Q)$ and $\pm(P-Q)$. These biquadratic forms are the foundation of pseudo-addition and doubling laws on $\mathcal{K}$: if the "difference" $\pm D$ is known, then we can use the B_{ij} to compute $\pm S$.

Proposition 4. *Let $\{B_{ij} : 1 \le i, j \le 4\}$ be a set of biquadratic forms on $\mathcal{K} \times \mathcal{K}$ satisfying Eq. (11) for all $\pm P$, $\pm Q$, $\pm(P+Q)$, and $\pm(P-Q)$. Then*

$$\pm R = (Z_1^R : Z_2^R : Z_3^R : Z_4^R) \in \{\pm(P+Q), \pm(P-Q)\}$$

if and only if (writing B_{ij} for $B_{ij}(Z_1^P, \ldots, Z_4^Q)$) we have

$$B_{jj} \cdot (Z_i^R)^2 - 2B_{ij} \cdot Z_i^R Z_j^R + B_{ii} \cdot (Z_j^R)^2 = 0 \quad \textit{for all}\, 1 \le i < j \le 4. \tag{12}$$

Proof. Looking at Eq. (11), we see that the system of six quadratics from Eq. (12) cuts out a zero-dimensional degree-2 subscheme of $\mathcal{K}$: that is, the pair of points $\{\pm(P+Q), \pm(P-Q)\}$ (which may coincide). Hence, if $(Z_1^R : Z_2^R : Z_3^R : Z_4^R) = \pm R$ satisfies all of the equations, then it must be one of them. □

5.2 Deriving Efficiently Computable Forms

Proposition 4 is the exact analogue of Proposition 3 for Kummer surfaces. All that we need to turn it into a `Check` algorithm for `qDSA` is an explicit and efficiently computable representation of the B_{ij}. These forms depend on the projective model of the Kummer surface; so we write B_{ij}^{Can}, B_{ij}^{Sqr}, and B_{ij}^{Int} for the forms on the canonical, squared, and intermediate surfaces.

On the canonical surface, the forms B_{ij}^{Can} are classical (see e.g. [3, Sect. 2.2]). The on-diagonal forms B_{ii}^{Can} are

$$B_{11}^{\text{Can}} = \frac{1}{4}\Big(\frac{V_1}{\widehat{\mu}_1} + \frac{V_2}{\widehat{\mu}_2} + \frac{V_3}{\widehat{\mu}_3} + \frac{V_4}{\widehat{\mu}_4}\Big), \quad B_{22}^{\text{Can}} = \frac{1}{4}\Big(\frac{V_1}{\widehat{\mu}_1} + \frac{V_2}{\widehat{\mu}_2} - \frac{V_3}{\widehat{\mu}_3} - \frac{V_4}{\widehat{\mu}_4}\Big), \tag{13}$$

$$B_{33}^{\text{Can}} = \frac{1}{4}\Big(\frac{V_1}{\widehat{\mu}_1} - \frac{V_2}{\widehat{\mu}_2} + \frac{V_3}{\widehat{\mu}_3} - \frac{V_4}{\widehat{\mu}_4}\Big), \quad B_{44}^{\text{Can}} = \frac{1}{4}\Big(\frac{V_1}{\widehat{\mu}_1} - \frac{V_2}{\widehat{\mu}_2} - \frac{V_3}{\widehat{\mu}_3} + \frac{V_4}{\widehat{\mu}_4}\Big), \tag{14}$$

where

$$\begin{aligned}
V_1 &= \big((T_1^P)^2 + (T_2^P)^2 + (T_3^P)^2 + (T_4^P)^2\big)\big((T_1^Q)^2 + (T_2^Q)^2 + (T_3^Q)^2 + (T_4^Q)^2\big),\\
V_2 &= \big((T_1^P)^2 + (T_2^P)^2 - (T_3^P)^2 - (T_4^P)^2\big)\big((T_1^Q)^2 + (T_2^Q)^2 - (T_3^Q)^2 - (T_4^Q)^2\big),\\
V_3 &= \big((T_1^P)^2 - (T_2^P)^2 + (T_3^P)^2 - (T_4^P)^2\big)\big((T_1^Q)^2 - (T_2^Q)^2 + (T_3^Q)^2 - (T_4^Q)^2\big),\\
V_4 &= \big((T_1^P)^2 - (T_2^P)^2 - (T_3^P)^2 + (T_4^P)^2\big)\big((T_1^Q)^2 - (T_2^Q)^2 - (T_3^Q)^2 + (T_4^Q)^2\big),
\end{aligned}$$

while the off-diagonal forms B_{ij} with $i \neq j$ are

$$B_{ij}^{\mathrm{Can}} = \frac{2}{\widehat{\mu}_i\widehat{\mu}_j - \widehat{\mu}_k\widehat{\mu}_l}\left(\begin{array}{l} \alpha_i\alpha_j\big(T_i^P T_j^P T_i^Q T_j^Q + T_k^P T_l^P T_k^Q T_l^Q\big) \\ -\,\alpha_k\alpha_l\big(T_i^P T_j^P T_k^Q T_l^Q + T_k^P T_l^P T_i^Q T_j^Q\big) \end{array}\right) \tag{15}$$

where $\{i, j, k, l\} = \{1, 2, 3, 4\}$.

All of these forms can be efficiently evaluated. The off-diagonal B_{ij}^{Can} have a particularly compact shape, while the symmetry of the on-diagonal B_{ii}^{Can} makes them particularly easy to compute simultaneously: indeed, that is exactly what we do in Gaudry's fast pseudo-addition algorithm for $\mathcal{K}^{\mathrm{Can}}$ [21, Sect. 3.2].

Ideally, we would like to evaluate the B_{ij}^{Sqr} on $\mathcal{K}^{\mathrm{Sqr}}$, since that is where our inputs $\pm P$, $\pm Q$, and $\pm R$ live. We can compute the B_{ij}^{Sqr} by dualizing the B_{ij}^{Can}, then pulling the $\widehat{B}_{ij}^{\mathrm{Can}}$ on $\widehat{\mathcal{K}}^{\mathrm{Can}}$ back to $\mathcal{K}^{\mathrm{Sqr}}$ via $\widehat{\mathcal{C}} \circ \mathcal{H}$. But while the resulting on-diagonal B_{ii}^{Sqr} maintain the symmetry and efficiency of the B_{ii}^{Can},[4] the off-diagonal B_{ij}^{Sqr} turn out to be much less pleasant, with less apparent exploitable symmetry. For our applications, this means that evaluating B_{ij}^{Sqr} for $i \neq j$ implies taking a significant hit in terms of stack and code size, not to mention time.

We could avoid this difficulty by mapping the inputs of `Check` from $\mathcal{K}^{\mathrm{Sqr}}$ into $\widehat{\mathcal{K}}^{\mathrm{Can}}$, and then evaluating the $\widehat{B}_{ij}^{\mathrm{Can}}$. But this would involve using—and, therefore, storing— the four large unsquared $\widehat{\alpha}_i$, which is an important drawback.

Why do the nice $\widehat{B}_{ij}^{\mathrm{Can}}$ become so ugly when pulled back to $\mathcal{K}^{\mathrm{Sqr}}$? The map $\widehat{\mathcal{C}} : \mathcal{K}^{\mathrm{Int}} \to \widehat{\mathcal{K}}^{\mathrm{Can}}$ has no impact on the shape or number of monomials, so most of the ugliness is due to the Hadamard transform $\mathcal{H} : \mathcal{K}^{\mathrm{Sqr}} \to \mathcal{K}^{\mathrm{Int}}$. In particular, if we only pull back the $\widehat{B}_{ij}^{\mathrm{Can}}$ as far as $\mathcal{K}^{\mathrm{Int}}$, then the resulting B_{ij}^{Int} retain the nice form of the B_{ij}^{Can} but do not involve the $\widehat{\alpha}_i$. This fact prompts our solution: we map $\pm P$, $\pm Q$, and $\pm R$ through $\mathcal{H}$ onto $\mathcal{K}^{\mathrm{Int}}$, and verify using the forms B_{ij}^{Int}.

Theorem 1. *Up to a common projective factor, the on-diagonal biquadratic forms on the intermediate surface $\mathcal{K}^{\mathrm{Int}}$ are*

$$B_{11}^{\mathrm{Int}} = \widehat{\mu}_1\left(\kappa_1 F_1 + \kappa_2 F_2 + \kappa_3 F_3 + \kappa_4 F_4\right), \tag{16}$$

$$B_{22}^{\mathrm{Int}} = \widehat{\mu}_2\left(\kappa_2 F_1 + \kappa_1 F_2 + \kappa_4 F_3 + \kappa_3 F_4\right), \tag{17}$$

$$B_{33}^{\mathrm{Int}} = \widehat{\mu}_3\left(\kappa_3 F_1 + \kappa_4 F_2 + \kappa_1 F_3 + \kappa_2 F_4\right), \tag{18}$$

$$B_{44}^{\mathrm{Int}} = \widehat{\mu}_4\left(\kappa_4 F_1 + \kappa_3 F_2 + \kappa_2 F_3 + \kappa_1 F_4\right), \tag{19}$$

[4] As they should, since they are the basis of the efficient pseudo-addition on $\mathcal{K}^{\mathrm{Sqr}}$!.

where

$$F_1 = P_1Q_1 + P_2Q_2 + P_3Q_3 + P_4Q_4, \quad F_2 = P_1Q_2 + P_2Q_1 + P_3Q_4 + P_4Q_3,$$
$$F_3 = P_1Q_3 + P_3Q_1 + P_2Q_4 + P_4Q_2, \quad F_4 = P_1Q_4 + P_4Q_1 + P_2Q_3 + P_3Q_2,$$

where $P_i = \widehat{\epsilon}_i(Y_i^P)^2$ *and* $Q_i = \widehat{\epsilon}_i(Y_i^Q)^2$ *for* $1 \leq i \leq 4$. *Up to the same common projective factor, the off-diagonal forms are*

$$B_{ij}^{\mathrm{Int}} = C \cdot C_{ij} \cdot \Big(\widehat{\mu}_k\widehat{\mu}_l\big(Y_{ij}^P - Y_{kl}^P\big)\big(Y_{ij}^Q - Y_{kl}^Q\big) + \big(\widehat{\mu}_i\widehat{\mu}_j - \widehat{\mu}_k\widehat{\mu}_l\big)Y_{kl}^PY_{kl}^Q\Big) \qquad (20)$$

for $\{i,j,k,l\} = \{1,2,3,4\}$ *where* $C_{ij} := \widehat{\mu}_i\widehat{\mu}_j(\widehat{\mu}_i\widehat{\mu}_k - \widehat{\mu}_j\widehat{\mu}_l)(\widehat{\mu}_i\widehat{\mu}_l - \widehat{\mu}_j\widehat{\mu}_k)$, $Y_{ij}^P := Y_i^PY_j^P$, $Y_{ij}^Q := Y_i^QY_j^Q$, *and*

$$C := \frac{8(\mu_1\mu_2\mu_3\mu_4)(\widehat{\mu}_1\widehat{\mu}_2\widehat{\mu}_3\widehat{\mu}_4)}{(\widehat{\mu}_1\widehat{\mu}_2 - \widehat{\mu}_3\widehat{\mu}_4)(\widehat{\mu}_1\widehat{\mu}_3 - \widehat{\mu}_2\widehat{\mu}_4)(\widehat{\mu}_1\widehat{\mu}_4 - \widehat{\mu}_2\widehat{\mu}_3)}\,.$$

Proof. By definition, $\widehat{T}_i^S\widehat{T}_j^D + \widehat{T}_j^S\widehat{T}_i^D = \widehat{B}_{ij}^{\mathrm{Can}}(\widehat{T}_1^P, \ldots, \widehat{T}_4^Q)$. Pulling back via $\widehat{\mathcal{C}}$ using $\widehat{T}_i = Y_i/\widehat{\alpha}_i$ yields

$$\begin{aligned} B_{ij}^{\mathrm{Int}}(Y_1^P, \ldots, Y_4^Q) = Y_i^SY_j^D + Y_j^SY_i^D &= \widehat{\alpha}_i\widehat{\alpha}_j\big(\widehat{T}_i^S\widehat{T}_j^D + \widehat{T}_j^S\widehat{T}_i^D\big) \\ &= \widehat{\alpha}_i\widehat{\alpha}_j \cdot \widehat{B}_{ij}^{\mathrm{Can}}(\widehat{T}_1^P, \ldots, \widehat{T}_4^Q) \\ &= \widehat{\alpha}_i\widehat{\alpha}_j \cdot \widehat{B}_{ij}^{\mathrm{Can}}(Y_1^P/\widehat{\alpha}_1, \ldots, Y_4^Q/\widehat{\alpha}_4)\,. \end{aligned}$$

Dualizing the B_{ij}^{Can} from Eqs. (13), (14), and (15), we find

$$\begin{aligned} B_{11}^{\mathrm{Int}} &= \widehat{\mu}_1/\big(4\mu_1\mu_2\mu_3\mu_4(\widehat{\mu}_1\widehat{\mu}_2\widehat{\mu}_3\widehat{\mu}_4)^2\big) \cdot \big(\kappa_1F_1 + \kappa_2F_2 + \kappa_3F_3 + \kappa_4F_4\big), \\ B_{22}^{\mathrm{Int}} &= \widehat{\mu}_2/\big(4\mu_1\mu_2\mu_3\mu_4(\widehat{\mu}_1\widehat{\mu}_2\widehat{\mu}_3\widehat{\mu}_4)^2\big) \cdot \big(\kappa_2F_1 + \kappa_1F_2 + \kappa_4F_3 + \kappa_3F_4\big), \\ B_{33}^{\mathrm{Int}} &= \widehat{\mu}_3/\big(4\mu_1\mu_2\mu_3\mu_4(\widehat{\mu}_1\widehat{\mu}_2\widehat{\mu}_3\widehat{\mu}_4)^2\big) \cdot \big(\kappa_3F_1 + \kappa_4F_2 + \kappa_1F_3 + \kappa_2F_4\big), \\ B_{44}^{\mathrm{Int}} &= \widehat{\mu}_4/\big(4\mu_1\mu_2\mu_3\mu_4(\widehat{\mu}_1\widehat{\mu}_2\widehat{\mu}_3\widehat{\mu}_4)^2\big) \cdot \big(\kappa_4F_1 + \kappa_3F_2 + \kappa_2F_3 + \kappa_1F_4\big), \end{aligned}$$

while the off-diagonal forms B_{ij} with $i \neq j$ are

$$B_{ij}^{\mathrm{Int}} = \frac{2}{\widehat{\mu}_k\widehat{\mu}_l(\widehat{\mu}_i\widehat{\mu}_j - \widehat{\mu}_k\widehat{\mu}_l)}\begin{pmatrix} \widehat{\mu}_k\widehat{\mu}_l\big(Y_{ij}^P - Y_{kl}^P\big)\big(Y_{ij}^Q - Y_{kl}^Q\big) \\ + (\widehat{\mu}_i\widehat{\mu}_j - \widehat{\mu}_k\widehat{\mu}_l)Y_{kl}^PY_{kl}^Q \end{pmatrix}$$

for $\{i,j,k,l\} = \{1,2,3,4\}$. Multiplying all of these forms by a common projective factor of $4(\mu_1\mu_2\mu_3\mu_4)(\widehat{\mu}_1\widehat{\mu}_2\widehat{\mu}_3\widehat{\mu}_4)^2$ eliminates the denominators in the coefficients, and yields the forms of the theorem. □

5.3 Signature Verification

We are now finally ready to implement the `Check` algorithm for $\mathcal{K}^{\mathrm{Sqr}}$. Algorithm 3 does this by applying $\mathcal{H}$ to its inputs, then using the biquadratic forms of Theorem 1. Its correctness is implied by Proposition 4.

Algorithm 3. Checking the verification relation for points on $\mathcal{K}^{\mathrm{Sqr}}$

```
function Check
    Input: $\pm P$, $\pm Q$, $\pm R$ in $\mathcal{K}^{\mathrm{Sqr}}(\mathbb{F}_p)$
    Output: True if $\pm R \in \{\pm(P+Q), \pm(P-Q)\}$, False otherwise
    Cost: $76\mathbf{M} + 8\mathbf{S} + 88\mathbf{C} + 42\mathbf{a} + 42\mathbf{s}$
    $(\mathsf{Y}^P, \mathsf{Y}^Q) \leftarrow (\mathtt{Had}(\pm P), \mathtt{Had}(\pm Q))$
    $(\mathsf{B}_{11}, \mathsf{B}_{22}, \mathsf{B}_{33}, \mathsf{B}_{44}) \leftarrow \mathtt{BiiValues}(\mathsf{Y}^P, \mathsf{Y}^Q)$
    $\mathsf{Y}^R \leftarrow \mathtt{Had}(\pm R)$
    for $(i,j)$ in $\{(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)\}$ do
        $\mathsf{LHS} \leftarrow \mathsf{B}_{ii} \cdot (\mathsf{Y}_j^R)^2 + \mathsf{B}_{jj} \cdot (\mathsf{Y}_i^R)^2$
        $\mathsf{B}_{ij} \leftarrow \mathtt{BijValue}(\mathsf{Y}^P, \mathsf{Y}^Q, (i,j))$
        $\mathsf{RHS} \leftarrow 2\mathsf{B}_{ij} \cdot \mathsf{Y}_i^R \cdot \mathsf{Y}_j^R$
        if $\mathsf{LHS} \neq \mathsf{RHS}$ then return False
    return True
function BiiValues
    Input: $\pm P$, $\pm Q$ in $\mathcal{K}^{\mathrm{Int}}(\mathbb{F}_p)$
    Output: $(B_{ii}^{\mathrm{Int}}(\pm P, \pm Q))_{i=1}^4$ in $\mathbb{F}_p^4$
    Cost: $16\mathbf{M} + 8\mathbf{S} + 28\mathbf{C} + 24\mathbf{a}$
    // Use Eqs. (16), (17), (8) and (19) in Theorem 1
function BijValue
    Input: $\pm P$, $\pm Q$ in $\mathcal{K}^{\mathrm{Int}}(\mathbb{F}_p)$ and $(i,j)$ with $1 \leq i,j \leq 4$ and $i \neq j$
    Output: $B_{ij}^{\mathrm{Int}}(\pm P, \pm Q)$ in $\mathbb{F}_p$
    Cost: $10\mathbf{M} + 10\mathbf{C} + 1\mathbf{a} + 5\mathbf{s}$
    // Use Eq. (20) in Theorem 1
```

5.4 Using Cryptographic Parameters

Gaudry and Schost take $p = 2^{127} - 1$ and $(\mu_1 : \mu_2 : \mu_3 : \mu_4) = (-11 : 22 : 19 : 3)$ in [22]. We also need the constants $(\widehat{\mu}_1 : \widehat{\mu}_2 : \widehat{\mu}_3 : \widehat{\mu}_4) = (-33 : 11 : 17 : 49)$, $(\kappa_1 : \kappa_2 : \kappa_3 : \kappa_4) = (-4697 : 5951 : 5753 : -1991)$, and $(\widehat{\epsilon}_1 : \widehat{\epsilon}_2 : \widehat{\epsilon}_3 : \widehat{\epsilon}_4) = (-833 : 2499 : 1617 : 561)$.[5] In practice, where these constants are "negative", we reverse their sign and amend the formulæ above accordingly. All of these constants are small, and fit into one or two bytes each (and the $\widehat{\epsilon}_i$ are already stored for use in `Ladder`). We store one large constant

$$C = \mathtt{0x40F50EEFA320A2DD46F7E3D8CDDDA843},$$

and recompute the C_{ij} on the fly.

[5] Following the definitions of Sect. 4.1, the $\widehat{\mu}_i$ are scaled by -2, the $\widehat{\epsilon}_i$ by $1/11$, and C by $2/11^2$. These changes influence the B_{ij}^{Int}, but only up to the same projective factor.

6 Kummer Point Compression

Our public keys are points on $\mathcal{K}^{\mathrm{Sqr}}$, and each signature includes one point on $\mathcal{K}^{\mathrm{Sqr}}$. Minimizing the space required by Kummer points is therefore essential.

A projective Kummer point is composed of four field elements; normalizing by dividing through by a nonzero coordinate reduces us to three field elements (this can also be achieved using Bernstein's "wrapping" technique [6], as in [7, 39]). But we are talking about Kummer *surfaces*—two-dimensional objects—so we might hope to compress to two field elements, plus a few bits to enable us to correctly recover the whole Kummer point. This is analogous to elliptic curve point compression, where we compress projective points $(X : Y : Z)$ by normalizing to $(x, y) = (X/Z, Y/Z)$, then storing (x, σ), where σ is a bit indicating the "sign" of y. Decompressing the datum (x, σ) to $(X : Y : Z) = (x : y : 1)$ then requires solving a simple quadratic to recover the correct y-coordinate.

For some reason, no such Kummer point compression method has explicitly appeared in the literature. Bernstein remarked in 2006 that if we compress a Kummer point to two coordinates, then decompression appears to require solving a complicated quartic equation [6]. This would be much more expensive than computing the single square root required for elliptic decompression; this has perhaps discouraged implementers from attempting to compress Kummer points.

But while it may not always be obvious from their defining equations, the classical theory tells us that every Kummer is in fact a double cover of $\mathbb{P}^2$, just as elliptic curves are double covers of $\mathbb{P}^1$. We use this principle below to show that we can always compress any Kummer point to two field elements plus two auxiliary bits, and then decompress by solving a quadratic. In our applications, this gives us a convenient packaging of Kummer points in exactly 256 bits.

6.1 The General Principle

First, we sketch a general method for Kummer point compression that works for any Kummer presented as a singular quartic surface in $\mathbb{P}^3$.

Recall that if N is any point in $\mathbb{P}^3$, then projection away from N defines a map $\pi_N : \mathbb{P}^3 \to \mathbb{P}^2$ sending points in $\mathbb{P}^3$ on the same line through N to the same point in $\mathbb{P}^2$. (The map π_N is only a rational map, and not a morphism; the image of N itself is not well-defined.) Now, let N be a node of a Kummer surface $\mathcal{K}$: that is, N is one of the 16 singular points of $\mathcal{K}$. The restriction of π_N to $\mathcal{K}$ forms a double cover of $\mathbb{P}^2$. By definition, π_N maps the points on $\mathcal{K}$ that lie on the same line through N to the same point of $\mathbb{P}^2$. Now $\mathcal{K}$ has degree 4, so each line in $\mathbb{P}^3$ intersects $\mathcal{K}$ in four points; but since N is a double point of $\mathcal{K}$, every line through N intersects $\mathcal{K}$ at N *twice*, and then in two other points. These two remaining points may be "compressed" to their common image in $\mathbb{P}^2$ under π_N, plus a single bit to distinguish the appropriate preimage.

To make this more concrete, let L_1, L_2, and L_3 be linearly independent linear forms on $\mathbb{P}^3$ vanishing on N; then N is the intersection of the three planes in $\mathbb{P}^3$ cut out by the L_i. We can now realise the projection $\pi_N : \mathcal{K} \to \mathbb{P}^2$ as

$$\pi_N : (P_1 : \cdots : P_4) \longmapsto \big(L_1(P_1, \ldots, P_4) : L_2(P_1, \ldots, P_4) : L_3(P_1, \ldots, P_4)\big) .$$

Replacing (L_1, L_2, L_3) with another basis of $\langle L_1, L_2, L_3\rangle$ yields another projection, which corresponds to composing π_N with a linear automorphism of $\mathbb{P}^2$.

If L_1, L_2, and L_3 are chosen as above to vanish on N, and L_4 is any linear form not in $\langle L_1, L_2, L_3\rangle$, then the fact that π_N is a double cover of the (L_1, L_2, L_3)-plane implies that the defining equation of $\mathcal{K}$ can be rewritten in the form

$$\mathcal{K} : K_2(L_1, L_2, L_3)L_4^2 - 2K_3(L_1, L_2, L_3)L_4 + K_4(L_1, L_2, L_3) = 0$$

where each K_i is a homogeneous polynomial of degree i in L_1, L_2, and L_3. This form, quadratic in L_4, allows us to replace the L_4-coordinate with a single bit indicating the "sign" in the corresponding root of this quadratic; the remaining three coordinates can be normalized to an affine plane point. The net result is a compression to two field elements, plus one bit indicating the normalization, plus another bit to indicate the correct value of L_4.

Remark 1. Stahlke gives a compression algorithm in [42] for points on genus-2 Jacobians in the usual Mumford representation. The first step can be seen as a projection to the most general model of the Kummer (as in [12, Chap. 3]), and then the second is an implicit implementation of the principle above.

6.2 From Squared Kummers to Tetragonal Kummers

We want to define an efficient point compression scheme for $\mathcal{K}^{\mathrm{Sqr}}$. The general principle above makes this possible, but it leaves open the choice of node N and the choice of forms L_i. These choices determine the complexity of the resulting K_i, and hence the cost of evaluating them; this in turn has a non-negligible impact on the time and space required to compress and decompress points, as well as the number of new auxiliary constants that must be stored.

In this section we define a choice of L_i reflecting the special symmetry of $\mathcal{K}^{\mathrm{Sqr}}$. A similar procedure for $\mathcal{K}^{\mathrm{Can}}$ appears in more classical language[6] in [26, Sect. 54]. The trick is to distinguish not one node of $\mathcal{K}^{\mathrm{Sqr}}$, but rather the four nodes forming the kernel of the $(2, 2)$-isogeny $\widehat{\mathcal{S}} \circ \widehat{\mathcal{C}} \circ \mathcal{H} : \mathcal{K}^{\mathrm{Sqr}} \to \widehat{\mathcal{K}}^{\mathrm{Sqr}}$, namely

$$\begin{aligned} \pm 0 = N_0 &= (\mu_1 : \mu_2 : \mu_3 : \mu_4), & N_1 &= (\mu_2 : \mu_1 : \mu_4 : \mu_3),\\ N_2 &= (\mu_3 : \mu_4 : \mu_1 : \mu_2), & N_3 &= (\mu_4 : \mu_3 : \mu_2 : \mu_1). \end{aligned}$$

We are going to define a coordinate system where these four nodes become the vertices of a coordinate tetrahedron; then, projection onto any three of the four coordinates will represent a projection away from one of these four nodes. The result will be an isomorphic Kummer $\mathcal{K}^{\mathrm{Tet}}$ whose defining equation is quadratic in *all four* of its variables. This might seem like overkill for point compression—quadratic in just one variable would suffice—but it has the agreeable effect of

[6] The analogous model of $\mathcal{K}^{\mathrm{Can}}$ in [26, Sect. 54] is called "the equation referred to a Rosenhain tetrad", whose defining equation "...may be deduced from the fact that Kummer's surface is the focal surface of the congruence of rays common to a tetrahedral complex and a linear complex." Modern cryptographers will understand why we have chosen to give a little more algebraic detail here.

dramatically reducing the overall complexity of the defining equation, saving time and memory in our compression and decompression algorithms.

The key is the matrix identity

$$\begin{pmatrix} \widehat{\kappa}_4 & \widehat{\kappa}_3 & \widehat{\kappa}_2 & \widehat{\kappa}_1 \\ \widehat{\kappa}_3 & \widehat{\kappa}_4 & \widehat{\kappa}_1 & \widehat{\kappa}_2 \\ \widehat{\kappa}_2 & \widehat{\kappa}_1 & \widehat{\kappa}_4 & \widehat{\kappa}_3 \\ \widehat{\kappa}_1 & \widehat{\kappa}_2 & \widehat{\kappa}_3 & \widehat{\kappa}_4 \end{pmatrix} \begin{pmatrix} \mu_1 & \mu_2 & \mu_3 & \mu_4 \\ \mu_2 & \mu_1 & \mu_4 & \mu_3 \\ \mu_3 & \mu_4 & \mu_1 & \mu_2 \\ \mu_4 & \mu_3 & \mu_2 & \mu_1 \end{pmatrix} = 8\widehat{\mu}_1\widehat{\mu}_2\widehat{\mu}_3\widehat{\mu}_4 \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \tag{21}$$

which tells us that the projective isomorphism $\mathcal{T}\colon \mathbb{P}^3 \to \mathbb{P}^3$ defined by

$$\mathcal{T}\colon \begin{pmatrix} X_1 \\ : X_2 \\ : X_3 \\ : X_4 \end{pmatrix} \mapsto \begin{pmatrix} L_1 \\ : L_2 \\ : L_3 \\ : L_4 \end{pmatrix} = \begin{pmatrix} \widehat{\kappa}_4 X_1 + \widehat{\kappa}_3 X_2 + \widehat{\kappa}_2 X_3 + \widehat{\kappa}_1 X_4 \\ : \widehat{\kappa}_3 X_1 + \widehat{\kappa}_4 X_2 + \widehat{\kappa}_1 X_3 + \widehat{\kappa}_2 X_4 \\ : \widehat{\kappa}_2 X_1 + \widehat{\kappa}_1 X_2 + \widehat{\kappa}_4 X_3 + \widehat{\kappa}_3 X_4 \\ : \widehat{\kappa}_1 X_1 + \widehat{\kappa}_2 X_2 + \widehat{\kappa}_3 X_3 + \widehat{\kappa}_4 X_4 \end{pmatrix}$$

maps the four "kernel" nodes to the corners of a coordinate tetrahedron:

$$\begin{aligned} \mathcal{T}(N_0) &= (0:0:0:1), & \mathcal{T}(N_2) &= (0:1:0:0), \\ \mathcal{T}(N_1) &= (0:0:1:0), & \mathcal{T}(N_3) &= (1:0:0:0). \end{aligned}$$

The image of $\mathcal{K}^{\mathrm{Sqr}}$ under $\mathcal{T}$ is the **tetragonal surface**

$$\mathcal{K}^{\mathrm{Tet}} : 4tL_1L_2L_3L_4 = \begin{aligned} & r_1^2(L_1L_2 + L_3L_4)^2 + r_2^2(L_1L_3 + L_2L_4)^2 + r_3^2(L_1L_4 + L_2L_3)^2 \\ & \quad - 2r_1s_1((L_1^2 + L_2^2)L_3L_4 + L_1L_2(L_3^2 + L_4^2)) \\ & \quad - 2r_2s_2((L_1^2 + L_3^2)L_2L_4 + L_1L_3(L_2^2 + L_4^2)) \\ & \quad - 2r_3s_3((L_1^2 + L_4^2)L_2L_3 + L_1L_4(L_2^2 + L_3^2)) \end{aligned}$$

where $t = 16\mu_1\mu_2\mu_3\mu_4\widehat{\mu}_1\widehat{\mu}_2\widehat{\mu}_3\widehat{\mu}_4$ and

$$\begin{aligned} r_1 &= (\mu_1\mu_3 - \mu_2\mu_4)(\mu_1\mu_4 - \mu_2\mu_3), & s_1 &= (\mu_1\mu_2 - \mu_3\mu_4)(\mu_1\mu_2 + \mu_3\mu_4), \\ r_2 &= (\mu_1\mu_2 - \mu_3\mu_4)(\mu_1\mu_4 - \mu_2\mu_3), & s_2 &= (\mu_1\mu_3 - \mu_2\mu_4)(\mu_1\mu_3 + \mu_2\mu_4), \\ r_3 &= (\mu_1\mu_2 - \mu_3\mu_4)(\mu_1\mu_3 - \mu_2\mu_4), & s_3 &= (\mu_1\mu_4 - \mu_2\mu_3)(\mu_1\mu_4 + \mu_2\mu_3). \end{aligned}$$

As promised, the defining equation of $\mathcal{K}^{\mathrm{Tet}}$ is quadratic in all four of its variables.

For compression we project away from $\mathcal{T}(\pm 0) = (0:0:0:1)$ onto the $(L_1 : L_2 : L_3)$-plane. Rewriting the defining equation as a quadratic in L_4 gives

$$\mathcal{K}^{\mathrm{Tet}} : K_4(L_1, L_2, L_3) - 2K_3(L_1, L_2, L_3)L_4 + K_2(L_1, L_2, L_3)L_4^2 = 0$$

where

$$\begin{aligned} K_2 &:= r_3^2L_1^2 + r_2^2L_2^2 + r_1^2L_3^2 - 2\left(r_3s_3L_2L_3 + r_2s_2L_1L_3 + r_1s_1L_1L_2\right), \\ K_3 &:= r_1s_1(L_1^2 + L_2^2)L_3 + r_2s_2(L_1^2 + L_3^2)L_2 + r_3s_3(L_2^2 + L_3^2)L_1 \\ & \qquad + (2t - (r_1^2 + r_2^2 + r_3^2))L_1L_2L_3, \\ K_4 &:= r_3^2L_2^2L_3^2 + r_2^2L_1^2L_3^2 + r_1^2L_1^2L_2^2 - 2\left(r_3s_3L_1 + r_2s_2L_2 + r_1s_1L_3\right)L_1L_2L_3. \end{aligned}$$

Lemma 1. *If* $(l_1 : l_2 : l_3 : l_4)$ *is a point on* $\mathcal{K}^{\mathrm{Tet}}$*, then*

$$K_2(l_1, l_2, l_3) = K_3(l_1, l_2, l_3) = K_4(l_1, l_2, l_3) = 0 \iff l_1 = l_2 = l_3 = 0\,.$$

Proof. Write k_i for $K_i(l_1, l_2, l_3)$. If $(l_1, l_2, l_3) = 0$ then $(k_2, k_3, k_4) = 0$, because each K_i is nonconstant and homogeneous. Conversely, if $(k_2, k_3, k_4) = 0$ and $(l_1, l_2, l_3) \neq 0$ then we could embed a line in $\mathcal{K}^{\mathrm{Tet}}$ via $\lambda \mapsto (l_1 : l_2 : l_3 : \lambda)$; but this is a contradiction, because $\mathcal{K}^{\mathrm{Tet}}$ contains no lines. □

6.3 Compression and Decompression for $\mathcal{K}^{\mathrm{Sqr}}$

In practice, we compress points on $\mathcal{K}^{\mathrm{Sqr}}$ to tuples (l_1, l_2, τ, σ), where l_1 and l_2 are field elements and τ and σ are bits. The recipe is

(1) Map $(X_1 : X_2 : X_3 : X_4)$ through $\mathcal{T}$ to a point $(L_1 : L_2 : L_3 : L_4)$ on $\mathcal{K}^{\mathrm{Tet}}$.
(2) Compute the unique (l_1, l_2, l_3, l_4) in one of the forms $(*, *, 1, *)$, $(*, 1, 0, *)$, $(1, 0, 0, *)$, or $(0, 0, 0, 1)$ such that $(l_1 : l_2 : l_3 : l_4) = (L_1 : L_2 : L_3 : L_4)$.
(3) Compute $k_2 = K_2(l_1, l_2, l_3)$, $k_3 = K_3(l_1, l_2, l_3)$, and $k_4 = K_4(l_1, l_2, l_3)$.
(4) Define the bit $\sigma = \texttt{Sign}(k_2 l_4 - k_3)$; then (l_1, l_2, l_3, σ) determines l_4. Indeed, $q(l_4) = 0$, where $q(X) = k_2 X^2 - 2k_3 X + k_4$; and Lemma 1 tells us that $q(X)$ is either quadratic, linear, or identically zero.
   - If q is a nonsingular quadratic, then l_4 is determined by (l_1, l_2, l_3) and σ, because $\sigma = \texttt{Sign}(R)$ where R is the correct square root in the quadratic formula $l_4 = (k_3 \pm \sqrt{k_3^2 - k_2 k_4})/k_2$.
   - If q is singular or linear, then (l_1, l_2, l_3) determines l_4, and σ is redundant.
   - If $q = 0$ then $(l_1, l_2, l_3) = (0, 0, 0)$, so $l_4 = 1$; again, σ is redundant.

   Setting $\sigma = \texttt{Sign}(k_2 l_4 - k_3)$ in every case, regardless of whether or not we need it to determine l_4, avoids ambiguity and simplifies code.
(5) The normalization in Step 2 forces $l_3 \in \{0, 1\}$; so encode l_3 as a single bit τ.

The datum (l_1, l_2, τ, σ) completely determines (l_1, l_2, l_3, l_4), and thus determines $(X_1 : X_2 : X_3 : X_4) = \mathcal{T}^{-1}((l_1 : l_2 : l_2 : l_4))$. Conversely, the normalization in Step 2 ensures that (l_1, l_2, τ, σ) is uniquely determined by $(X_1 : X_2 : X_3 : X_4)$, and is independent of the representative values of the X_i.

Algorithm 4 carries out the compression process above; the most expensive step is the computation of an inverse in $\mathbb{F}_p$. Algorithm 5 is the corresponding decompression algorithm; its cost is dominated by computing a square root in $\mathbb{F}_p$.

Proposition 5. *Algorithms 4 and 5 (*`Compress` *and* `Decompress`*) satisfy the following properties: given* (l_1, l_2, τ, σ) *in* $\mathbb{F}_p^2 \times \{0, 1\}^2$, `Decompress` *always returns either a valid point in* $\mathcal{K}^{\mathrm{Sqr}}(\mathbb{F}_p)$ *or* $\perp$*; and for every* $\pm P$ *in* $\mathcal{K}^{\mathrm{Sqr}}(\mathbb{F}_p)$*, we have*

$$\texttt{Decompress}(\texttt{Compress}(\pm P)) = \pm P\,.$$

Proof. In Algorithm 5 we are given (l_1, l_2, τ, σ). We can immediately set $l_3 = \tau$, viewed as an element of $\mathbb{F}_p$. We want to compute an l_4 in $\mathbb{F}_p$, if it exists, such that $k_2 l_4^2 - 2k_3 l_4 + k_4 = 0$ and $\texttt{Sign}(k_2 l_4 - l_3) = \sigma$ where $k_i = K_i(l_1, l_2, l_3)$.

Algorithm 4. Kummer point compression for $\mathcal{K}^{\mathrm{Sqr}}$

```
function Compress
    Input: $\pm P$ in $\mathcal{K}^{\mathrm{Sqr}}(\mathbb{F}_p)$
    Output: $(l_1, l_2, \tau, \sigma)$ with $l_1, l_2 \in \mathbb{F}_p$ and $\sigma, \tau \in \{0, 1\}$
    Cost: $8\mathbf{M} + 5\mathbf{S} + 12\mathbf{C} + 8\mathbf{a} + 5\mathbf{s} + 1\mathbf{I}$
    $\begin{pmatrix} \mathsf{L}_1, \mathsf{L}_2, \\ \mathsf{L}_3, \mathsf{L}_4 \end{pmatrix} \leftarrow \begin{pmatrix} \mathtt{Dot}(\pm P, (\widehat{\kappa}_4, \widehat{\kappa}_3, \widehat{\kappa}_2, \widehat{\kappa}_1)), \mathtt{Dot}(\pm P, (\widehat{\kappa}_3, \widehat{\kappa}_4, \widehat{\kappa}_1, \widehat{\kappa}_2)), \\ \mathtt{Dot}(\pm P, (\widehat{\kappa}_2, \widehat{\kappa}_1, \widehat{\kappa}_4, \widehat{\kappa}_3)), \mathtt{Dot}(\pm P, (\widehat{\kappa}_1, \widehat{\kappa}_2, \widehat{\kappa}_3, \widehat{\kappa}_4)) \end{pmatrix}$
    if $\mathsf{L}_3 \neq 0$ then
        $(\tau, \lambda) \leftarrow (1, \mathsf{L}_3^{-1})$                // Normalize to $(* : * : 1 : *)$
    else if $\mathsf{L}_2 \neq 0$ then
        $(\tau, \lambda) \leftarrow (0, \mathsf{L}_2^{-1})$                // Normalize to $(* : 1 : 0 : *)$
    else if $\mathsf{L}_1 \neq 0$ then
        $(\tau, \lambda) \leftarrow (0, \mathsf{L}_1^{-1})$                // Normalize to $(1 : 0 : 0 : *)$
    else
        $(\tau, \lambda) \leftarrow (0, \mathsf{L}_4^{-1})$                // Normalize to $(0 : 0 : 0 : 1)$
    $(\mathsf{l}_1, \mathsf{l}_2, \mathsf{l}_4) \leftarrow (\mathsf{L}_1 \cdot \lambda, \mathsf{L}_2 \cdot \lambda, \mathsf{L}_4 \cdot \lambda)$   // $(\mathsf{l}_1 : \mathsf{l}_2 : \tau : \mathsf{l}_4) = (\mathsf{L}_1 : \mathsf{L}_2 : \mathsf{L}_3 : \mathsf{L}_4)$
    $(\mathsf{k}_2, \mathsf{k}_3) \leftarrow (K_2(\mathsf{l}_1, \mathsf{l}_2, \tau), K_3(\mathsf{l}_1, \mathsf{l}_2, \tau))$
    $\mathsf{R} \leftarrow \mathsf{k}_2 \cdot \mathsf{l}_4 - \mathsf{k}_3$
    $\sigma \leftarrow \mathtt{Sign}(\mathsf{R})$
    return $(\mathsf{l}_1, \mathsf{l}_2, \tau, \sigma)$
```

If such an l_4 exists, then we will have a preimage $(l_1 : l_2 : l_3 : l_4)$ in $\mathcal{K}^{\mathrm{Tet}}(\mathbb{F}_p)$, and we can return the decompressed $\mathcal{T}^{-1}((l_1 : l_2 : l_3 : l_4))$ in $\mathcal{K}^{\mathrm{Sqr}}$.

If $(k_2, k_3) = (0, 0)$ then $k_4 = 2k_3 l_4 - k_2 l_4^2 = 0$, so $l_1 = l_2 = \tau = 0$ by Lemma 1. The only legitimate datum in this form is is $(l_1 : l_2 : \tau : \sigma) = (0 : 0 : 0 : \mathtt{Sign}(0))$. If this was the input, then the preimage is $(0 : 0 : 0 : 1)$; otherwise we return $\perp$.

If $k_2 = 0$ but $k_3 \neq 0$, then $k_4 = 2k_3 l_4$, so $(l_1 : l_2 : \tau : l_4) = (2k_3 l_1 : 2k_3 l_2 : 2k_3 \tau : k_4)$. The datum is a valid compression unless $\sigma \neq \mathtt{Sign}(-k_3)$, in which case we return $\perp$; otherwise, the preimage is $(2k_3 l_1 : 2k_3 l_2 : 2k_3 \tau : k_4)$.

If $k_2 \neq 0$, then the quadratic formula tells us that any preimage satisfies $k_2 l_4 = k_3 \pm \sqrt{k_3^2 - k_2 k_4}$, with the sign determined by $\mathtt{Sign}(k_2 l_4 - k_3)$. If $k_3^2 - k_2 k_4$ is not a square in $\mathbb{F}_p$ then there is no such l_4 in $\mathbb{F}_p$; the input is illegitimate, so we return $\perp$. Otherwise, we have a preimage $(k_2 l_1 : k_2 l_2 : k_2 l_3 : l_3 \pm \sqrt{k_3^2 - k_2 k_4})$.

Line 17 maps the preimage $(l_1 : l_2 : l_3 : l_4)$ in $\mathcal{K}^{\mathrm{Tet}}(\mathbb{F}_p)$ back to $\mathcal{K}^{\mathrm{Sqr}}(\mathbb{F}_p)$ via $\mathcal{T}^{-1}$, yielding the decompressed point $(X_1 : X_2 : X_3 : X_4)$. □

6.4 Using Cryptographic Parameters

Our compression scheme works out particularly nicely for the Gaudry–Schost Kummer over $\mathbb{F}_{2^{127}-1}$. First, since every field element fits into 127 bits, every compressed point fits into exactly 256 bits. Second, the auxiliary constants are small: we have $(\widehat{\kappa}_1 : \widehat{\kappa}_2 : \widehat{\kappa}_3 : \widehat{\kappa}_4) = (-961 : 128 : 569 : 1097)$, each of which fits

Algorithm 5. Kummer point decompression to $\mathcal{K}^{\mathrm{Sqr}}$

```
function Decompress
    Input: $(l_1, l_2, \tau, \sigma)$ with $l_1, l_2 \in \mathbb{F}_p$ and $\tau, \sigma \in \{0, 1\}$
    Output: The point $\pm P$ in $\mathcal{K}^{\mathrm{Sqr}}(\mathbb{F}_p)$ such that
            $\mathtt{Compress}(\pm P) = (l_1, l_2, \tau, \sigma)$, or $\perp$ if no such $\pm P$ exists
    Cost: $10\mathbf{M} + 9\mathbf{S} + 18\mathbf{C} + 13\mathbf{a} + 8\mathbf{s} + 1\mathbf{E}$
    $(\mathsf{k}_2, \mathsf{k}_3, \mathsf{k}_4) \leftarrow (K_2(l_1, l_2, \tau), K_3(l_1, l_2, \tau), K_4(l_1, l_2, \tau))$
    if $\mathsf{k}_2 = 0$ and $\mathsf{k}_3 = 0$ then
        if $(l_1, l_2, \tau, \sigma) \neq (0, 0, 0, \mathtt{Sign}(0))$ then
            return $\perp$                                   // Invalid compression
        $\mathsf{L} \leftarrow (0, 0, 0, 1)$
    else if $\mathsf{k}_2 = 0$ and $\mathsf{k}_3 \neq 0$ then
        if $\sigma \neq \mathtt{Sign}(-\mathsf{k}_3)$ then
            return $\perp$                                   // Invalid compression
        $\mathsf{L} \leftarrow (2 \cdot l_1 \cdot \mathsf{k}_3, 2 \cdot l_2 \cdot \mathsf{k}_3, 2 \cdot \tau \cdot \mathsf{k}_3, \mathsf{k}_4)$          // $\mathsf{k}_4 = 2\mathsf{k}_3 l_4$
    else
        $\Delta \leftarrow \mathsf{k}_3^2 - \mathsf{k}_2\mathsf{k}_4$
        $\mathsf{R} \leftarrow \mathtt{HasSquareRoot}(\Delta, \sigma)$   // $\mathsf{R} = \perp$ or $\mathsf{R}^2 = \Delta$, $\mathtt{Sign}(\mathsf{R}) = \sigma$
        if $\mathsf{R} = \perp$ then
            return $\perp$                              // No preimage in $\mathcal{K}^{\mathrm{Tet}}(\mathbb{F}_p)$
        $\mathsf{L} \leftarrow (\mathsf{k}_2 \cdot l_1, \mathsf{k}_2 \cdot l_2, \mathsf{k}_2 \cdot \tau, \mathsf{k}_3 + \mathsf{R})$              // $\mathsf{k}_3 + \mathsf{R} = \mathsf{k}_2 l_4$
    $\begin{pmatrix} X_1, X_2, \\ X_3, X_4 \end{pmatrix} \leftarrow \begin{pmatrix} \mathtt{Dot}(\mathsf{L}, (\mu_4, \mu_3, \mu_2, \mu_1)), \mathtt{Dot}(\mathsf{L}, (\mu_3, \mu_4, \mu_1, \mu_2)), \\ \mathtt{Dot}(\mathsf{L}, (\mu_2, \mu_1, \mu_4, \mu_3)), \mathtt{Dot}(\mathsf{L}, (\mu_1, \mu_2, \mu_3, \mu_4)) \end{pmatrix}$
    return $(X_1 : X_2 : X_3 : X_4)$
```

into well under 16 bits. Computing the polynomials K_2, K_3, K_4 and dividing them all through by 11^2 (which does not change the roots of the quadratic) gives

$$K_2(l_1, l_2, \tau) = (q_5 l_1)^2 + (q_3 l_2)^2 + (q_4 \tau)^2 - 2q_3\big(q_2 l_1 l_2 + \tau(q_0 l_1 - q_1 l_2)\big), \tag{22}$$

$$K_3(l_1, l_2, \tau) = q_3\big(q_0(l_1^2 + \tau)l_2 - q_1 l_1(l_2^2 + \tau) + q_2(l_1^2 + l_2^2)\tau\big) - q_6 q_7 l_1 l_2 \tau, \tag{23}$$

$$K_4(l_1, l_2, \tau) = ((q_3 l_1)^2 + (q_5 l_2)^2 - 2q_3 l_1 l_2\big(q_0 l_2 - q_1 l_1 + q_2\big))\tau + (q_4 l_1 l_2)^2, \tag{24}$$

where $(q_0, \ldots, q_7) = (3575, 9625, 4625, 12259, 11275, 7475, 6009, 43991)$; each of the q_i fits into 16 bits. In total, the twelve new constants we need for `Compress` and `Decompress` together fit into less than two field elements' worth of space.

7 Implementation

In this section we present the results of the implementation of the scheme on the AVR ATmega and ARM Cortex M0 platforms. We have a total of four implementations: on both platforms we implemented both the Curve25519-based scheme and the scheme based on a fast Kummer surface in genus 2. The benchmarks for the AVR software are obtained from the Arduino MEGA development board containing an ATmega2560 MCU, compiled with GCC v4.8.1. For the Cortex M0, they are measured on the STM32F051R8 MCU on the STMF0Discovery board, compiled with Clang v3.5.0. We refer to the (publicly available) code for more detailed compiler settings. For both Diffie–Hellman and signatures we follow the eBACS [4] API.

7.1 Core Functionality

The arithmetic of the underlying finite fields is well-studied and optimized, and we do not reinvent the wheel. For field arithmetic in $\mathbb{F}_{2^{255}-19}$ we use the highly optimized functions presented by Hutter and Schwabe [27] for the AVR ATmega, and the code from Düll et al. [17] for the Cortex M0. For arithmetic in $\mathbb{F}_{2^{127}-1}$ we use the functions from Renes et al. [39], which in turn rely on [27] for the AVR ATmega, and on [17] for the Cortex M0.

The `SHAKE128` functions for the ATmega are taken from [10], while on the Cortex M0 we use a modified version from [2]. Cycle counts for the main functions defined in the rest of this paper are presented in Table 2. Notably, the `Ladder` routine is by far the most expensive function. In genus 1 the `Compress` function is relatively costly (it is essentially an inversion), while in genus 2 `Check`, `Compress` and `Decompress` have only minor impact on the total run-time. More interestingly, as seen in Tables 3 and 4, the simplicity of operating only on the Kummer variety allows smaller code and less stack usage.

Table 2. Cycle counts for the four key functions of `qDSA` at the 128-bit security level.

Genus	Function	Ref.	AVR ATmega	ARM Cortex M0
1	`Ladder`	–	12 539 098	3 338 554
	`Check`	Algorithm 2	46 546	17 044
	`Compress`	Sect. 3.1	1 067 004	270 867
	`Decompress`	Sect. 3.1	694	102
2	`Ladder`	–	9 624 637	2 683 371
	`Check`[a]	Algorithm 3	84 424	24 249
	`Compress`	Algorithm 4	212 374	62 165
	`Decompress`	Algorithm 5	211 428	62 471

[a] The implementation decompresses $\pm R$ within `Check`, while Algorithm 3 assumes $\pm R$ to be decompressed. We have subtracted the cost of the `Decompress` function once.

7.2 Comparison to Previous Work

There are not many implementations of complete signature and key exchange schemes on microcontrollers. On the other hand, there are implementations of scalar multiplication on elliptic curves. The current fastest on our platforms are presented by Düll et al. [17], and since we are relying on exactly the same arithmetic, we have essentially the same results. Similarly, the current records for scalar multiplication on Kummer surfaces are presented by Renes et al. [39]. Since we use the same underlying functions, we have similar results.

More interestingly, we compare the speed and memory usage of signing and verification to best known results of implementations of complete signature schemes. To the best of our knowledge, the only other works are the Ed25519-based scheme by Nascimento et al. [34], the Four$\mathbb{Q}$-based scheme (obtaining fast scalar multiplication by relying on easily computable endomorphisms) by Liu et al. [30], and the genus-2 implementation from [39].

AVR ATmega. As we see in Table 3, our implementation of the scheme based on Curve25519 outperforms the Ed25519-based scheme from [34] in every way. It reduces the number of clock cycles needed for `sign` resp. `verify` by more than 26% resp. 17%, while reducing stack usage by more than 65% resp. 47%. Code size is not reported in [34]. Comparing against the Four$\mathbb{Q}$ implementation of [30], we see a clear trade-off between speed and size: Four$\mathbb{Q}$ has a clear speed advantage, but `qDSA` on Curve25519 requires only a fraction of the stack space.

The implementation based on the Kummer surface of the genus-2 Gaudry–Schost Jacobian does better than the Curve25519-based implementation across the board. Compared to [39], the stack usage of `sign` resp. `verify` decreases by more than 54% resp. 38%, while decreasing code size by about 11%. On the other

Table 3. Performance comparison of the `qDSA` signature scheme against the current best implementations, on the AVR ATmega platform.

Ref.	Object	Function	Clock cycles	Stack	Code size[a]
[34]	Ed25519	`sign`	19 047 706	1 473 bytes	–
		`verify`	30 776 942	1 226 bytes	
[30]	Four$\mathbb{Q}$	`sign`	5 174 800	1 572 bytes	25 354 bytes
		`verify`	11 003 800	4 957 bytes	33 372 bytes
This work	Curve25519	`sign`	14 067 995	512 bytes	21 347 bytes
		`verify`	25 355 140	644 bytes	
[39]	Gaudry–Schost $\mathcal{J}$	`sign`	10 404 033	926 bytes	20 242 bytes
		`verify`	16 240 510	992 bytes	
This work	Gaudry–Schost $\mathcal{K}$	`sign`	10 477 347	417 bytes	17 880 bytes
		`verify`	20 423 937	609 bytes	

[a] All reported code sizes except those from [30, Table 6] include support for both signatures and key exchange

Table 4. Performance comparison of the `qDSA` signature scheme against the current best implementations, on the ARM Cortex M0 platform.

Ref.	Object	Function	Clock cycles	Stack	Code size[a]
This work	Curve25519	`sign`	3 889 116	660 bytes	18 443 bytes
		`verify`	6 793 695	788 bytes	
[39]	Gaudry–Schost $\mathcal{J}$	`sign`	2 865 351	1 360 bytes	19 606 bytes
		`verify`	4 453 978	1 432 bytes	
This work	Gaudry–Schost $\mathcal{K}$	`sign`	2 908 215	580 bytes	18 064 bytes
		`verify`	5 694 414	808 bytes	

[a] In this work 8 448 bytes come from the `SHAKE128` implementation, while [39] uses 6 938 bytes. One could probably reduce this significantly by optimizing the implementation, or by using a more memory-friendly hash function

hand, verification is about 26% slower. This is explained by the fact that in [39] the signature is compressed to 48 bytes (following Schnorr's suggestion), which means that one of the scalar multiplications in verification is only half length. Comparing to the Four$\mathbb{Q}$ implementation of [30], again we see a clear trade-off between speed and size, but this time the loss of speed is less pronounced than in the comparison with Curve25519-based `qDSA`.

ARM Cortex M0. In this case there is no elliptic-curve-based signature scheme to compare to, so we present the first. As we see in Table 4, it is significantly slower than its genus-2 counterpart in this paper (as should be expected), while using a similar amount of stack and code.

The genus-2 signature scheme has similar trade-offs on this platform when compared to the implementation by Renes et al. [39]. The stack usage for `sign` resp. `verify` is reduced by about 57% resp. 43%, while code size is reduced by about 8%. For the same reasons as above, verification is about 28% slower.

Acknowledgements. We thank Peter Schwabe for his helpful contributions to discussions during the creation of this paper.

References

1. Accredited Standards Committee X9: American National Standard X9.62-1999, Public key cryptography for the financial services industry: the elliptic curve digital signature algorithm (ECDSA). Technical report. ANSI (1999)
2. Alkim, E., Jakubeit, P., Schwabe, P.: NewHope on ARM cortex-M. In: Carlet, C., Hasan, M.A., Saraswat, V. (eds.) SPACE 2016. LNCS, vol. 10076, pp. 332–349. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49445-6_19
3. Baily Jr., W.L.: On the theory of θ-functions, the moduli of abelian varieties, and the moduli of curves. Ann. Math. **2**(75), 342–381 (1962)
4. Bernstein, D.J., Lange, T.: eBACS: ECRYPT Benchmarking of Cryptographic Systems. https://bench.cr.yp.to/index.html

5. Bernstein, D.J.: Curve25519: new Diffie-Hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 207–228. Springer, Heidelberg (2006). https://doi.org/10.1007/11745853_14
6. Bernstein, D.J.: Elliptic vs. hyperelliptic, part 1 (2006)
7. Bernstein, D.J., Chuengsatiansup, C., Lange, T., Schwabe, P.: Kummer strikes back: new DH speed records. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8873, pp. 317–337. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45611-8_17
8. Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.-Y.: High-speed high-security signatures. J. Cryptogr. Eng. **2**(2), 77–89 (2012)
9. Bernstein, D.J., Lange, T., Schwabe, P.: The security impact of a new cryptographic library. In: Hevia, A., Neven, G. (eds.) LATINCRYPT 2012. LNCS, vol. 7533, pp. 159–176. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33481-8_9
10. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: The KECCAK sponge function family (2016)
11. Bos, J.W., Costello, C., Hisil, H., Lauter, K.E.: Fast cryptography in genus 2. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 194–210. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38348-9_12
12. Cassels, J.W.S., Flynn, E.V.: Prolegomena to a Middlebrow Arithmetic of Curves of Genus 2, vol. 230. Cambridge University Press, Cambridge (1996)
13. Chudnovsky, D.V., Chudnovsky, G.V.: Sequences of numbers generated by addition in formal groups and new primality and factorization tests. Adv. Appl. Math. **7**, 385–434 (1986)
14. Chung, P.-N., Costello, C., 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)
15. Cosset, R.: Applications des fonctions theta à la cryptographie sur les courbes hyperelliptiques. Ph.D. thesis, Université Henri Poincaré - Nancy I (2011)
16. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Trans. Inf. Theor. **22**(6), 644–654 (1976)
17. Düll, M., Haase, B., Hinterwälder, G., Hutter, M., Paar, C., Sánchez, A.H., Schwabe, P.: High-speed curve25519 on 8-bit, 16-bit and 32-bit microcontrollers. Des. Codes Cryptogr. **77**(2), 493–514 (2015)
18. Dworkin, M.J.: SHA-3 standard: Permutation-based hash and extendable-output functions. Technical report. National Institute of Standards and Technology (NIST) (2015)
19. Fiat, A., Shamir, A.: How to prove yourself: practical solutions to identification and signature problems. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 186–194. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-47721-7_12
20. Gamal, T.E.: A public key cryptosystem and a signature scheme based on discrete logarithms. In: Blakley, G.R., Chaum, D. (eds.) CRYPTO 1984. LNCS, vol. 196, pp. 10–18. Springer, Heidelberg (1985). https://doi.org/10.1007/3-540-39568-7_2
21. Gaudry, P.: Fast genus 2 arithmetic based on theta functions. J. Math. Cryptol. **1**(3), 243–265 (2007)
22. Gaudry, P., Schost, E.: Genus 2 point counting over prime fields. J. Symb. Comput. **47**(4), 368–400 (2012)
23. Hamburg, M.: Fast and compact elliptic-curve cryptography. Cryptology ePrint Archive, Report 2012/309 (2012)
24. Hamburg, M.: The STROBE protocol framework. Cryptology ePrint Archive, Report 2017/003 (2017)

25. Hazay, C., Lindell, Y.: Efficient Secure Two-Party Protocols. Springer, Heidelberg (2010)
26. Hudson, R.W.H.T.: Kummer's Quartic Surface. Cambridge University Press, Cambridge (1905)
27. Hutter, M., Schwabe, P.: NaCl on 8-Bit AVR microcontrollers. In: Youssef, A., Nitaj, A., Hassanien, A.E. (eds.) AFRICACRYPT 2013. LNCS, vol. 7918, pp. 156–172. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38553-7_9
28. Karati, S., Das, A.: Faster batch verification of standard ECDSA signatures using summation polynomials. In: Boureanu, I., Owesarski, P., Vaudenay, S. (eds.) ACNS 2014. LNCS, vol. 8479, pp. 438–456. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07536-5_26
29. Koblitz, N.: Elliptic curve cryptosystems. Math. Comput. **48**, 203–209 (1987)
30. Liu, Z., Longa, P., Pereira, G., Reparaz, O., Seo, H.: Four$\mathbb{Q}$ on embedded devices with strong countermeasures against side-channel attacks. Cryptology ePrint Archive, Report 2017/434 (2017)
31. Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (1986). https://doi.org/10.1007/3-540-39799-X_31
32. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Math. Comput. **48**, 243–264 (1987)
33. Naccache, D., M'Raïhi, D., Vaudenay, S., Raphaeli, D.: Can D.S.A. be improved? — complexity trade-offs with the digital signature standard —. In: De Santis, A. (ed.) EUROCRYPT 1994. LNCS, vol. 950, pp. 77–85. Springer, Heidelberg (1995). https://doi.org/10.1007/BFb0053426
34. 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.) SPACE 2015. LNCS, vol. 9354, pp. 289–309. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24126-5_17
35. 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.) CHES 2001. LNCS, vol. 2162, pp. 126–141. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44709-1_12
36. Perrin, T.: The XEdDSA and VXEdDSA Signature Schemes. https://whispersystems.org/docs/specifications/xeddsa/
37. Pointcheval, D., Stern, J.: Security proofs for signature schemes. In: Maurer, U. (ed.) EUROCRYPT 1996. LNCS, vol. 1070, pp. 387–398. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-68339-9_33
38. Pointcheval, D., Stern, J.: Security arguments for digital signatures and blind signatures. J. Cryptol. **13**(3), 361–396 (2000)
39. Renes, J., Schwabe, P., Smith, B., Batina, L.: μKummer: efficient hyperelliptic signatures and key exchange on microcontrollers. In: Gierlichs, B., Poschmann, A.Y. (eds.) CHES 2016. LNCS, vol. 9813, pp. 301–320. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53140-2_15
40. Schnorr, C.P.: Efficient identification and signatures for smart cards. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 239–252. Springer, New York (1990). https://doi.org/10.1007/0-387-34805-0_22
41. Semaev, I.A.: Summation polynomials and the discrete logarithm problem on elliptic curves. IACR Cryptology ePrint Archive **2004**, 31 (2004)
42. Stahlke, C.: Point compression on jacobians of hyperelliptic curves over $\mathbb{F}_q$. Cryptology ePrint Archive, Report 2004/030 (2004)