Fast and compact elliptic-curve cryptography

Mike Hamburg*

Abstract

Elliptic curve cryptosystems have improved greatly in speed over the past few years. Here we outline a new elliptic curve signature and key agreement implementation which achieves record speeds while remaining relatively compact. For example, on Intel Sandy Bridge, a curve with about 2^{250} points produces a signature in just under 52k clock cycles, verifies in under 170k clock cycles, and computes a Diffie-Hellman shared secret in under 153k clock cycles. Our implementation has a small footprint: the library is under 60kB.

Our implementation is also fast on ARM processors, verifying a signature in under 625k Tegra-2 cycles.

We introduce faster field arithmetic, a new point compression algorithm, an improved fixed-base scalar multiplication algorithm and a new way to verify signatures without inversions or coordinate recovery. Some of these improvements should be applicable to other systems.

1 Introduction

Of the many applications for digital signatures, most use some variant of the venerable RSA scheme. RSA signatures have several advantages, such as age, simplicity and extreme speed of verification. For

*Cryptography Research, a division of Rambus.

example, NIST's recommendations [2] hold RSA-2048 and the elliptic curve signature scheme ECDSA-p_{224} to be similarly secure[1], and the former verifies signatures more than $13\times$ faster than the latter [7].

Elliptic curve signatures have many advantages, however. Current attacks against elliptic curves scale exponentially with key size. Therefore ECC key and signatures can be considerably smaller than their RSA counterparts, and key generation and signing are much faster. Still, elliptic curve signatures' historically slow verification has kept these signatures out of protocols such as DNSSEC (though a draft is in progress [17]), and still makes them a difficult choice for less powerful systems such as embedded devices and smart phones.

This performance gap has narrowed in recent years. Edwards' elliptic curves [9] and their twists provide faster operations [5, 16, 15] than the traditional projective or Jacobian coordinates for Weierstrass curves, and make it easier to achieve resistance to side-channel attacks. Bernstein et. al.'s Ed25519 software [6], which uses these curves, verifies signatures in some 234k cyles on Intel's Sandy Bridge microarchitecture. While RSA-2048 is still considerably faster at approximately 98k (even in the conservative OpenSSL implementation), this is a significant improvement over ECDSA [7].

Here we continue to push the envelope. With several improvements, most importantly faster finite field arithmetic, we achieve approximately 170k median Sandy Bridge cycles for verification, 52k for signing and 55k for key generation. These benchmarks were measured with SUPERCOP [7] and adjusted for Turbo Boost, so they should be reproducible.

Our code is not optimized purely for speed. Rather, it is designed to balance speed with security, simplicity, portability and small cache footprint. For example, we decided to use only 7.5kiB of tables for key generation and signing (compare to 30kiB in [6]). We believe the

[1]Eurocrypt's report [1] estimates ECDSA-p_{224} to be slightly stronger than RSA-2048.

above speeds are fast enough that additional cache pollution would not be worth the modest speed increase in most applications. We also provide standard protection against timing and cacheing attacks, by avoiding the use of secret data in conditional branches and array indices. Since this is a software implementation, we made no attempt to protect against physical side-channels such as DPA.

Fast Sandy Bridge benchmarks are somewhat unsatisfying. We set new records, but the reduction in verification times from 73µs to 53µs will be irrelevant most of the time. However, asymmetric cryptography is somewhat more costly on smartphones, with OpenSSL ECC verifies taking 7.7ms on our 1GHz Tegra 2 (ARM Cortex A9) test machine. On this platform, our implementation also does well. For example, it verifies signatures in about 620k cycles, that is, well under a millisecond. Our ARM code does not currently take advantage of ARM's NEON vector instructions; indeed, our Tegra 2 test machine does not support NEON. Still, our ARM code's performance is similar to Bernstein and Schwabe's NEON results [?]: slightly slower for key agreement, slightly faster for verification, significantly faster for signing.

We also found a new technique for signature verification which avoids the need to decompress points in the signature or public key. Instead of using an integrated linear combination algorithm, we perform two separate scalar multiplications. With fast scalar multiplication algorithms and no point decompression, this approaches the speed of the traditional method. However, with the parameters we chose and Hişil's mixed coordinates from [15], the traditional method is a hair faster than our new technique. Still, our technique may be useful in some situations, so we present it in Appendix A.

2 Overview

The main body of this paper is organized as follows:

- In Section 3.1, we demonstrate a previously unrecognized class of primes which allow for particularly fast arithmetic. The arithmetic in our system is done modulo such a prime.
- In Section 3.2, we describe the Montgomery curve and equivalent twisted Edwards curves which will be used in our system. These curves were chosen to have security properties similar to Curve25519 [3]. Our system uses q-torsion points on these curves.
- In Section 3.3, we describe how q-torsion points are encoded and decoded as field elements, without any extra bits. In order to encode and decode elements efficiently, we use a *simultaneous inversion and square root* algorithm, which to our knowledge is novel.
- In Section 4.1, we describe our signatures, which are modeled on Ed25519 [6] and are similar to Schnorr signatures [19].
- In Section 4.2, we describe the "extensible" coordinates we use for twisted Edwards curves. These are a variant of Hışıl's "mixed homogeneous projective coordinates" [15] with Niels coordinates [6] for readdition. Our coordinates combine many of the strengths of projective coordinates and extended coordinates, without Hışıl's requirement to schedule doublings and additions in advance.
- In Section 4.3, we describe how we sign messages. We introduce a new "multiple signed comb" algorithm which is simpler, more flexible and more efficient than previous algorithms for scalar multiplication with precomputation.
- In Section 4.4, we describe how to verify signatures.

- In Section 5, we describe how we implement elliptic-curve Diffie-Hellman with Montgomery curves, including a technique to recover the v-coordinate from a Montgomery ladder.
- In Section 6, we show benchmarks on Intel Sandy Bridge and Nvidia Tegra-2 processors. These benchmarks were taken with SUPERCOP [7], so they should be reproducible.
- In Appendix A, we describe our alternative technique for signature verification. This approach can be used to verify signatures (up to sign) without decompressing any of the points involved.

3 Design parameters

We made a number of choices in our system's design in order to simultaneously achieve high speed and high security.

3.1 Special Montgomery fields

Most implementations of arithmetic in general prime-order fields use Montgomery reduction, because it is usually faster than other options such as Barrett reduction. When choosing special prime-order fields, we would like to use Mersenne primes, but there are no such primes between $2^{127}-1$ and $2^{521}-1$. Therefore, special fields of other forms are used. These fields usually use Barrett reduction, of which Solinas reduction [20] is a special case.

However, it is possible — and profitable — to use Montgomery reduction in special prime-order fields. For example, let $p = k \cdot 2^{aw} - 1$, where w is the machine's word size, $a \geq 1$, and k is an arbitrary constant. We call such a p a *special Montgomery number*, or a *special Montgomery prime* if it is prime. To perform a Montgomery reduction step $x \to x/2^w$, we use the identity

$$1/2^w \equiv k \cdot 2^{(a-1)w} \pmod{p}$$

That is, to divide by 2^w, shift each word down (virtually) by one position. The bottom word cannot be shifted down; instead, shift it up $a-1$ words, multiply it by k, and accumulate. If k is a single word, then this process can be a single multiply-and-accumulate instruction; if k is low-weight, then this process can be accomplished with a few shifts and adds, similar to Solinas reduction. Unlike Barrett reduction, no contortions are required to avoid carry propagation with this reduction method, because the reduction accumulates into the top words.

For our implementation, we wanted a field with slightly fewer than 2^{256} elements. We chose $p := 2^{252} - 2^{232} - 1$, a special Montgomery prime with a single-word k on both 32- and 64-bit architectures. The gap between p and 2^{256} enables lazy reduction: the result of a multiply is under $2p$, which can be added or subtracted once without reduction; the results can be Montgomery multiplied to produce results under $16p^2/2^{256} + p < 2p$. In cases where more additions or subtractions are needed, Barrett reduction is still efficient. Alternatively, when projective coordinates are used, we can insert an extra Montgomery reduction step – a one-word multiply and accumulate – in two balancing places. This option is applicable mainly in twisted Edwards doubling formulas, where it maximizes speed and minimizes code size.

We considered several other primes for our implementation, but ultimately decided on this 252-bit p. Our analysis of the alternatives is found in Appendix B.

3.2 Curve choice

todo: curve subject to change; $A' = 2803/2^{32}$**?** We chose a Montgomery curve

$$\mathcal{E}_m : v^2 = u^3 + Au^2 + u \pmod{p}$$

The laddering operation on such curves requires a multiplication by

$A' = (A-2)/4$, so we set $A' := 1107/2^{64}$ in order to use Montgomery multiplication. This gives a curve of order

$$4q = p - k \quad \text{where} \quad k = 48305947151610022181991269137732172107$$

with twist of order

$$4\hat{q} = p + k + 2$$

Here q and $\hat{q}$ are both primes, and are both slightly smaller than 2^{250}. Thus both $\mathcal{E}_m$ and its quadratic twist have cofactor 4. This is important for Curve25519-style protocols, where parties need not check that a point is on $\mathcal{E}_m$ before operating on it [3]. For our designs, we only use the order-q subgroup of $\mathcal{E}_m$.

We have checked that neither $\mathcal{E}_m$ nor its quadratic twist have any low-degree complex endomorphisms, and that the orders of p modulo q and $\hat{q}$ are large — in fact, p is a generator modulo both.

The curve $\mathcal{E}_m$ is isomorphic to the twisted Edwards curve

$$\mathcal{E} : y^2 - x^2 = 1 + dx^2y^2 \quad \text{where} \quad d = -\frac{A-2}{A+2} = -\frac{1107}{1107 + 2^{64}}$$

We work only in the order-q subgroups of these curves. Because Edwards curves are faster for most operations, we spent most of our effort on the q-torsion group of $\mathcal{E}$.

3.3 Point compression

We compress q-torsion points in $\mathcal{E}$ and $\mathcal{E}_m$ down to a single element of $\mathbb{F}$. That is, we do not send a separate sign bit. Saving a bit is somewhat gratuitous with our 252-bit prime, but it would be more relevant to a system whose elements have sizes that are a multiple of a byte. In order to do this, we make use of the following lemma:

Lemma. *Let* $P_1 = (u_1, v_1), P_2 = (u_2, v_2), P_3 = (u_3, v_3)$ *be finite points on a curve defined by* $v^2 = u^3 + Au^2 + Bu$*, with* $P_1 = P_2 + P_3$*. Then* $u_1u_2u_3$ *is a quadratic residue in* $\mathcal{F}$*.*

Proof. The points $-P_1, P_2$ and P_3 lie on a line $v = mu + b$, so that $u^3 + Au^2 + Bu - (mu + b)^2 = 0$ at u_1, u_2 and u_3. Thus

$$u^3 + Au^2 + Bu - (mu + b)^2 = (u - u_1) \cdot (u - u_2) \cdot (u - u_3)$$

and $u_1 \cdot u_2 \cdot u_3 = b^2$ is a quadratic residue. □

Corollary. *If $P = (u, v) = 2Q$ on such a curve, then u is a quadratic residue.*

In light of this lemma, we choose to represent points of $\mathcal{E}_m$ as $1/\sqrt{u}$ with the same sign as v, encoded as 32 bytes in little-endian order. Call this encoding $\underline{\mathrm{P}}$. For "sign" we choose the Legendre symbol, but another choice such as the least-significant bit would also suffice. With this representation, the identity is encoded as 0. The point $(0, 0) \in \mathcal{E}_m$ is not a q-torsion point, so we need not worry about u being 0. This form of point compression is also compatible with a fast Montgomery ladder.

To compress a point $(x/z, y/z) \in \mathcal{E}$, we need to compute $\tau := \sqrt{(z - y)/(z + y)}$ with the same sign as xz. To decompress a point encoded as τ, we need to compute

$$y = \frac{1 + \tau^2}{1 - \tau^2} \quad \text{and} \quad x = \tau \sqrt{\frac{-A - 2}{\tau^4 + A\tau^2 + 1}}$$

In order to perform these computations efficiently, we use a *simultaneous inversion and square root* algorithm, similar to Montgomery's simultaneous inversion trick or batch RSA [11]. Since $p \equiv 3 \pmod 4$, for $a \in \mathbb{F}(p)^*$ we might compute $1/a = a^{p-2}$ and $\sqrt{\pm a} = a^{(p+1)/4}$. We can just as easily compute the intermediate $1/\sqrt{\pm a} = a^{(p-3)/4}$[2]. The inverses and square roots in decompression have the form $1/a$ and $\sqrt{1/c}$, which must be computed with Legendre symbol 1. We can compute these by first setting $s = 1/\sqrt{a^2 c}$, whence $1/a = s^2 ac = sa \cdot s \cdot c$

[2]This takes slightly less than an inversion and slightly more than a square root, since $(p + 1)/4$ is highly even.

and $\sqrt{1/c} = sa$. Similarly, for decompression, we must compute $\sqrt{a/b}$ with the same sign as c; we can compute this as $a^2c/\sqrt{a^3bc^2}$. For any of these formulas, if one of the inputs is zero it will ruin the computation; we can avoid this problem by replacing zeros with ones in constant time.

Note that formulas like these can compute any number of inversions and any number of relatively prime roots or symbols simultaneously. In essence, they compute a suitable generator of the lattice containing the logarithms of the desired results, quotiented by the lattice of the logarithms of the inputs. But when the roots to be taken are not relatively prime, this quotient lattice has dimension higher than 1, so more than one expensive root operation will be required. In particular, these formulas do not reduce the time required to take two separate square roots.

4 Signatures

We follow the form of Schnorr signatures [19] used in Ed25519 [6]. Our base point P is the q-torsion point with the numerically minimum encoding, the byte sequence $[4, 0, 0, \ldots, 0]$. A secret key is an integer $a \in [0, q)$, with corresponding public key $Q = a \cdot P$.

4.1 Signature form

To sign a message, the signer chooses a pseudorandom $r \xleftarrow{\text{R}} \mathbb{Z}/q\mathbb{Z}$ and computes $R \leftarrow r \cdot P$. It computes a hash $c \leftarrow H(\underline{\text{Q}}, \underline{\text{R}}, m)$ — we use SHA-256 for H — and sets

$$t \leftarrow r + c \cdot a \pmod q$$

The signature is then $(\underline{\text{R}}, t)$. The verifier can likewise compute c and then check that R is on the curve and that

$$t \cdot P \stackrel{?}{=} c \cdot Q \; + \; R$$

We could simplify our implementation by not checking the sign of R, i.e. verifying this equation for $\pm R$ instead of R. This would save some 3% in verification and a few kiB of code size, and probably would not impact security at all. Additionally, we could choose a hash with a 128-bit output [4, 19]. But in order to remain conservative, and to enable apples-to-apples comparisons with other systems, we chose to use a 256-bit output and to verify the x-coordinate.

The signer may choose r at random, but following [6], we instead choose $r = H(a, m)$ except when a streaming API is desired.

4.2 Coordinate choice

Our signature implementation internally uses points on the twisted Edwards curve $\mathcal{E}$, so it is important to choose efficient coordinates for this curve. Projective coordinates (with 3 coefficients) support the fastest doublings [**?**], but extended coordinates [16] (with 4 coefficients) support the fastest additions.

Hışıl proposes "mixed homogeneous projective coordinates" [15]. With this technique, one predicts whether the next operation will be an addition or a doubling, and if so the extra coordinate T must be computed; otherwise, it is omitted. We wished to avoid such predictions. So instead we store (X, Y, Z, T_1, T_2), where $x = X/Z, y = Y/Z$ and $T_1 \cdot T_2 = T = XY/Z$. The last step of a doubling or addition with extended coordinates [16] amounts to computing $T = T_1 \cdot T_2$. We skip that step and leave T_1 and T_2 in memory, saving a multiply. The T-coordinate is required for additions, so we compute it at the cost of a multiply when beginning an addition. Thus, as with mixed homogeneous projective coordinates, our "extensible" coordinates cost 4 squares and 3 multiplies to double, and 8 multiplies for addition (7 for mixed addition). Since in most cases only one point is in this form, we aren't concerned with the 5-element size. Occasionally we may end up computing T twice, but this is rare and therefore insgignificant. Because we use only q-torsion points, the formulas in [16] are strongly unified even though -1 is not a quadratic residue in $\mathbb{F}(p)$.

For mixed readdition, we use a variant of "Niels coordinates"[3], of the form

$$\alpha = (y - x)/2, \beta = (y + x)/2, \gamma = dxy$$

where d is the Edwards curve constant. For unmixed readdition, we add a Z coordinate.

4.3 Key generation and signing

The bulk of the work in key generation and signing is the computation of $e \cdot P$ for (pseudo)-random values e. For this we use a novel *signed multi-comb* algorithm. This is similar to the algorithm of Lim and Lee [18]. It is also similar in spirit to the signed MSB-set comb algorithm of Feng et. al. [10]. However, our algorithm is much simpler, and is faster and more flexible due to the use of multiple combs.

Let $D > \log_2 q$ be enough digits to write the scalar e. We first write the scalar in signed binary form:

$$e \equiv \sum_{i=0}^{D} d_i \cdot 2^i \pmod{q} \quad \text{where} \quad d_i \in \{\pm 1\}$$

To do this, note that

$$\frac{e + 2^D - 1}{2} = \sum_{k=0}^{D} \frac{d_k + 1}{2} \cdot 2^k$$

[3]Named after Niels Duif. So far as I know, these are so-called only in the source code for Ed25519. Our coordinates are exactly half of Ed25519's Niels coordinates, which saves a multiply by 2.

so that $(d_k + 1)/2 \in \{0, 1\}$ is the kth binary digit of $(e + 2^D - 1)/2 \pmod q$. Here we take advantage of the fact that q is odd, so halving is always possible mod q. Next, we divide the digits into disjoint blocks:

$$e \equiv \sum_{j=0}^{n-1} B_j \quad \text{where} \quad B_j := \sum_{i=o_j}^{o_{j+1}-1} d_i \cdot 2^i$$

Here o_j is the (suitably chosen) offset of the jth block, and $o_n := D$ so that all the digits are accounted for. Finally, we divide the digits in each block into combs. The jth combset has t_j teeth and spacing s_j, where $s_j \cdot t_j = o_{j+1} - o_j$. Set

$$B_j = 2^{o_j} \cdot \sum_{k=0}^{s_j-1} 2^k \cdot C_{j,k} \quad \text{where} \quad C_{j,k} := \sum_{i=0}^{t_j-1} d_{o_j+s_j i+k} \cdot 2^{s_j i}$$

We then have

$$e \quad \equiv \quad \sum_{k=0}^{\max s_j-1} 2^k \cdot \sum_{\substack{j=0 \\ s_j>k}}^{n-1} C_{j,k} \quad = \quad \sum_{k=0}^{\max s_j-1} 2^k \cdot \sum_{\substack{j=0 \\ s_j>k}}^{n-1} \pm|C_{j,k}| \pmod q$$

Our algorithm follows directly. Given a base point P, precompute the $2^{t_j} - 1$ values of $|C_{j,k}| \cdot P$ for each $j \in [0, n)$. To perform a multiplication, compute the signed digits d_i of e, then evaluate

$$e \cdot P \quad = \quad \sum_{k=0}^{\max s_j-1} 2^k \cdot \sum_{\substack{j=0 \\ s_j>k}}^{n-1} \pm|C_{j,k}| \cdot P$$

with $\left(\sum_{j=0}^{n-1} s_j \cdot t_j\right) - 1 = D - 1$ additions (from the inner sum) and $\max s_j - 1$ doublings (from the outer sum).

In a conventional point multiplication algorithm, point additions can be skipped when the coefficient happens to be zero. A similar optimization is possible with our comb algorithm. When $C_{j,k} = -C_{j,k+1}$, then the former can be inverted and the latter skipped. This process replaces a higher and a lower coefficient with only the lower one, and so should be performed from the top down; in this case, it should save $n(s-1)/2^t$ point additions in expectation. We could divide the bits into combs differently, so that they interlock; in this case, the above optimization saves $(ns-1)/2^t$ additions in expectation. We do not perform this optimization, because it would introduce a timing or SPA attack. We also do not interlock the combs, because gathering the digits from the entirety of e is slightly more complicated and expensive.

To prevent timing attacks, we fetch points from memory using a linear pass over the table with arithmetic operations. With Sandy Bridge's vector unit[4], this costs about 2.5 cycles per coefficient loaded on our test platform, or about 4-5% of a multiply. This penalty discourages large tables, and also encourages to use representations with fewer coordinates.

We can use either affine or Niels coordinates for these tables. An extensible+Niels addition costs $7M$, and an extensible+affine addition costs $8M$. Additionally, the Niels addition is strongly unified, but a strongly unified affine addition costs another multiply by d. We do not consider this in Table 4.3, as most applications of fixed-base scalar multiplication do not require unified addition (for example, key generation and signing do not require it). Affine coordinates have two elements of $\mathbb{F}(p)$, and Niels coordinates have three; in many cases, a 50% bigger table is faster than the Niels addition formulas.

With $\lceil \log_2 q \rceil = 250$, we can set D to either 250 or 252 and use the same t for each t_j and the same s for each s_j. This works well because such D divide evenly into appropriately-sized $n \cdot t \cdot s$. Alternatively, we can set $D = 250$, $n = 4$ and $s_0 = s_1 = 12$ but $s_2 = s_3 = 13$, or some other combination. We used a script to predict the cost for different choices of n, t and s, with either affine or Niels coordinates. Some

[4]SSE2 and AVX seem to perform similarly here, because the SSE2 pipeline issues more instructions per cycle. Our Tegra-2 test machine does not have a vector unit.

			Affine			Niels		
t	n	s	size	const	var	size	const	var
5	2	25	2.0	2.45	2.11	3.0	2.40	1.92
5	3	17	3.0	2.29	1.95	4.5	2.24	1.75
5	4	12,13	4.0	2.16	1.82	6.0	2.11	1.62
5	5	10	5.0	2.08	1.74	7.5	2.03	1.55
6	3	14	6.0	2.12	1.61	9.0	2.20	1.45
5	10	5	10.0	1.96	1.62	15.0	1.91	1.43
6	6	7	12.0	1.95	1.44	18.0	2.03	1.28
6	7	6	14.0	1.92	1.42	21.0	2.01	1.25
7	4	9	16.0	2.16	1.31	24.0	2.44	1.17

Table 1: Precomputation trade-off: space in KiB vs. approximate time in multiplies per bit, 250-bit scalar.

of the results are shown in Table 4.3. Side-channel protection makes large t exponentially costly, so that the benefits top out at $t = 5$ with Niels coordinates and $t = 6$ with affine coordinates. On the other hand, $t = 5$ is generally faster than $t = 4$ even for small tables.

We somewhat arbitrarily chose $(t, n, s) = (5, 5, 10)$ with Niels coordinates. This makes our tables fairly compact, yet fast for operations with and without side channel protection. We would need to nearly double the table size in order to achieve a 5% speedup.

We can generate the tables reasonably efficiently by iterating over the points in each comb in Gray-code order. We compute the first point in the comb with $t-1$ additions. Each additional point requires one addition or subtraction. We compute all the combs in projective Niels coordinates, then normalize all of them with one simultaneous inversion. We benchmarked this precomputation at around 217k Sandy Bridge cycles for $(t, n, s) = (5, 5, 10)$, making it worthwhile when more than 3 point multiplications will be done with the same base.

4.4 Verification

For verification, we use a standard WNAF linear-combination algorithm. For the multiples of the fixed base point P, we use a 2^6-element precomputed table in Niels coordinates. This requires 1/9 mixed readdition per bit in expectation. For the multiples of the public key Q, we precompute 2^3 points on the fly in projective Niels coordinates, requiring 1/6 readdition per bit in expectation.

We decompress the public key Q, but not the signature's challenge point R. Instead, we check the signature directly from the encoding of R, using the output of the linear combination algorithm in projective coordinates. This requires a few multiplications, plus a Legendre symbol computation to check the sign.

5 EC Diffie-Hellman key agreement

For Diffie-Hellman key exchange, the input point (the public key) is encoded using $\mathcal{E}_m$, so we use a Montgomery ladder. We use the formulas in Appendix A.3 to recover the sign of the v-coordinate so that we can encode the output point. By using the simultaneous multiplication and square root formulae, this costs only a dozen or so multiplications (<1kcy) more than compressing without the v-coordinate.

The Montgomery ladder is faster than using Edwards coordinates because the input point does not have to be decompressed. When computing multiples of a point which is already decompressed (for example, in a PAKE protocol), it is faster to use a 4-bit signed-binary fixed-window algorithm. We cannot use a WNAF algorithm in this situation because we wish to protect against side-channel attacks.

6 Benchmarks

We benchmarked using SUPERCOP [7] benchmarking suite to make our benchmarks are fair and reproducible. We modified our key generation routines to use `randombytes` for their entropy, while our signature algorithm uses the deterministic option, hashing its secret key and the message to choose its nonce. This is significant because `randombytes` runs very slowly, taking about 10k Sandy Bridge cycles to generate 256 bits of randomness. We used our own (simple and slow) implementation of SHA256 for the hash, but the hashing time is a small fraction of the numbers presented here.

Our first test machine is a laptop with a 2.2GHz Intel Core i7 2720QM Sandy Bridge processor, running on only one of its 4 cores. The processor Turbo Boosts to at most 3.2GHz, while its cycle counter always runs at 2.2GHz. SUPERCOP does not notice this, so we multiplied our cycle counts accordingly.

Our second test machine is a TrimSlice nettop with a dual-core 1GHz Tegra 2 core (Cortex A9, no NEON vector unit), running on only one of its 2 cores. SUPERCOP uses the Linux perf-events system to measure timings.

todo: rigorize? iron out variance? other platforms?

todo: get final numbers

todo: variants?

These benchmarks compare quite favorably with previous systems. Our software takes $20 - 25\%$ less time on Sandy Bridge than Bernstein's $\mathbb{F}(2^{255} - 19)$ software [6, 7]. We did not find any other elliptic curve signing and key exchange algorithms which are optimized for NEON-less ARM processors. Curve25519 and Ed25519's reference code runs on ARM, but is completely unoptimized. We spent little effort on ARM optimization, but we provided intrinsics for `umaal` and an interleaved multiplier which calls them. As a result, our code

	Our System		Ed/Curve25519	
Operation	Sandy Bridge	Tegra 2	Sandy Bridge	Cortex-A8 (NEON)
Generate keys	55kcy	255kcy	73kcy	N/A
Sign	52kcy	256kcy	70kcy	368kcy
Verify	170kcy	624kcy	226kcy	650kcy
Verify without x	165kcy	603kcy	N/A	N/A
Compute shared secret	153kcy	619kcy	194kcy	527kcy

Table 2: Benchmarks. Median cycles. Our results rounded up to the next multiple of 1,000 cycles, previous results rounded down.

takes 55-63% fewer cycles than Curve25519 and Ed25519 for the same operations.

Hüseyin Hışıl's ecfp256 software is harder to compare, since it does not implement signatures. Its fastest key generation is reported as 53kcy on Sandy Bridge with 192kiB tables, no `randombytes` and no side-channel protection [7]. It was designed to test performance of different Edwards curve parameterizations, so it does not use a Montgomery ladder for key agreement; thus our key agreement algorithm takes 38% fewer cycles. ecfp256 does not run on ARM.

7 Conclusions and Future Work

We have presented a new, faster implementation of elliptic curve cryptography. In particular, we have demonstrated faster fields, a new point compression algorithm and a new algorithm for scalar multiplication with precomputation. Our system is faster than previous work on PCs and on smartphones without vector units, and competitive when a vector unit is present.

We have not investigated elliptic curves with endomorphisms, but our form of pseudo-Mersenne prime should be suitable for implementing such curves. We do not have a NEON-accelerated version of our code, and would be interested in developing one. We would also be interested to test other cryptographic algorithms with our toolkit: batch signature verification, oblivious function evaluation, password-authenticated key exchange and so on.

References

[1] Yearly report on algorithms and keysizes (2010-2011), 2011.

[2] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. Recommendation for key management–part 1: General (revision 3). *NIST special publication*, 800:57, 2011.

[3] D. Bernstein. Curve25519: new diffie-hellman speed records. *Public Key Cryptography-PKC 2006*, pages 207–228, 2006.

[4] D. Bernstein. ElGamal vs schnorr vs ECDSA vs..., August 29 2006. Posted to sci.crypt.

[5] D. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters. Twisted edwards curves. *Progress in Cryptology–AFRICACRYPT 2008*, pages 389–405, 2008.

[6] D.J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.Y. Yang. High-speed high-security signatures. *Cryptographic Hardware and Embedded Systems, CHES 2011*, 2011.

[7] D.J. Bernstein and T. Lange. eBACS: Ecrypt benchmarking of cryptographic systems. http://bench.cr.yp.to, accessed 28 October 2011.

[8] P.G. Comba. *Experiments in fast multiplication of large integers.* IBM, 1979.

[9] H.M. Edwards. A normal form for elliptic curves. *Bulletin-American Mathematical Society*, 44(3):393, 2007.

[10] M. Feng, B. Zhu, C. Zhao, and S. Li. Signed msb-set comb method for elliptic curve point multiplication. *Information Security Practice and Experience*, pages 13–24, 2006.

[11] A. Fiat. Batch rsa. *Journal of Cryptology*, 10(2):75–88, 1997.

[12] S. Galbraith, X. Lin, and M. Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. *Advances in Cryptology-EUROCRYPT 2009*, pages 518–535, 2009.

[13] R. Gallant, R. Lambert, and S. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In *Advances in CryptologyCRYPTO 2001*, pages 190–200. Springer, 2001.

[14] D.R. Hankerson, S.A. Vanstone, and A.J. Menezes. *Guide to elliptic curve cryptography*. Springer-Verlag New York Inc, 2004.

[15] H. Hişil. Elliptic curves, group law, and efficient computation. 2010.

[16] H. Hisil, K. Wong, G. Carter, and E. Dawson. Twisted edwards curves revisited. *Advances in Cryptology–ASIACRYPT 2008*, pages 326–343, 2008.

[17] P. Hoffman. Elliptic curve dsa for dnssec. http://tools.ietf.org/html/draft-hoffman-dnssec-ecdsa-04, accessed 14 March 2012.

[18] C. Lim and P. Lee. More flexible exponentiation with precomputation. In *Advances in CryptologyCRYPTO94*, pages 95–107. Springer, 1994.

[19] C. Schnorr. Efficient identification and signatures for smart cards. In *Advances in CryptologyCrypto89 Proceedings*, pages 239–252. Springer, 1990.

[20] J.A. Solinas. Generalized mersenne numbers, 1999.

A Verification without decompression

We experimented with an alternative strategy for signature verification, which we ultimately abandoned in favor of a traditional linear-combination implementation.

A.1 Verification

We wish to verify a signature which is of the form $t \cdot P = c \cdot Q + R$, for some scalars t, c and points P, Q, R, of which P is fixed. However, it suffices to verify a weaker equation, such as $t \cdot P = c \cdot Q \pm R$ or $t \cdot P = \pm c \cdot Q \pm R$. The usual strategy (and the one we ultimately adopted) is to compute $t \cdot P - c \cdot Q$. With extensible coordinates, this costs:

Decompressing R	$1S$
One doubling per bit	$3M + 4S$
One readdition every 6 bits in expectation	$8M/6$.
One mixed readdition every 9 bits in expectation	$7M/9$
Precomputation of about 8 points over 250 bits	$8M * 16/250$
Total	$5.62M + 5S$

With $S \approx 0.8M$, this comes out to some $9.62M$ per bit. An alternative is to use the fastest available algorithms to compute $t \cdot P$ and $c \cdot Q$ separately, and then to combine them. For $t \cdot P$ we use the signed multi-comb method, and for $c \cdot Q$ we use a Montgomery ladder over $\mathcal{E}_m$. Using the $(5, 5, 10)$ combs, this costs:

49 mixed readditions	$49 \cdot 7M/250$
9 doublings	$9 \cdot (3M + 4S)/250$
One mixed ladder step per bit	$5M + 4S$
Total	$6.48M + 4.14S$

With $S \approx 0.8M$, this comes out to $9.79M$ per bit. In our implementation with $(5, 5, 10)$ combs, this split method was indeed slightly slower

— 152k cycles vs 148k cycles. With larger precomputed tables, the trade-off is slightly more favorable to the Montgomery method. Likewise, if the hash is truncated to 128 bits, the split method has an advantage. But with our parameters, the traditional linear-combination method is faster.

A.2 Verifying an addition

A tricky question arises in our split formulation, however. The Montgomery ladder only computes the u-coordinate of $c \cdot Q$, and without performing decompression, we do not have the v coordinate of either $c \cdot Q$ nor R. How then can we verify the addition, even if we weaken it to

$$t \cdot P \stackrel{?}{=} \pm R \pm c \cdot Q$$

We cannot compute the right side of this equation, even up to sign, without recovering a v-coordinate. However, we can still verify the equation given the u-coordinates of these values, say u_1, u_2 and u_3 for $t \cdot P$, R and $c \cdot Q$ respectively. Recall that these points lie on the Montgomery curve

$$\mathcal{E} : v^2 = u^3 + Au^2 + u$$

The verification equation will hold if some line

$$v = mu + b$$

intersects $\mathcal{E}$ in three points with the given u-coordinates, with appropriate multiplicities if the coordinates are repeated. Thus we will have

$$\begin{aligned} u^3 + Au^2 + u - (mu + b)^2 &= (u - u_1)(u - u_2)(u - u_3) \\ &= u^3 - (u_1 + u_2 + u_3)u^2 \\ &\quad + (u_1u_2 + u_2u_3 + u_3u_1)u + u_1u_2u_3 \end{aligned}$$

whence

$$m^2 = u_1+u_2+u_3+A, \quad 2mb = 1-u_1u_2-u_2u_3-u_3u_1 \text{ and } b^2 = u_1u_2u_3$$

These equations will be solvable (i.e. the line will exist) over a quadratic extension of $\mathbb{F}$ if and only if

$$4(u_1+u_2+u_3+A)(u_1u_2u_3) = (1-u_1u_2-u_2u_3-u_3u_1)^2$$

If m and b are not in $\mathbb{F}$, then since their squares and product are in $\mathbb{F}$, they must both be "pure imaginary". In this case, we would still have

$$t \cdot P = \pm c \cdot Q \ \pm \ R$$

with $t \cdot P, c \cdot Q$ and R lying not on $\mathcal{E}$, but on its quadratic twist. However, this cannot happen because P and Q are on $\mathcal{E}$. So to check that $t \cdot P \stackrel{?}{=} \pm R \ \pm \ c \cdot Q$, it suffices to check this equation, which takes about as long as a point addition.

Note that u_1 and u_3 above will be given in projective coordinates, i.e. as U_1/Z_1 and U_3/Z_3, respectively; the formula is easily adapted to this case by clearing the denominators. In this case, the formula is still correct even if one or both of the points is the identity.

This formula is easily adapted to curves in other cubic forms, even over binary fields. For twisted Edwards curves, the birational equivalence to Montgomery curves computes the u coordinate on the Montgomery curve using only the y coordinate of the Edwards curve, so this equation can check relations on twisted Edwards curves with Y, Z coordinates as well.

A.3 Checking the v coordinate

The split equation does not check the v coordinate given in signatures. However, we will show in this section that it is possible to do so.

Given the v-coordinate of one of the points, we can solve for those of the others. We have

$$\begin{aligned} v_1 \cdot v_2 &= (mu_1 + b) \cdot (mu_1 + b) \\ &= m^2 u_1 u_2 + mb(u_1 + u_2) + b^2 \\ &= \frac{1}{2}\big((u_1 u_2 + 1)(u_1 + u_2) - u_3(u_1 - u_2)^2 + 2Au_1 u_2\big) \end{aligned}$$

by plugging in the formulas for $m^2, 2mb$ and b^2. Since the Edwards scalar-multiplication algorithm and isomorphism will produce the v-coordinate of $t \cdot P$, we can recover that of R and that of $c \cdot Q$. Since this last value was computed with a Montgomery ladder, we also have access to the u-coordinates of Q and $(c + 1) \cdot Q$, we can repeat this equation again to solve for the v-coordinate of Q.

Since we chose the Legendre symbol as our sign bit, we can check the sign of the v-coordinates with one Legendre symbol calculation each. In our experiments, the Legendre symbol takes about 40% as long as an inversion by exponentiation, so it is cheaper to verify both v-coordinates than to actually compute either of them.

B Other prime fields

We considered several other fields for our implementation. For clarity, let $p_{252} = 2^{252} - 2^{232} - 1$ be the prime we chose.

B.1 Special primes with Barrett reduction

We first considered a conventional choice such as $2^{255} - 19$, $2^{256} - 189$ or similar with Barrett reduction. We found Montgomery reduction to be significantly more efficient than Barrett reduction, because no contortions are required to limit carry-propagation. Multiplication modulo these primes took some 80 cycles in our tests, compared with 55 cycles for p_{252}. Perhaps more tuning would improve this.

Bernstein [3] changes the radix from 2^{64} to 2^{51} and stores elements in 5 words. This helps on processors such as Sandy Bridge which sport fast multipliers and slow carries – but in our measurements the increased number of multiplies, higher register pressure and larger memory footprint negated these benefits.

B.2 Slightly larger fields

We strongly considered using $p_{256} := 2^{256} - 2^{194} - 1$. This special Montgomery prime is slightly larger than our p_{252}, and its round size and Solinas form are appealing. Reductions can be done with shifting and addition; and its Barrett reduction algorithm is faster than that of p_{252}. We need one extra Montgomery reduction in the multiplication step to reduce from less than $2p$ to less than 2^{256}. However, if we wish to keep elements under 4 words with p_{256}, we will also have to reduce after every addition or subtraction; expanding to 5 words would slow down multiplication and worsen register pressure.

The performance difference between these primes depends heavily on the platform. On Sandy Bridge, p_{256} operations take 8% longer than p_{252}; on Core 2 Conroe (which sports a respectably fast multiplier, but slow add-with-carry), they take up to 50% longer; on AMD K8 (with a slower multiplier, but fast add-with-carry), they take 2% less time. We did not implement p_{256} arithmetic on ARM, but we conjecture that it would perform slightly worse than p_{252}.

In light of of p_{256}'s slow performance on Conroe, we chose p_{252} for this paper. However, we are interested in future work with p_{256}.

Adding an extra bit for point compression would ruin the round 256-bit size of p_{256} — but our compression technique does not require an extra bit (see Section 3.3).

We also considered $2^{255} - 2^{224} \cdot 29 - 1$. This prime is a compromise between p_{256} and p_{252}, and its non-Solinas form hurts the performance of Barrett reduction, but it does allow some degree of laziness. It also has an extra bit for point compression in protocols over prime-order curves, where the compression techniques in Section 3.3 do not apply.

B.3 Much larger fields

We are interested in trying larger pseudo-Mersenne and/or trinomial primes, but they are beyond the scope of this work.

B.4 Extension fields

We considered odd-prime-power extension fields, particularly the popular $\mathbb{F}((2^{127}-1)^2)$. The smaller multiplies and double reductions required for this field lengthen its multiplication to about 75 cycles in our implementation, with or without Karatsuba. However, this field allows endomorphisms for faster point multiplication [13] on a wide variety of curves [12].

The disadvantages of this field do not apply as strongly in larger extension fields, such as $\mathbb{F}((2^{192}-2^{64}-1)^2)$. But in this paper we target a security level near 128 bits.

B.5 Primes supporting endomorphisms

We considered primes supporting ordinary curves with endomorphism. For example, we looked into $2^{252}-739448\cdot 2^{224}-1$; $2^{255}-209264\cdot 2^{224}-1$; $2^{256}-44416\cdot 2^{224}-1$; etc. Modulo these primes, both

$$\mathcal{E}_3 : y^2 = x^3 - \frac{3}{4}x^2 - 2x - 1$$

and its quadratic twist have 8·prime order, and -7 is a quadratic residue. Such curves have complex multiplication modulo $(1+\sqrt{-7})/2$, and therefore support an efficiently computable endomorphism [13].

We leave exploration of such endomorphisms to future work. We suspect that they will noticeably speed up verification and key exchange, but not key generation or signing.