Curve41417: Karatsuba revisited

Daniel J. Bernstein[1,2], Chitchanok Chuengsatiansup[2], and Tanja Lange[2]

[1] Department of Computer Science
University of Illinois at Chicago
Chicago, IL 60607–7045, USA
djb@cr.yp.to
[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
c.chuengsatiansup@tue.nl, tanja@hyperelliptic.org

Abstract. This paper introduces constant-time ARM Cortex-A8 ECDH software that (1) is faster than the fastest ECDH option in the latest version of OpenSSL but (2) achieves a security level above 2^{200} using a prime above 2^{400}. For comparison, this OpenSSL ECDH option is not constant-time and has a security level of only 2^{80}. The new speeds are achieved in a quite different way from typical prime-field ECC software: they rely on a synergy between Karatsuba's method and choices of radix smaller than the CPU word size.

Keywords: performance, Karatsuba, refined Karatsuba, reduced refined Karatsuba, radix choices, vectorization, Edwards curves, Curve41417

1 Introduction

This paper introduces new ECDH software for a standard ARM Cortex-A8 CPU. This software is faster than the fastest ECDH option (`secp160r1`) in the latest version of OpenSSL (version 1.0.2-beta1, released 24 February 2014).

This performance bar was already reached in one previous paper, "NEON crypto" (CHES 2012) by Bernstein and Schwabe [11], implementing Bernstein's Curve25519 [2] elliptic curve. The difference is that we now reach the same performance bar at a much higher security level, implementing a very strong new "Curve41417" elliptic curve introduced informally by Bernstein and Lange in [7, page 12] and introduced formally in this paper.

We are not saying that Curve41417 is as fast as Curve25519. We are saying that it is fast enough for applications and provides a much higher security level than Curve25519. This paper addresses the scalability challenges that appear at higher security levels.

This work was supported by the National Science Foundation under grant 1018836 and by the Netherlands Organisation for Scientific Research (NWO) under grants 639.073.005 and 613.001.011. Permanent ID of this document: `8302181bad3a3e2fcf91ee3e72b49edd`. Date: 2014.07.06.

Hyperelliptic-curve DH has also recently reached this performance bar for the Cortex-A8: the HECDH implementation in [4] is even faster than Curve25519. However, the performance benefits of hyperelliptic curves are specific to DH, as admitted in [4], while elliptic curves are easily adapted to other important applications such as signatures. More importantly, the 128-bit hyperelliptic curve used in [4] came from a massive computation by Gaudry and Schost in [20], using more than 1000000 hours of CPU time. Finding a similar curve at a higher security level would be extraordinarily difficult.

1.1. Karatsuba's method in prime-field ECC software. The Cortex-A8 contains a large integer-multiplication unit that multiplies 32-bit words to produce 64-bit results. Of course, there are CPUs with even larger multipliers, and CPUs (and FPGAs) with smaller multipliers, but 32-bit multipliers have been a popular choice for many years and seem likely to remain in widespread use in embedded systems for many years to come. We focus on the Cortex-A8 for the same reasons as [11] and [4, Section 5].

The conventional approach in ECC software is to take advantage of 32-bit multipliers by splitting, e.g., 160-bit prime-field elements into 5 words to be multiplied, or 256-bit prime-field elements into 8 words to be multiplied. Karatsuba's method [29, Theorem 2] is well known to be useful for binary fields, and is occasionally also considered for prime-field ECC software, but is practically always dismissed as having too much overhead: one Karatsuba level saves 25% of the integer-multiply instructions, but this is outweighed by the cost of many extra additions. (Of course, this comparison is biased by the availability of a large multiplier and relatively little area spent on adders, but this is how mass-market CPUs have always been designed.)

It should be obvious that scaling to larger and larger input sizes will eventually reach a cutoff where one Karatsuba level is useful: the overhead is linear in the size, while the 25% savings is quadratic in the size. But the conventional wisdom is that this cutoff is far beyond ECC sizes, so one would not expect that aiming for high-security ECC would reach this cutoff. The heavily optimized GMP multiprecision library [22], which includes automated searches for optimal cutoffs, does not switch over from schoolbook multiplication to one Karatsuba level on the Cortex-A8 until it reaches 832-bit inputs. A recent RSA performance analysis by Bos, Montgomery, Shumow, and Zaverucha [15] avoided all use of Karatsuba's method even for 1024-bit modular multiplication.

We use *two* Karatsuba levels. There is a synergy between two design choices here: (1) we use Karatsuba's method; (2) we use a radix smaller than the CPU word size.

The conventional choice for b-bit CPUs is to use radix 2^b, minimizing the number of words that need to be multiplied. See, for example, the recent DH software from [24], [32], [13], [19], and [16]. However, a corner of the DH literature uses a smaller radix, with the goal of delaying carries, the same way that hardware multipliers typically use carry-save adders. See, for example, [11] and [4].

This corner of the literature does not seem to have exploited the fact that Karatsuba's method benefits heavily from a smaller radix. With radix 2^b, the extra additions in Karatsuba's method are add-with-carry chains. With a smaller radix, the extra additions in Karatsuba's method are independent additions without carries. Even on CPUs where add-with-carry is as cheap as add, having independent operations creates tremendous extra flexibility in register allocation, instruction scheduling, and vectorization.

Conversely, a smaller radix benefits from Karatsuba's method, especially as the security level increases. Reducing a radix from, e.g., 2^{32} to 2^{26} means that instead of w words one now needs $(32/26)w$ words and thus, without Karatsuba's method, $(32/26)^2w^2 \approx 1.5w^2$ multiplications instead of w^2 multiplications; this means that the benefits of eliminating carries have to be compared to the loss of $0.5w^2$ multiplications. Karatsuba's method moves the number of multiplications down to a smaller scale, improving this tradeoff.

1.2. Choice of prime and choice of curve. The standard NIST elliptic curves [34] use primes p designed to allow easy computation of $x \bmod p$ in radix 2^{32}. For example, the popular NIST P-256 curve uses $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$, and at a higher security level NIST P-384 uses $p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$.

We leave a gap between our radix and 2^{32} to speed up multiplications, as explained above, but this makes computation of $x \bmod p$ quite painful for the NIST primes p. The NIST primes are also suitable for a much smaller radix, namely 2^{16}, but that radix would make our multiplications considerably slower.

The Curve25519 prime, $2^{255} - 19$, is much less sensitive to the choice of radix, but our objective is to provide as much security as possible subject to a specified performance requirement, and in particular more security than Curve25519. An initial performance estimate indicated that a carefully designed curve of 384 bits or larger could meet our performance requirement, but we found very few 384-bit curves in the literature, and all of them have obvious performance problems.

We therefore designed a prime and curve from scratch. This also allowed us to take advantage of state-of-the-art curve shapes, while meeting stringent security criteria that are flunked by the NIST curves. See Section 2.

The prime we ended up with, namely $p = 2^{414} - 17$, has many attractive features from a performance perspective. It is extremely close to a power of 2. The difference 17 has just two bits set, allowing $2^{414}x \bmod p$ to be computed as $16x + x$ with a single shift-and-add operation. The exponent 414 is divisible by $9, 18, 23, 46$ and the exponent 416 (for $4p$) is divisible by $8, 13, 16, 26, 32, 52$, allowing easy choices of integer radix suitable with low overhead for practically any size of multiplier. A field element is easily transmitted in 32-bit words with under 1% wasted space ($13 \cdot 32 = 416$), while still allowing two extra bits for extensions, such as a bit typically used in encoding a compressed curve point.

For our software we decided to use a slightly harder, but slightly more efficient, non-integer radix, namely $2^{414/16} = 2^{25.875}$. We split 414-bit prime-field elements into 16 words, use one Karatsuba level to reduce 16-word multiplication to three 8-word multiplications, and use another Karatsuba level to reduce each 8-word multiplication to three 4-word multiplications. See Section 4 for details

curve	i.MX515 op/s	cycles	Sitara op/s	cycles
`secp160r1`	379.2	≈2.1 million	468.1	≈2.1 million
`nistp192`	274.3	≈2.9 million	350.9	≈2.8 million
`nistp224`	200.4	≈4.0 million	257.6	≈3.9 million
`nistp256`	201.1	≈4.0 million	258.7	≈3.9 million
`nistp384`	60.1	≈13.3 million	75.9	≈13.2 million
`nistp521`	26.9	≈29.7 million	33.7	≈29.7 million

Table 1. Prime-field ECC timings from `openssl speed ecdh` on two Cortex-A8 devices. Warning: `openssl speed ecdh` reports "operations per second" as the reciprocal of average seconds per operation without indicating standard deviation or other stability metrics. "i.MX515 op/s" column is reported by OpenSSL 1.0.2-beta1 compiled with gcc 4.4.3 on a Hercules eCafe laptop (`h4mx515e`) with a 2009 Freescale i.MX515 CPU running at 800MHz. "Sitara op/s" column is reported by OpenSSL 1.0.2-beta1 compiled with gcc 4.7.3 on a BeagleBone Black development board (`bblack`) with a 2012 TI Sitara XAM3359AZCZ100 CPU running at 1000MHz. The "cycles" columns translate "op/s" into CPU cycles per operation.

of our multiplication strategy, and Section 5 for the extra challenges created by vectorization.

1.3. Expected scalability. As a measurement of the conventional scaling of ECC performance to higher security levels, we compiled OpenSSL 1.0.2-beta1 on two Cortex-A8 devices and ran `openssl speed ecdh`. The prime-field results are shown in Table 1. We also checked that (as expected) the prime-field results were faster than the binary-field results at each security level; the binary-field results are not shown here. The fastest OpenSSL cycle count was 2.1 million cycles for `secp160r1` (2^{80} security).

The following back-of-the-envelope calculation suggests that moving from 256 bits to 384 bits increases costs by a factor of $1.5^3 = 3.375$: each multiplication input is longer by a factor of 1.5, increasing the multiplication cost by a factor of 1.5^2; and the scalar in ECDH is $1.5\times$ longer. The actual ratios between `nistp256` and `nistp384` in the table are close to this. The slowdown factor for `nistp521` is about 7.5, noticeably better than $(521/256)^3 \approx 8.4$, presumably because of the simpler prime shape used in P-521. The speedup factor for smaller curves is considerably worse than this calculation would suggest; presumably this reflects OpenSSL function-call overheads that become troublesome for smaller integers.

We also checked the eBACS [8] benchmarking site for Cortex-A8 results. The only results faster than 2.1 million cycles were 0.46 million cycles (i.MX515) and 0.50 million cycles (Sitara) for the Curve25519 (2^{125} security) implementation from [11]. The paper [4] reports better speeds, just 0.27 million Cortex-A8 cycles for HECDH; but scaling HECDH to higher security levels is very difficult, as mentioned earlier. The paper [14] reports 0.77 million Cortex-A8 cycles for 2^{103} security using a different type of curve, evidently not competitive.

The same type of back-of-the-envelope calculation suggests that moving from Curve25519 up to Curve41417 would cost a factor of 4.3, increasing 0.50 million

Sitara Cortex-A8 cycles to 2.15 million cycles. We do considerably better than this; see below.

1.4. Performance results. We tried our Curve41417 software on the same two Cortex-A8 machines shown in Table 1. On the FreeScale i.MX515 (`h4mx515e`) our software uses just 1648409 cycles (median; quartiles 1646391 and 1662710). On the TI Sitara (`bblack`) our software uses just 1775804 cycles (median; quartiles 1774878 and 1782850). These figures are for a complete scalar-multiplication operation, including unpacking a point from network format, precomputation, main computation, final inversion, and converting the result back to network format. We emphasize that our curve choice has security level above 2^{200}, and that the software is free of data-dependent branches and data-dependent array indices.

These speeds are, despite their very high security level, considerably faster than the 2.1 million cycles for the fastest ECDH in OpenSSL. These speeds are also considerably faster than the 2.15 million cycles predicted above by extrapolation from Curve25519. This paper explains the design and implementation choices that led to this performance.

As a followup to our initial Curve41417 announcement, Hamburg announced a similar, slightly larger, curve "Ed448-Goldilocks". Hamburg's most recent performance report [25] says 3.6 million Cortex-A9 cycles for Ed448-Goldilocks, compared to 4.4 million Cortex-A9 cycles for the implementation of NIST P-256 in OpenSSL 1.0.1. There are several reasons that it is difficult to extrapolate from these results: the Cortex-A9 is not the same as the Cortex-A8; Hamburg's Ed448-Goldilocks software is not vectorized; and OpenSSL 1.0.1 was missing some NIST P-256 speedups that appear in the most recent version of OpenSSL.

1.5. Is high security useful? Most papers today consider security levels between 2^{80} and 2^{128}. The adequacy of 2^{80} is frequently a subject of dispute. There is general consensus that well-funded attackers and botnets can already perform 2^{80} operations; most HTTPS web sites have now switched from RSA-1024 (2^{80} security) to RSA-2048 (2^{112} security) or 256-bit ECC (2^{128} security). On the other hand, there are also many papers continuing to study 2^{80} security and stating that 2^{80} is ample protection for low-value targets.

The adequacy of 2^{128} is rarely a subject of dispute. It is easy to see that 2^{128} is far beyond any computation feasible today. Choosing 2^{128} is so common in the current literature that papers studying a 2^{128} security level rarely bother to justify this choice.

One can therefore reasonably ask whether there is any reason to go beyond 2^{128} security, and in particular whether we are accomplishing anything useful by going beyond 2^{200} security. We give five answers to this question, in what we consider to be increasing order of importance.

First, cryptographic primitives need time to be reviewed before they are standardized and deployed in embedded systems, so designers of cryptographic primitives today should be considering embedded systems designed at least 10 years from now. Some of those systems will have a lifetime of 30 years, and at the end of that lifetime could still be encrypting data that — even if recorded

by an attacker — should remain confidential for another 30 years, i.e., 70 years from now.

Today's mass-market GPUs perform approximately 2^{58} floating-point operations per year per watt. If computation becomes a factor of 10 more efficient each decade then mass-market chips in 70 years will perform approximately 2^{81} floating-point operations per year per watt. Carrying out a 1-year computation on the same scale as 2^{128} floating-point operations will thus require just 2^{47} watts. For comparison, the Earth's surface receives 2^{56} watts from the Sun.

We do not mean to suggest that typical cryptographic applications should worry about such large attacks. But we also see value in designing cryptographic systems that are not broken by such large attacks.

Second, even though many researchers have studied the security of ECC and expressed confidence in the security of prime-field ECC, there is still the possibility of an algorithmic breakthrough that considerably reduces the amount of computation required to break ECC. By moving to a much higher security level we are providing a security margin against unexpected attack improvements.

For comparison, over the past 18 months the security of small-characteristic multiplicative-group discrete logarithms has dropped dramatically. A very recent paper [21] reports 2^{59} security for a system previously thought to provide 2^{128} security. We do not mean to suggest that this is a threat to prime-field ECC (there are clear barriers between small characteristic and prime fields, and more importantly between multiplicative groups and ECC) but it does illustrate the general principle that attack cost can suddenly drop.

Third, sometimes cryptographic protocols are not as secure as the underlying cryptographic primitives. Often there is a security proof putting a bound on the gap, but usually the security proofs are not "tight". In particular, many ECC protocols are not guaranteed to provide 2^{128} security using 256-bit curves, even assuming the standard security conjectures for ECDLP on those curves. Achieving a 2^{128} guarantee requires taking larger curves. We thank an anonymous referee for pointing out this argument.

Fourth, we suggest that the right question is not how efficiently a particular security level can be achieved, but rather how much security can be provided subject to the performance requirements set by the users. Of course, a typical cryptographic system also relies on block ciphers, hash functions, etc., and if those are breakable in time 2^{128} then the attacker does not have to bother breaking a 414-bit elliptic curve; but AES-256 costs only 40% more than AES-128, and standard hashes also provide high-security options. It is natural for research into high-performance ECC to similarly provide high-security options for users who can afford those options.

The normal reason for users to reject high-security options is not that the users dislike high security, but rather that the high-security options are too slow. If a user rejects OpenSSL's `nistp384` in favor of `secp160r1`, probably the reason is that the user's performance budget does not allow 13.3 million cycles, while it does allow 2.1 million cycles. Unless there are severe bandwidth constraints,

the user will be happier with Curve41417, which provides much higher security within the same performance budget.

Fifth, there are at least some users already demanding cryptography beyond a 2^{128} security level. For example, NSA's Suite B allows NIST P-256 for Secret information, but for Top Secret information it requires NIST P-384, SHA-384, and AES-256. This project began when Silent Circle requested a non-NIST curve to replace NIST P-384; we realized that we could design a curve that simultaneously provided better performance and better security. Silent Circle is now using Curve41417 by default.

2 Design of Curve41417

The IEEE standard P1363 [27] and the Brainpool recommendations [17] specify procedures to generate secure elliptic curves. Research has identified several other properties a secure curve should satisfy. A recent collection of these properties is provided by Bernstein and Lange in the "SafeCurves" web site [10].

2.1. Standard security criteria. There are several standard criteria on which all methods cited on [10] agree. The elliptic curve E must be defined over a prime field $\mathbf{F}_p$ or a binary field $\mathbf{F}_{2^p}$, for p a prime; its group order must be divisible by a large prime ℓ; this prime must not match the field characteristic; and the embedding degree must be large. Over a prime field $\mathbf{F}_p$ the embedding degree is defined as the smallest positive integer k so that ℓ divides $p^k - 1$. Brainpool requires $k \geq (\ell - 1)/100$, and P1363 imposes a weaker requirement.

For efficiency and security reasons we focus on prime fields, a recommendation supported by Brainpool and the more recent NIST/NSA documents [35].

2.2. Additional security criteria. SafeCurves imposes several further requirements to avoid "conflicts between simplicity, efficiency, and security". Specifically, it requires curves to support "simple, fast, complete, constant-time" algorithms for single-coordinate single-scalar multiplication and for multi-scalar multiplication. Montgomery curves [33] meet the single-coordinate single-scalar requirement; Edwards curves [18], when chosen to be complete [6], meet all of the requirements. Compared to Weierstrass curves, these curves make it easier to implement the curve arithmetic correctly: scalar multiplication is a very regular operation without exceptional cases that require special handling and that could reveal information about the scalar. The NIST curves do not meet these requirements.

SafeCurves also requires curves to be twist-secure. Twist-security means that the order of the twist, namely $2p + 2 - \#E(\mathbf{F}_p)$, is nearly prime. This criterion eliminates security problems caused by single-coordinate single-scalar multiplication algorithms that do not take extra effort to validate their inputs: for example, when a curve is given in Montgomery form and only the x-coordinate is transmitted and used, twist-security eliminates the need to check that the incoming x-coordinate is on the curve.

The NIST curve constants are not explained: in the SafeCurves terminology, the NIST curve choice is not "rigid". This has led to speculation about how the

NIST curves were designed and about whether the NSA has implemented a back door in the choice of the curves. Our curve is "fully rigid": the prime and all curve constants are fully explained here.

2.3. Choice of prime field. Our target in designing the new curve was to generate an elliptic curve at a security level larger than 2^{192} that meets the SafeCurves requirements and that supports efficient implementations. To this aim we start with finding a prime for which field elements can be efficiently represented and modulo which reductions are efficient. Prime numbers of the form $2^j - c$ for $12 \cdot 32 < j < 13 \cdot 32$ and $0 < c < 32$ are rare: the only possibilities are $2^{389} - 21$, $2^{401} - 31$, $2^{413} - 21$, and $2^{414} - 17$. We selected $p = 2^{414} - 17$ because 17 is the smallest c in this list; it also has the lowest Hamming weight. Section 4 explains how we perform arithmetic in $\mathbf{F}_p$; this prime also leaves enough space in the limbs when we represent field elements as 16 words of 32 bits that carries between the limbs and reductions modulo p can be delayed for long enough to be useful in the curve arithmetic. The next larger candidate prime would be $2^{444} - 17$ which does not have this feature; our p is already very large for our security needs.

2.4. Choice of curve shape. For efficient and secure arithmetic in Diffie–Hellman key exchange and digital signature applications we insist on a curve in Edwards form. Note that each curve in Edwards form is birationally equivalent to one in Montgomery form, so there is no need to choose one over the other. The coefficient d in the Edwards curve $x^2 + y^2 = 1 + dx^2y^2$ appears as a factor in the addition formulas, so choosing d to be small in absolute value is good for efficiency. For security we choose a complete Edwards curve (d is not a square in $\mathbf{F}_p$) and insist on the same level of twist-security as Curve25519 — the cofactors of the curve and its twist are in $\{4, 8\}$.

2.5. A safe curve. Curve41417 (named after the prime field) is defined as

$$x^2 + y^2 = 1 + 3617x^2y^2 \text{ over } \mathbf{F}_p, \qquad p = 2^{414} - 17.$$

Its order is 8ℓ, where
$\ell = 2^{411} - 3336414086375514252081017769409838517898472720041120858959475$.
The order of the twist is also 8 times a prime. The value $d = 3617$ is the smallest integer in absolute value meeting the above security requirements.

3 ECC arithmetic

Our featured application is static Diffie–Hellman in which a user Alice computes her private key a and her public key $P_A = aP$ once and then publishes P_A. If Alice wants to communicate with user Bob she looks up Bob's public key P_B and computes aP_B. This means that the computations use variable base points. The computations involve the long-term secret key a and need to be protected against side-channel attacks by attackers sitting on the same device or having a connection to it. This means in particular that the scalar multiplication should

run in constant time, independent of the scalar a, and that there should be no data-dependent branches or table lookups involving a.

We use a windowing method with fixed window width for constant-time single-scalar multiplication on Curve41417 in Edwards form. Our analysis also allows good estimates of, e.g., the cost of signature verification using Curve41417. Another option for single-scalar multiplication is the Montgomery ladder for the Montgomery form of Curve41417; this is not quite as fast as the Edwards form but has the advantage of fitting the computation into less SRAM.

3.1. Coordinate systems. The fastest doubling formulas in the EFD [9] for curves in Edwards form are in projective coordinates X, Y, Z with $x = X/Z, y = Y/Z$ for $Z \neq 0$. These take $3\mathbf{M} + 4\mathbf{S}$ per doubling where $\mathbf{M}$ and $\mathbf{S}$ denote field multiplication and field squaring respectively. See Appendix A for the formulas used in this paper.

The fastest addition formulas are in extended coordinates X, Y, Z, T with $x = X/Z, y = Y/Z$, and $xy = T/Z$ for $Z \neq 0$. These take $9\mathbf{M} + 1\mathbf{M}_d$. Here $\mathbf{M}_d$ is a multiplication by curve constant d; for us $d = 3617$, which is significantly smaller than p, so this multiplication $\mathbf{M}_d$ is cheaper than general multiplications $\mathbf{M}$. (The curve $-x^2 + y^2 = 1 - dx^2y^2$ allows faster additions, saving $1\mathbf{M}$ in each addition. If -1 were a square in $\mathbf{F}_p$ then we could apply an isomorphism to that curve. However, -1 is not a square in $\mathbf{F}_p$, so that curve is not complete.)

Achieving the best performance requires combining these two coordinate systems: computing the extra T coordinate for a doubling output that will be used for addition, and skipping the extra T coordinate for an addition output that will be used only for doubling. This suggestion was made in [26], the paper introducing extended coordinates.

3.2. Scalar multiplication. Constant-time sliding windows are difficult so we use fixed windows. We analyzed operation counts for signed fixed windows for window widths $w = 4$, $w = 5$, and $w = 6$, and concluded that $w = 5$ is optimal. We therefore precompute $0P_B = (0, 1), P_B, 2P_B, \ldots, 16P_B$ and store the results in a table. We do table lookups in constant time using the same technique as in, e.g., [5]: we load the entire table into registers and perform the selection via arithmetic.

Precomputation is done as follows. We double P_B to obtain $2P_B$; add P_B to obtain $3P_B$; double $2P_B$ to obtain $4P_B$; add P_B to obtain $5P_B$; double $3P_B$ to obtain $6P_B$; add P_B to obtain $7P_B$; and so on through $16P_B$. We also multiply each resulting T coordinate by $d = 3617$, eliminating the multiplications by d in the main computation.

In total 8 doublings, 7 additions, and 16 multiplications by d are required. Note that these doublings are followed by additions and thus need one extra $\mathbf{M}$ for the T coordinate in the transition to extended coordinates. Note also that we have to compute T for P_B which costs $1\mathbf{M}$. For the first doubling, (X, Y, Z, T) is $(x, y, 1, xy)$. We save $1\mathbf{S}$ by not having to compute Z^2 since $Z = 1$; we save another $1\mathbf{S}$ by not having to compute $(x + y)^2$ but using the equality $(x + y)^2 - x^2 - y^2 = 2xy = 2T$; and we use $Z = 1$ again for an $\mathbf{S} - \mathbf{M}$ tradeoff. The overall cost for the first doubling is $3\mathbf{M}+3\mathbf{S}$ while for the rest it is $4\mathbf{M}+4\mathbf{S}$.

Note that all additions in the precomputation are adding P_B which has $Z = 1$. We thus use mixed addition which saves $1\mathbf{M}$. This results in the total cost for precomputation of $1\mathbf{M} + (3\mathbf{M} + 3\mathbf{S}) + 7(4\mathbf{M} + 4\mathbf{S}) + 7(8\mathbf{M}) + 16\mathbf{M}_d = 88\mathbf{M} + 31\mathbf{S} + 16\mathbf{M}_d$.

The main computation uses a fixed pattern of five doublings followed by one addition. Four regular doublings in a block of five take $3\mathbf{M} + 4\mathbf{S}$ each. The fifth doubling in a block requires 1 more $\mathbf{M}$ to calculate T for the following addition. On the other hand, addition does not need to compute T since the following doubling is in projective coordinates. Furthermore, dT was precomputed for each T in the table. Therefore the addition takes only $8\mathbf{M}$. In total the five doublings and one addition take only $4(3\mathbf{M} + 4\mathbf{S}) + (4\mathbf{M} + 4\mathbf{S}) + (8\mathbf{M}) = 24\mathbf{M} + 20\mathbf{S}$.

Note that, since the Edwards addition law is complete, no special handling is required for the neutral element $0P_B$. An addition when the coefficient of the scalar happens to be 0 is handled the same way as any other addition.

A scalar between 0 and $2^{414} - 1$ uses 82 signed windows of width 5, after an initial selection from $0P_B, 1P_B, \ldots, 16P_B$. The total cost for scalar multiplication including precomputation is $(88\mathbf{M} + 31\mathbf{S} + 16\mathbf{M}_d) + 82(24\mathbf{M} + 20\mathbf{S}) = 2056\mathbf{M} + 1671\mathbf{S} + 16\mathbf{M}_d$, plus 1 inversion and $2\mathbf{M}$ to convert to $X/Z, Y/Z$ for output.

4 Karatsuba multiplication

Karatsuba, Toom, and the FFT are polynomial-multiplication methods that are asymptotically faster than schoolbook multiplication. However, for small input sizes the speedups are outweighed by the expense of more additions and subtractions, which in turn require more carries. These effects are particularly noticeable for polynomials of low degree — or equivalently for integers occupying just a few words. In software implementations of cryptography we rarely find integers large enough to justify use of FFT or Toom, and even Karatsuba's method is commonly only used in implementations of RSA and not ECC.

In this section we explain how to reduce the cost of carries by working with multiple levels of redundancy in the representation and thereby delaying carries. We also introduce "reduced refined Karatsuba", a new variant of the "refined Karatsuba" method; this variant eliminates some additions by merging Karatsuba multiplication with a subsequent modular reduction.

4.1. Redundant number representation. We decompose an integer f modulo $2^{414} - 17$ into 16 integer pieces in radix $2^{414/16} = 2^{25.875}$, i.e., we write f as $f_0 + 2^{26} f_1 + 2^{52} f_2 + 2^{78} f_3 + 2^{104} f_4 + 2^{130} f_5 + 2^{156} f_6 + 2^{182} f_7 + 2^{207} f_8 + 2^{233} f_9 + 2^{259} f_{10} + 2^{285} f_{11} + 2^{311} f_{12} + 2^{337} f_{13} + 2^{363} f_{14} + 2^{389} f_{15}$. With this decomposition, each limb $f_0, f_1, \ldots, f_{14}, f_{15}$ is small enough to fit into a 32-bit integer and to still have space to delay carries occurring when adding these pieces. The results of the 32-bit-by-32-bit multiplications fit into 64-bit words, and we can add thousands of them together before causing an overflow.

Note that f_7 is multiplied by 2^{207}, not 2^{208}. Having f_7 and f_{15} contain 25 bits makes $f_0, \ldots, f_7$ symmetric to $f_8, \ldots, f_{15}$, aiding vectorization. We considered

using fewer limbs, but the advantage of saving multiplications is outweighed by the disadvantages of (1) extra carries and (2) extra vectorization overhead.

4.2. Two-level Karatsuba: decomposition strategy. As mentioned in Section 1, we use 2 Karatsuba levels. This fits nicely into the 128-bit Cortex-A8 vector units, and uses less arithmetic than 3 or 1 (or 0) Karatsuba levels.

We start with what Bernstein in [3] calls the "refined Karatsuba identity"

$$(F_0 + t^n F_1)(G_0 + t^n G_1) = (1 - t^n)(F_0 G_0 - t^n F_1 G_1) + t^n (F_0 + F_1)(G_0 + G_1).$$

This uses fewer additions than the original Karatsuba identity from [29].

For the *first* level of Karatsuba, we split one 16-limb integer f into two 8-limb integers F_0 and F_1 with $f = F_0 + 2^{207} F_1$ as:

$$\begin{aligned} F_0 &= f_0 + 2^{26} f_1 + 2^{52} f_2 \ + 2^{78} f_3 \ + 2^{104} f_4 \ + 2^{130} f_5 \ + 2^{156} f_6 \ + 2^{182} f_7 \ ; \\ F_1 &= f_8 + 2^{26} f_9 + 2^{52} f_{10} + 2^{78} f_{11} + 2^{104} f_{12} + 2^{130} f_{13} + 2^{156} f_{14} + 2^{182} f_{15}. \end{aligned}$$

We also decompose another integer g similarly to f. Then, we have

$$fg = (1 - 2^{207})(F_0 G_0 - 2^{207} F_1 G_1) + 2^{207} (F_0 + F_1)(G_0 + G_1).$$

For the *second* level of Karatsuba, we further split the 8 limbs of F_0 (and those of F_1) into two 4-limb integers F_{00}, F_{01} (and F_{10}, F_{11}) with $F_0 = F_{00} + 2^{104} F_{01}$ (and $F_1 = F_{10} + 2^{104} F_{11}$) as:

$$\begin{aligned} F_{00} &= f_0 + 2^{26} f_1 + 2^{52} f_2 \ + 2^{78} f_3 \ ; & F_{01} &= f_4 \ + 2^{26} f_5 \ + 2^{52} f_6 \ + 2^{78} f_7; \\ F_{10} &= f_8 + 2^{26} f_9 + 2^{52} f_{10} + 2^{78} f_{11}; & F_{11} &= f_{12} + 2^{26} f_{13} + 2^{52} f_{14} + 2^{78} f_{15}. \end{aligned}$$

We similarly split G_0 and G_1 to obtain G_{00}, G_{01}, G_{10}, and G_{11}. Then

$$\begin{aligned} F_0 G_0 &= (1 - 2^{104})(F_{00} G_{00} - 2^{104} F_{01} G_{01}) + 2^{104} (F_{00} + F_{01})(G_{00} + G_{01}); \\ F_1 G_1 &= (1 - 2^{104})(F_{10} G_{10} - 2^{104} F_{11} G_{11}) + 2^{104} (F_{10} + F_{11})(G_{10} + G_{11}). \end{aligned}$$

To compute $(F_0 + F_1)(G_0 + G_1)$ we first compute $F_0 + F_1$ and $G_0 + G_1$ without carries and then apply the same type of decomposition. For example, we split $F_0 + F_1$ into two 4-limb integers, namely $F_{00} + F_{10}$ and $F_{01} + F_{11}$.

4.3. Lowest-level multiplication. On the lowest level we need to multiply two 4-limb integers; we do this by schoolbook multiplication. For $F_{00} G_{00}$ this works as follows:

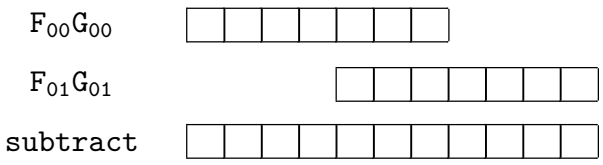
$$\begin{aligned} h_0 &= f_0 g_0, & & \\ h_1 &= f_0 g_1 + f_1 g_0, & h_4 &= f_1 g_3 + f_2 g_2 + f_3 g_1, \\ h_2 &= f_0 g_2 + f_1 g_1 + f_2 g_0, & h_5 &= f_2 g_3 + f_3 g_2, \\ h_3 &= f_0 g_3 + f_1 g_2 + f_2 g_1 + f_3 g_0, & h_6 &= f_3 g_3. \end{aligned}$$

We store each input limb f_i and g_i in a word of 32 bits and use the processor's multiplication and addition units to compute each h_i. This takes 16 32-bit-by-32-bit multiplications and 9 64-bit additions. Each of the initial limbs has at

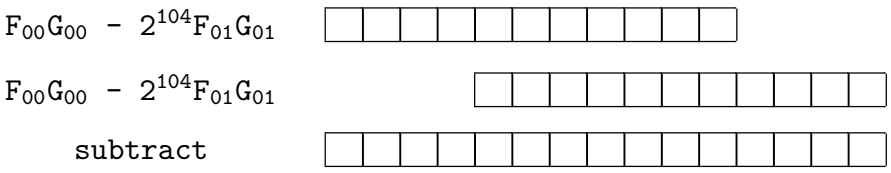
most 26 bits and each of the h_i fits into 64 bits. The values h_4, h_5, and h_6 belong to the powers $2^{104}, 2^{130}$, and 2^{156}, i.e., they are implicitly multiplied by 2^{104}.

4.4. Middle-level recombination. After computing the three lowest-level products $F_{00}G_{00}, F_{01}G_{01}$ and $(F_{00}+F_{01})(G_{00}+G_{01})$, we obtain F_0G_0 as follows.

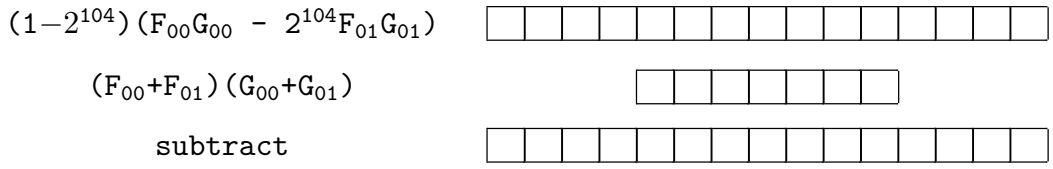
Step 1.1: Compute $F_{00}G_{00}-2^{104}F_{01}G_{01}$. We merge $F_{01}G_{01}$ to $F_{00}G_{00}$ at the 2^{104} boundary using 3 subtractions of 64-bit words. In other words, we align the 5th limb of $F_{00}G_{00}$ with the 1st limb of $F_{01}G_{01}$ as shown in the following diagram. The result is thus 11 limbs long. The top limbs are not actually subtracted from 0; they are tracked as being implicitly negated.



Step 1.2: Compute $(1-2^{104})(F_{00}G_{00}-2^{104}F_{01}G_{01})$. This is equivalent to merging $F_{00}G_{00}-2^{104}F_{01}G_{01}$ to itself at the 2^{104} boundary. We conduct this merge similarly to Step 1.1: we align the 5th limb of $F_{00}G_{00}-2^{104}F_{01}G_{01}$ with the 1st limb and subtract. The following diagram depicts this step. This merge requires 7 subtractions of 64-bit words, and the result is 15 limbs long.



Step 1.3: Compute F_0G_0. We finish this level of computation by adding $2^{104}(F_{00}+F_{01})(G_{00}+G_{01})$ to $(1-2^{104})(F_{00}G_{00}-2^{104}F_{01}G_{01})$. This is done by merging the former to the latter at the 2^{104} boundary, i.e., the 5th limb of $(1-2^{104})(F_{00}G_{00}-2^{104}F_{01}G_{01})$ is aligned with the 1st limb of $(F_{00}+F_{01})(G_{00}+G_{01})$ as shown in the following diagram. Note that this merge requires 7 additions of 64-bit words, and the result remains 15 limbs long.



When combining the results we need to pay attention to the 9th through 15th limbs. Those limbs are implicitly multiplied by 2^{207}. However, during the above computation they appear naturally as multiples of 2^{208} instead of 2^{207}. We therefore shift those seven limbs by one bit.

To summarize, the computation of the product F_0G_0 consists of

- 2×4 32-bit additions for $F_{00}+F_{01}$ and $G_{00}+G_{01}$;
- 3×16 32-bit-by-32-bit-producing-64-bit multiplications for $F_{00}G_{00}$, $F_{01}G_{01}$, and $(F_{00}+F_{01})(G_{00}+G_{01})$;

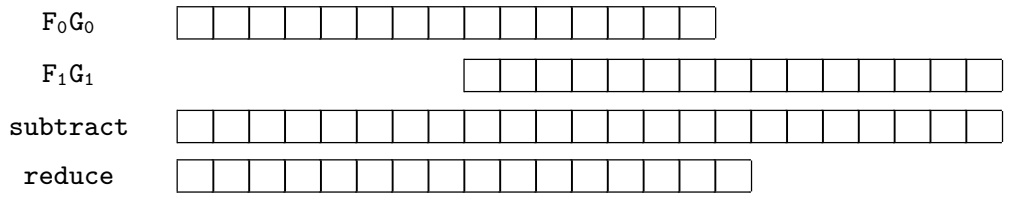
- 3×9 64-bit additions for computing the h_i;
- 1×3 64-bit subtractions for computing Step 1.1;
- 1×7 64-bit subtractions for computing Step 1.2;
- 1×7 64-bit additions for computing Step 1.3;
- 1×7 64-bit shifts for handling 2^{207} and 2^{208}.

The total is 8 32-bit additions (counting subtractions as additions), 48 32-bit-by-32-bit multiplications, 44 64-bit additions, and 7 64-bit shifts.

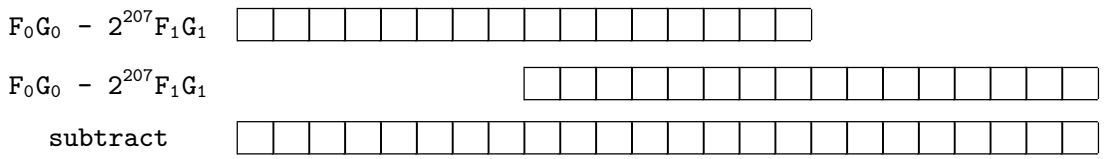
We compute products F_1G_1 and $(F_0 + F_1)(G_0 + G_1)$ in the same way as F_0G_0. The total cost for these three products and the computation of $F_0 + F_1$ and $G_0 + G_1$ is 40 32-bit additions, 144 32-bit-by-32-bit multiplications, 132 64-bit additions, and 21 64-bit shifts.

4.5. Top-level recombination and reduction. After computing F_0G_0 etc., we compute $fg = (1 - 2^{207})(F_0G_0 - 2^{207}F_1G_1) + 2^{207}(F_0 + F_1)(G_0 + G_1)$ as follows. This top-level recombination is immediately followed by a reduction, and we save some additions by interleaving the reduction into the refined-Karatsuba computation, a technique that we call "reduced refined Karatsuba". What is important here is that we reduce $F_0G_0 - 2^{207}F_1G_1$ before multiplying by $1 - 2^{207}$.

Step 2.1: Compute $F_0G_0 - 2^{207}F_1G_1$. This is similar to Step 1.1 but includes an extra reduction. The merge of F_1G_1 to F_0G_0 is at the 2^{207} boundary and uses 7 subtractions of 64-bit words; the 9th limb of F_0G_0 is aligned with the 1st limb of F_1G_1. The intermediate result is 23 limbs long. Then we reduce modulo $2^{414} - 17$: we multiply the 17th through 23rd limbs by 17 (using shifts and additions) and add to the 1st through 7th limbs. This requires another 7 shifts and 14 additions of 64-bit words. The result is thus only 16 limbs long, as indicated in the following diagram.

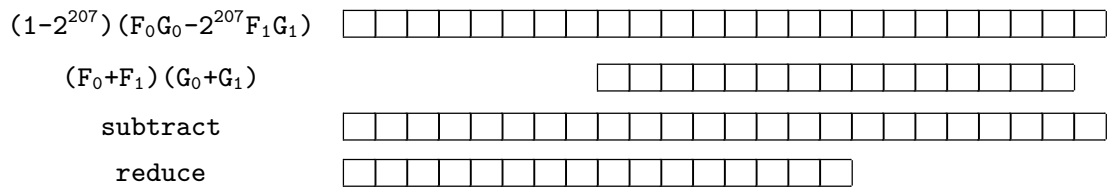


Step 2.2: Compute $(1 - 2^{207})(F_0G_0 - 2^{207}F_1G_1)$. This is similar to Step 1.2. The earlier reduction in Step 2.1 means that Step 2.2 uses only 8 subtractions of 64-bit words. The result is 24 limbs long as shown in the following diagram. We do *not* perform an extra reduction here: by keeping this long result of 24 limbs, we save 8 shifts and 16 additions.



Step 2.3: Compute fg. We finish by adding 15-limb $2^{207}(F_0+F_1)(G_0+G_1)$ to 24-limb $(1-2^{207})(F_0G_0-2^{207}F_1G_1)$. This is done by merging the former to the latter at the 2^{207} boundary: i.e., the 9th limb of $(1 - 2^{207})(F_0G_0 - 2^{207}F_1G_1)$ is

aligned with the 1st limb of $(F_0+F_1)(G_0+G_1)$. This merge requires 15 additions of 64-bit words and results in 24 limbs. We do another reduction similar to Step 2.1 to bring the result back to 16 limbs; this requires another 8 shifts and 16 additions of 64-bit words. The following diagram illustrates this step.



To summarize, computing fg from F_0G_0, F_1G_1 and $(F_0+F_1)(G_0+G_1)$ uses

- 7 64-bit subtractions for computing Step 2.1;
- 7 64-bit shift instructions for reduction in Step 2.1;
- 14 64-bit additions for reduction in Step 2.1;
- 8 64-bit subtractions for computing Step 2.2;
- 15 64-bit additions for computing Step 2.3;
- 8 64-bit shift instructions for reduction in Step 2.3;
- 16 64-bit additions for reduction in Step 2.3.

This sums up to 60 64-bit additions and 15 64-bit shift instructions. Therefore, the total cost for computing fg is 40 32-bit additions, 144 32-bit-by-32-bit multiplications, $132 + 60 = 192$ 64-bit additions, and $21 + 15 = 36$ 64-bit shifts.

4.6. Principles behind reduced refined Karatsuba. Our elimination of some additions can be viewed as following the general strategy of reducing *inputs* to a multiplication rather than *outputs* of a multiplication. Specifically, we reduce $F_0G_0 - 2^{207}F_1G_1$ before multiplying it by $1-2^{207}$; we do not reduce the product until after adding it to $(F_0 + F_1)(G_0 + G_1)$; if fg were being added to other products then we would similarly delay the reduction until after the addition. What is new here is seeing the multiplication by $1 - t^n$ inside refined Karatsuba as a useful target of the general strategy, despite the sparsity of $1 - t^n$.

5 Vectorization

The "NEON" vector unit in each Cortex-A8 core can compute a vector of two 64-bit products ac and bd in just 2 cycles given 32-bit inputs a, b, c, d. It can compute a vector of two 64-bit sums or four 32-bit sums in just 1 cycle. The latencies of these operations are actually higher, up to 7 cycles, but throughput is improved by pipelining. Taking advantage of this computational power requires that at every moment there are 2 or 4 identical computations to perform, and on top of this enough independent computations to hide latencies.

5.1. Karatsuba vectorization. Most of the computations in Section 4 are suitable for vectorization. For example, $F_{01}G_{01}$ takes $f_4, f_5, f_6, f_7, g_4, g_5, g_6, g_7$ as input; $F_{10}G_{10}$ takes $f_8, f_9, f_{10}, f_{11}, g_8, g_9, g_{10}, g_{11}$. There are no dependencies

between these two identical sets of multiplications. Similar comments apply to $F_{00}G_{00}$ and $F_{11}G_{11}$; $(F_{00}+F_{10})(G_{00}+G_{10})$ and $(F_{01}+F_{11})(G_{01}+G_{11})$; and $(F_{00}+F_{01})(G_{00}+G_{01})$ and $(F_{10}+F_{11})(G_{10}+G_{11})$. The remaining multiplication consists of 16 32-bit products, which we partition into 8 vectorized products at the cost of some shuffling. Similarly, we vectorize between combining F_0G_0 and combining F_1G_1, and at the cost of some shuffling we vectorize within the computation of $(F_0+F_1)(G_0+G_1)$. NEON also supports a multiply-accumulate instruction, allowing us to eliminate many addition instructions.

5.2. Carry vectorization. At the end of the Karatsuba computation, reduction modulo p produces a product of the form $\sum_{i=0}^{7} m_i 2^{26i} + 2^{207} \sum_{i=0}^{7} m_{i+8} 2^{26i}$. We then use a sequence of carries to bring each limb down to 26 (or in some cases 25) bits. We vectorize between a carry $m_0 \to m_1$ and a carry $m_8 \to m_9$, between a carry $m_1 \to m_2$ and a carry $m_9 \to m_{10}$, etc.

Each carry has very high latency, so we perform four carry chains in parallel. Specifically, we vectorize between a carry $m_0 \to m_1$ and a carry $m_8 \to m_9$, and in parallel vectorize between a carry $m_4 \to m_5$ and a carry $m_{12} \to m_{13}$; we then vectorize between a carry $m_1 \to m_2$ and a carry $m_9 \to m_{10}$, and in parallel vectorize between a carry $m_5 \to m_6$ and a carry $m_{13} \to m_{14}$; and so on. This hides almost all latency.

5.3. Performance. See Section 1.4 for our Cortex-A8 performance results.

References

[1] Josh Benaloh (editor), *Topics in cryptology — CT-RSA 2014 — The cryptographer's track at the RSA conference 2014, San Francisco, CA, USA, February 25–28, 2014, proceedings*, Lecture Notes in Computer Science, 8366, Springer, 2014. ISBN 978-3-319-04851-2. See [19].

[2] Daniel J. Bernstein, *Curve25519: new Diffie-Hellman speed records*, in PKC 2006 [**41**] (2006), 207–228. URL: http://cr.yp.to/papers.html#curve25519. Citations in this document: §1.

[3] Daniel J. Bernstein, *Batch binary Edwards*, in Crypto 2009 [**23**] (2009), 317–336. URL: http://cr.yp.to/papers.html#bbe. Citations in this document: §4.2.

[4] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, Peter Schwabe, *Kummer strikes back: new DH speed records* (2014). URL: https://eprint.iacr.org/2014/134. Citations in this document: §1, §1, §1, §1.1, §1.1, §1.3.

[5] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, Bo-Yin Yang, *High-speed high-security signatures*, in CHES 2011 [**38**] (2011). URL: http://eprint.iacr.org/2011/368. Citations in this document: §3.2.

[6] Daniel J. Bernstein, Tanja Lange, *Faster addition and doubling on elliptic curves*, in Asiacrypt 2007 [**30**] (2007), 29–50. URL: http://eprint.iacr.org/2007/286. Citations in this document: §2.2.

[7] Daniel J. Bernstein, Tanja Lange, *Security dangers of the NIST curves* (2013). URL: http://cr.yp.to/talks/2013.09.16/slides-djb-20130916-a4.pdf. Citations in this document: §1.

[8] Daniel J. Bernstein, Tanja Lange (editors), *eBACS: ECRYPT Benchmarking of Cryptographic Systems*, accessed 13 June 2014 (2014). URL: http://bench.cr.yp.to. Citations in this document: §1.3.

[9] Daniel J. Bernstein, Tanja Lange (editors), *Explicit Formulas Database*, accessed 13 June 2014 (2014). URL: http://hyperelliptic.org/EFD. Citations in this document: §3.1, §A.

[10] Daniel J. Bernstein, Tanja Lange, *SafeCurves: choosing safe curves for elliptic-curve cryptography*, accessed 13 June 2014 (2014). URL: http://safecurves.cr.yp.to. Citations in this document: §2, §2.1.

[11] Daniel J. Bernstein, Peter Schwabe, *NEON crypto*, in CHES 2012 [**39**] (2012), 320–339. URL: http://cr.yp.to/papers.html#neoncrypto. Citations in this document: §1, §1.1, §1.1, §1.3.

[12] Guido Bertoni, Jean-Sébastien Coron (editors), *Cryptographic hardware and embedded systems — CHES 2013 — 15th international workshop, Santa Barbara, CA, USA, August 20–23, 2013, proceedings*, Lecture Notes in Computer Science, 8086, Springer, 2013. ISBN 978-3-642-40348-4. See [14].

[13] Joppe W. Bos, Craig Costello, Huseyin Hisil, Kristin Lauter, *Fast cryptography in genus 2*, in Eurocrypt 2013 [**28**] (2013), 194–210. URL: http://eprint.iacr.org/2012/670. Citations in this document: §1.1.

[14] Joppe W. Bos, Craig Costello, Huseyin Hisil, Kristin Lauter, *High-performance scalar multiplication using 8-dimensional GLV/GLS decomposition*, in CHES 2013 [**12**] (2013), 331–348. URL: http://eprint.iacr.org/2013/146. Citations in this document: §1.3.

[15] Joppe W. Bos, Peter L. Montgomery, Daniel Shumow, Gregory M. Zaverucha, *Montgomery multiplication using vector instructions*, in SAC 2013 [**31**] (2014), 471–489. URL: http://eprint.iacr.org/2013/519. Citations in this document: §1.1.

[16] Craig Costello, Huseyin Hisil, Benjamin Smith, *Faster compact Diffie–Hellman: endomorphiimery, Daniel Shumow, averucha,*

[24] Mike Hamburg, *Fast and compact elliptic-curve cryptography* (2012). URL: http://eprint.iacr.org/2012/309. Citations in this document: §1.1.

[25] Mike Hamburg, *New Ed448-Goldilocks release* (2014). URL: https://moderncrypto.org/mail-archive/curves/2014/000101.html. Citations in this document: §1.4.

[26] Hüseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, Ed Dawson, *Twisted Edwards curves revisited*, in Asiacrypt 2008 [**37**] (2008), 326–343. URL: http://eprint.iacr.org/2008/522. Citations in this document: §3.1.

[27] Institute of Electrical and Electronics Engineers, *IEEE 1363-2000: Standard specifications for public key cryptography*, Preliminary draft at (2000). URL: http://grouper.ieee.org/groups/1363/P1363/draft.html. Citations in this document: §2.

[28] Thomas Johansson, Phong Q. Nguyen (editors), *Advances in cryptology — EUROCRYPT 2013, 32nd annual international conference on the theory and applications of cryptographic techniques, Athens, Greece, May 26–30, 2013, proceedings*, Lecture Notes in Computer Science, 7881, Springer, 2013. ISBN 978-3-642-38347-2. See [13].

[29] Anatoly A. Karatsuba, Y. Ofman, *Multiplication of multidigit numbers on automata*, Soviet Physics Doklady **7** (1963), 595–596. ISSN 0038-5689. Citations in this document: §1.1, §4.2.

[30] Kaoru Kurosawa (editor), *Advances in cryptology — ASIACRYPT 2007, 13th international conference on the theory and application of cryptology and information security, Kuching, Malaysia, December 2–6, 2007, proceedings*, Lecture Notes in Computer Science, 4833, Springer, 2007. ISBN 978-3-540-76899-9. See [6].

[31] Tanja Lange, Kristin Lauter, Petr Lisonek (editors), *Selected areas in cryptography — SAC 2013 — 20th international conference, Burnaby, BC, Canada, August 14–16, 2013, revised selected papers*, Lecture Notes in Computer Science, 8282, Springer, 2014. ISBN 978-3-662-43413-0. See [15].

[32] Patrick Longa, Francesco Sica, *Four-dimensional Gallant–Lambert–Vanstone scalar multiplication*, in Asiacrypt 2012 [**40**] (2012), 718–739. URL: http://eprint.iacr.org/2011/608. Citations in this document: §1.1.

[33] Peter L. Montgomery, *Speeding the Pollard and elliptic curve methods of factorization*, Mathematics of Computation **48** (1987), 243–264. ISSN 0025-5718. MR 88e:11130. URL: http://links.jstor.org/sici?sici=0025-5718(198701)48:177<243:STPAEC>2.0.CO;2-3. Citations in this document: §2.2.

[34] National Institute for Standards and Technology, *Digital signature standard. Federal Information Processing Standards Publication 186-2* (2000). URL: http://csrc.nist.gov/publications/fips/archive/fips186-2/fips186-2.pdf. Citations in this document: §1.2.

[35] National Security Agency, *Suite B Cryptography / Cryptographic Interoperability* (2009). URL: http://www.nsa.gov/ia/programs/suiteb_cryptography/. Citations in this document: §2.1.

[36] Phong Q. Nguyen, Elisabeth Oswald (editors), *Advances in cryptology — EUROCRYPT 2014 — 33rd annual international conference on the theory and applications of cryptographic techniques, Copenhagen, Denmark, May 11–15, 2014, proceedings*, Lecture Notes in Computer Science, 8441, Springer, 2014. ISBN 978-3-642-55219-9. See [16].

[37] Josef Pieprzyk (editor), *Advances in cryptology — ASIACRYPT 2008, 14th international conference on the theory and application of cryptology and information security, Melbourne, Australia, December 7–11, 2008*, Lecture Notes in Computer Science, 5350, 2008. ISBN 978-3-540-89254-0. See [26].

[38] Bart Preneel, Tsuyoshi Takagi (editors), *Cryptographic hardware and embedded systems — CHES 2011, 13th international workshop, Nara, Japan, September 28–October 1, 2011, proceedings*, Lecture Notes in Computer Science, 6917, Springer, 2011. ISBN 978-3-642-23950-2. See [5].

[39] Emmanuel Prouff, Patrick Schaumont (editors), *Cryptographic hardware and embedded systems — CHES 2012 — 14th international workshop, Leuven, Belgium, September 9–12, 2012, proceedings*, Lecture Notes in Computer Science, 7428, Springer, 2012. ISBN 978-3-642-33026-1. See [11].

[40] Xiaoyun Wang, Kazue Sako (editors), *Advances in cryptology — ASIACRYPT 2012, 18th international conference on the theory and application of cryptology and information security, Beijing, China, December 2–6, 2012, proceedings*, Lecture Notes in Computer Science, Springer, 2012. ISBN 978-3-642-34960-7. See [32].

[41] Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, Tal Malkin (editors), *Public key cryptography — 9th international conference on theory and practice in public-key cryptography, New York, NY, USA, April 24–26, 2006, proceedings*, Lecture Notes in Computer Science, 3958, Springer, 2006. ISBN 978-3-540-33851-2. See [2].

A Point arithmetic formulas

This appendix presents the formulas that we use for doubling and addition of curve points. Most of these formulas are taken from the EFD [9]. To simplify the cost statements we count only field multiplications and squarings, not additions and subtractions.

A.1. Formulas for doubling. We use three different formulas for point doubling. The slowest formulas are the following:

Input: X_1, Y_1, Z_1
Output: X_3, Y_3, Z_3, T_3
Cost: $4\mathbf{M} + 4\mathbf{S}$

$$\begin{aligned} A &= X_1^2, & G &= A + B, & X_3 &= EF, \\ B &= Y_1^2, & F &= G - C, & Y_3 &= GH, \\ C &= 2Z_1^2, & H &= A - B, & Z_3 &= FG, \\ E &= (X_1 + Y_1)^2 - A - B, & & & T_3 &= EH. \end{aligned}$$

We use these formulas once in each five-doubling window, specifically for the last doubling before point addition. Each of the other four doublings costs just $3\mathbf{M} + 4\mathbf{S}$: we save $1\mathbf{M}$ by skipping the computation of T_3.

For the first doubling in the precomputation we use the following faster formulas.

Input: X_1, Y_1, T_1 where $Z_1 = 1$
Output: X_3, Y_3, Z_3, T_3
Cost: $3\mathbf{M} + 3\mathbf{S}$

$$\begin{aligned} A &= X_1^2, & G &= A + B, & X_3 &= EF, \\ B &= Y_1^2, & F &= G - 2, & Y_3 &= GH, \\ E &= 2T_1, & H &= A - B, & Z_3 &= G^2 - 2G, \\ & & & & T_3 &= EH. \end{aligned}$$

A.2. Formulas for addition. All additions in the precomputation use the following formulas. These formulas save $1\mathbf{M}$ using $Z_2 = 1$.

Input: $X_1, Y_1, Z_1, T_1, X_2, Y_2, dT_2$ where $Z_2 = 1$
Output: X_3, Y_3, Z_3, T_3
Cost: $8\mathbf{M}$

$$\begin{aligned} A &= X_1X_2, & F &= Z_1 - C, & X_3 &= EF, \\ B &= Y_1Y_2, & G &= Z_1 + C, & Y_3 &= GH, \\ C &= T_1dT_2, & H &= B - A, & Z_3 &= FG, \\ E &= (X_1 + Y_1)(X_2 + Y_2) - A - B, & & & T_3 &= EH. \end{aligned}$$

All additions in the main computation use the following formulas. These formulas save $1\mathbf{M}$ by skipping the computation of T_3; the next operation is doubling, which does not use T.

Input: $X_1, Y_1, Z_1, T_1, X_2, Y_2, Z_2, dT_2$
Output: X_3, Y_3, Z_3
Cost: $8\mathbf{M}$

$$\begin{aligned} A &= X_1X_2, & E &= (X_1 + Y_1)(X_2 + Y_2) - A - B, & & \\ B &= Y_1Y_2, & F &= D - C, & X_3 &= EF, \\ C &= T_1dT_2, & G &= D + C, & Y_3 &= GH, \\ D &= Z_1Z_2, & H &= B - A, & Z_3 &= FG. \end{aligned}$$