Curve25519: new Diffie-Hellman speed records

Daniel J. Bernstein *

djb@cr.yp.to

Abstract. This paper explains the design and implementation of a high-security elliptic-curve-Diffie-Hellman function achieving record-setting speeds: e.g., 832457 Pentium III cycles (with several side benefits: free key compression, free key validation, and state-of-the-art timing-attack protection), more than twice as fast as other authors' results at the same conjectured security level (with or without the side benefits).

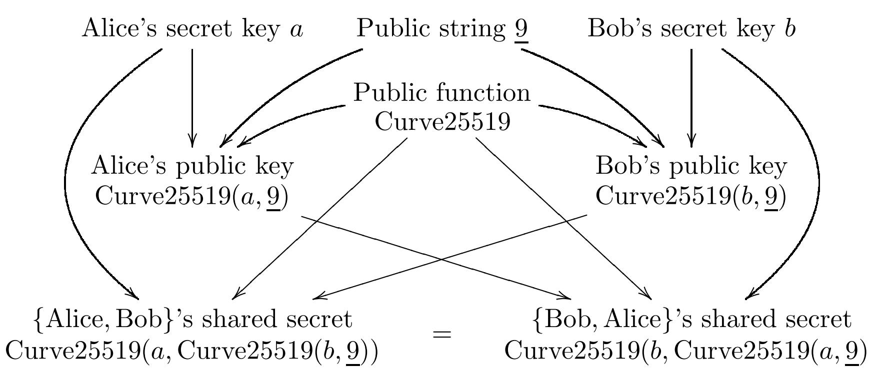
Keywords: Diffie-Hellman, elliptic curves, point multiplication, new curve, new software, high conjectured security, high speed, constant time, short keys

1 Introduction

This paper introduces and analyzes Curve25519, a state-of-the-art elliptic-curve-Diffie-Hellman function suitable for a wide variety of cryptographic applications. This paper uses Curve25519 to obtain new speed records for high-security Diffie-Hellman computations.

Here is the high-level view of Curve25519: Each Curve25519 user has a 32-byte secret key and a 32-byte public key. Each set of two Curve25519 users has a 32-byte shared secret used to authenticate and encrypt messages between the two users.

Medium-level view: The following picture shows the data flow from secret keys through public keys to a shared secret.



* Thanks to Tanja Lange for her extensive comments. Date of this document: 2006.02.09. Permanent ID of this document: 4230efdfa673480fc079449d90f322c0. This document is final and may be freely cited.

A hash of the shared secret Curve25519(a, Curve25519(b, $\underline{9}$)) is used as the key for a secret-key authentication system (to authenticate messages), or as the key for a secret-key authenticated-encryption system (to simultaneously encrypt and authenticate messages).

Low-level view: The Curve25519 function is $\mathbf{F}_p$-restricted x-coordinate scalar multiplication on $E(\mathbf{F}_{p^2})$, where p is the prime number $2^{255} - 19$ and E is the elliptic curve $y^2 = x^3 + 486662x^2 + x$. See Section 2 for further details.

Conjectured security level. Breaking the Curve25519 function—for example, computing the shared secret from the two public keys—is conjectured to be extremely difficult. Every known attack is more expensive than performing a brute-force search on a typical 128-bit secret-key cipher.

The general problem of elliptic-curve discrete logarithms has been attacked for two decades with very little success. Generic discrete-logarithm algorithms break prime groups that are not sufficiently large, but the prime group used in this paper has size above 2^{252}. Elliptic curves with certain special algebraic structures can be broken much more quickly by non-generic algorithms, but $E(\mathbf{F}_{p^2})$ does not have those structures. See Section 3 of this paper for more detailed comments on the security of the Curve25519 function.

If large quantum computers are built then they will break Curve25519 and all other short-key discrete-logarithm systems. See [56] for details of a general elliptic-curve-discrete-logarithm algorithm. The ramifications of this observation are orthogonal to the topic of this paper and are not discussed further.

Efficiency. My public-domain Curve25519 software provides several efficiency features, thanks in large part to the choice of the Curve25519 function:

- **Extremely high speed.** My software computes Curve25519 in just 832457 cycles on a Pentium III, 957904 cycles on a Pentium 4, 640838 cycles on a Pentium M, and 624786 cycles on an Athlon. Each of these numbers is a new speed record for high-security Diffie-Hellman functions. I am working on implementations for the UltraSPARC, PowerPC, etc.; I expect to end up with similar cycle counts.
- **No time variability.** Most speed reports in the cryptographic literature are for software without any protection against timing attacks. See [12], [51], and [50] for some successful attacks. Adding protection can dramatically slow down the computation. In contrast, my Curve25519 software is already immune to timing attacks, including hyperthreading attacks and other cache-timing attacks. It avoids all input-dependent branches, all input-dependent array indices, and other instructions with input-dependent timings.
- **Short secret keys.** The Curve25519 secret key is only 32 bytes. This is typical for high-security Diffie-Hellman functions.
- **Short public keys.** The Curve25519 public key is only 32 bytes. Typical elliptic-curve-Diffie-Hellman functions use 64-byte public keys; those keys can be compressed to half size, as suggested by Miller in [46], but the time for decompression is quite noticeable and usually not reported.
- **Free key validation.** Typical elliptic-curve-Diffie-Hellman functions can be broken if users do not validate public keys; see, e.g., [14, Section 4.1] and [3].

The time for key validation is quite noticeable and usually not reported. In contrast, every 32-byte string is accepted as a Curve25519 public key.
- **Short code.** My software is very small. The compiled code, including all necessary tables, is around 16 kilobytes on each CPU, and can easily fit alongside other networking tools in the CPU's instruction cache.

The new speed records are the highlight of this paper. Sections 4 and 5 explain the computation of Curve25519 in detail from the bottom up.

One can improve speed by choosing functions at lower security levels; for example, dropping from 255 bits down to 160 bits. But—as discussed in Section 3—I can easily imagine an attacker with the resources to break a 160-bit elliptic curve in under a year. Users should not expose themselves to this risk; they should instead move up to the comfortable security level of Curve25519.

Of course, when users exchange large volumes of data, their bottleneck is a secret-key cryptosystem, and the Curve25519 speed no longer matters.

Comparison to previous work. There is an extensive literature analyzing the speed of various implementations of various Diffie-Hellman functions at various conjectured security levels.

In particular, there have been some reports of high-security elliptic-curve scalar-multiplication speeds: [17, Table 8] reports 1920000 cycles on a 400 MHz Pentium II for field size $2^{256}-2^{224}+2^{192}+2^{96}-1$; [33, Table 7] reports 1740000 cycles on a 400 MHz Pentium II for field size 2^{283} using a subfield curve; [4, Table 4] reports 3086000 cycles on a 1000 MHz Athlon for a random 256-bit prime field. At a lower security level: [7, Table 3] reports 2650000 cycles on a 233 MHz Pentium MMX for field size $(2^{31}-1)^6$; [58, Table 4] reports 4500000 cycles on a 166 MHz Pentium Pro for field size $(2^{31}-19)^6$; [26, Table 6] reports 1720000 cycles on an 800 MHz Pentium III for field size 2^{233}.

The Curve25519 timings are more than twice as fast as the above reports. The comparison is actually even more lopsided than this, because the Curve25519 timings include free key compression, free key validation, and state-of-the-art timing-attack protection, while the above reports do not.

I have previously reported preliminary implementation work achieving about half of this speedup using a standard NIST curve. The other half of the speedup relies on switching to a better-designed curve. This paper covers both halves of the speedup.

At a lower level, designing and implementing an elliptic-curve-Diffie-Hellman function means making many choices that affect speed. Making a few bad choices can destroy performance. In the design and implementation of Curve25519 I have tried to globally optimize the entire sequence of choices:

- Use large characteristic, not characteristic 2.
- Use curve shape $y^2 = x^3 + Ax^2 + x$, with $(A-2)/4$ small, rather than $y^2 = x^3 - 3x + a_6$.
- Use x as a public key, not (x, y).
- Use a secure curve that also has a secure twist, rather than taking extra time to prohibit keys on the twist.

- Use x/z inside scalar multiplication, not $(x/z, y/z)$ or $(x/z^2, y/z^3)$.
- Convert variable array indexing into arithmetic.
- Use a fixed position for the leading 1 in the secret key.
- Multiply the secret key by a small power of 2 to account for cofactors in the curve group and the twist group.
- Use a prime field, not an extension field.
- Use a prime extremely close to 2^b for some b.
- Use radix $2^{b/w}$ for some w, even if b/w is not an integer.
- Allow coefficients slightly larger than the radix, rather than reducing each coefficient as soon as possible.
- Put coefficients into floating-point registers, not integer registers. Choose w accordingly.

See Sections 4 and 5 for details and credits. Beware that these choices interact across many levels of design and implementation: for example, there are other curve shapes and prime shapes for which $(x/z^2, y/z^3)$ is better than x/z. This type of interaction makes the optimal sequence of choices difficult to identify even when all possible choices are known.

2 Specification

This section defines the Curve25519 function. Readers not familiar with rings, fields, and elliptic curves should consult Appendix A for definitions and for a proof of Theorem 2.1.

Theorem 2.1. *Let p be a prime number with $p \geq 5$. Let A be an integer such that $A^2 - 4$ is not a square modulo p. Define E as the elliptic curve $y^2 = x^3 + Ax^2 + x$ over the field $\mathbf{F}_p$. Define $X_0 : E(\mathbf{F}_{p^2}) \to \mathbf{F}_{p^2}$ as follows: $X_0(\infty) = 0$; $X_0(x, y) = x$. Let n be an integer. Let q be an element of $\mathbf{F}_p$. Then there exists a unique $s \in \mathbf{F}_p$ such that $X_0(nQ) = s$ for all $Q \in E(\mathbf{F}_{p^2})$ such that $X_0(Q) = q$.*

In particular, define p as the prime $2^{255} - 19$. Define $\mathbf{F}_p$ as the prime field $\mathbf{Z}/p = \mathbf{Z}/(2^{255} - 19)$. Note that 2 is not a square in $\mathbf{F}_p$; define $\mathbf{F}_{p^2}$ as the field $(\mathbf{Z}/(2^{255} - 19))[\sqrt{2}]$. Define $A = 486662$. Note that $486662^2 - 4$ is not a square in $\mathbf{F}_p$. Define E as the elliptic curve $y^2 = x^3 + Ax^2 + x$ over $\mathbf{F}_p$. Define a function $X_0 : E(\mathbf{F}_{p^2}) \to \mathbf{F}_{p^2}$ as follows: $X_0(\infty) = 0$; $X_0(x, y) = x$. Define a function $X : E(\mathbf{F}_{p^2}) \to \{\infty\} \cup \mathbf{F}_{p^2}$ as follows: $X(\infty) = \infty$; $X(x, y) = x$.

At this point I could say that, given $n \in 2^{254} + 8\{0, 1, 2, 3, \ldots, 2^{251} - 1\}$ and $q \in \mathbf{F}_p$, the Curve25519 function produces s in Theorem 2.1. However, to match cryptographic reality and to catch the types of design error explained by Menezes in [45], I will instead define the inputs and outputs of Curve25519 as sequences of bytes.

The set of **bytes** is, by definition, $\{0, 1, \ldots, 255\}$. The encoding of a byte as a sequence of bits is not relevant to this document. Write $s \mapsto \underline{s}$ for the standard little-endian bijection from $\{0, 1, \ldots, 2^{256} - 1\}$ to the set $\{0, 1, \ldots, 255\}^{32}$ of 32-byte strings: in other words, for each integer $s \in \{0, 1, \ldots, 2^{256} - 1\}$, define $\underline{s} = (s \bmod 256, \lfloor s/256 \rfloor \bmod 256, \ldots, \lfloor s/256^{31} \rfloor \bmod 256)$.

The set of Curve25519 **public keys** is, by definition, $\{0, 1, \ldots, 255\}^{32}$; in other words, $\{\underline{q} : q \in \{0, 1, \ldots, 2^{256} - 1\}\}$. The set of Curve25519 **secret keys** is, by definition, $\{0, 8, 16, 24, \ldots, 248\} \times \{0, 1, \ldots, 255\}^{30} \times \{64, 65, 66, \ldots, 127\}$; in other words, $\{\underline{n} : n \in 2^{254} + 8\{0, 1, 2, 3, \ldots, 2^{251} - 1\}\}$.

Now Curve25519 : {Curve25519 secret keys} × {Curve25519 public keys} → {Curve25519 public keys} is defined as follows. Fix $q \in \{0, 1, \ldots, 2^{256} - 1\}$ and $n \in 2^{254} + 8\{0, 1, 2, 3, \ldots, 2^{251} - 1\}$. By Theorem 2.1, there is a unique integer $s \in \{0, 1, 2, \ldots, 2^{255} - 20\}$ with the following property: $s = X_0(nQ)$ for all $Q \in E(\mathbf{F}_{p^2})$ such that $X_0(Q) = q \bmod 2^{255} - 19$. Finally, Curve25519$(\underline{n}, \underline{q})$ is defined as $\underline{s}$. Note that Curve25519 is not surjective: in particular, its final output bit is always 0 and need not be transmitted.

3 Security

This section discusses attacks on Curve25519. The bottom line is that all known attacks are extremely expensive.

Responsibilities of the user. The legitimate users are assumed to generate independent uniform random secret keys. A user can, for example, generate 32 uniform random bytes, clear bits $0, 1, 2$ of the first byte, clear bit 7 of the last byte, and set bit 6 of the last byte.

Large deviations from uniformity can eliminate all security. For example, if the first 16 bytes of the secret key $\underline{n}$ were instead chosen as a public constant, then a moderately large computation would deduce the remaining bytes of $\underline{n}$ from the public key Curve25519$(\underline{n}, \underline{9})$. This is not Curve25519's fault; the user is responsible for putting enough randomness into keys.

Legitimate users are also assumed to keep their secret keys secret. This means that a secret key $\underline{n}$ is not used except to compute the public key Curve25519$(\underline{n}, \underline{9})$ and to compute the shared-secret hash $H($Curve25519$(\underline{n}, \underline{q}))$ given $\underline{q}$.

Users are *not* assumed to throw $\underline{n}$ away after a single $\underline{q}$. Diffie-Hellman secret keys can—and, for efficiency, should—be reused with many public keys, as in [23, Section 3]. Each user's secret key $\underline{n}$ is combined with many other users' public keys $\underline{q_1}, \underline{q_2}, \underline{q_3}, \ldots$, producing shared-secret hashes $H($Curve25519$(\underline{n}, \underline{q_1}))$, $H($Curve25519$(\underline{n}, \underline{q_2}))$, $H($Curve25519$(\underline{n}, \underline{q_3}))$, ...

Choice of key-derivation function. There are no theorems guaranteeing the safety of any particular key-derivation function H with, e.g., 512-bit output. Some silly choices of H are breakable. As an extreme example, if H outputs just 64 bits followed by all zeros, then an attacker can perform a brute-force search for those 64 bits.

On the other hand, from the perspective of a secret-key cryptographer, it seems very easy to design a safe function H. A small amount of mixing, far less than necessary to make a safe secret-key cipher, stops all known attacks.

For concreteness I will define $H(x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7)$ as the 64-byte string Salsa20$(c_0, x_0, 0, x_1, x_2, c_1, x_3, 0, 0, x_4, c_2, x_5, x_6, 0, x_7, c_3)$. Here Salsa20 is the function defined in [13, Section 8]; (c_0, c_1, c_2, c_3) is "Curve25519output" in ASCII; and each x_i has 4 bytes.

If fewer than 64 bytes are needed then the Salsa20 output can simply be truncated. If more than 64 bytes are needed then Salsa20 can be invoked again with $(c_0, x_0, 1, x_1, \ldots)$ to produce another 64 bytes.

Powers of the attacker. An attacker sees public keys $\underline{q_1} = \text{Curve25519}(\underline{n_1}, \underline{9})$, $\underline{q_2} = \text{Curve25519}(\underline{n_2}, \underline{9})$, $\ldots$ generated from the legitimate users' independent uniform random secret keys $\underline{n_1}, \underline{n_2}, \ldots$.

The attacker also sees messages protected by a secret-key cryptosystem C where the keys for C are the shared-secret hashes $H(\text{Curve25519}(\underline{n_i}, \underline{q_j})) = H(\text{Curve25519}(\underline{n_j}, \underline{q_i}))$ for various sets $\{i, j\}$. The attacker's goal is to decrypt or forge these messages.

The attacker can also compute a public key $q' \notin \{\underline{q_1}, \underline{q_2}, \ldots\}$ and—by using q' in the Diffie-Hellman protocol—see messages protected by C where the keys for C are $H(\text{Curve25519}(\underline{n_1}, q'))$, $H(\text{Curve25519}(\underline{n_2}, q'))$, $\ldots$. This would be pointless if the attacker generated q' in the normal way, but the attacker is not required to generate q' in the normal way; legitimate users are *not* assumed to check that q' was generated from a secret key, let alone a secret key known to the attacker. The attacker might take $q' = \underline{1}$, for example, or $q' = \underline{q_1} \oplus \underline{1}$. The attacker can adaptively generate many public keys q'.

Of course, security depends on the choice of secret-key cryptosystem C. One could make a poor choice of C, allowing messages to be decrypted or forged without any weakness in Curve25519. But standard choices of C are conjectured to be safe. Further discussion of the choice of C is outside the scope of this document.

Simplified attack notions. There are many papers using simpler models of Diffie-Hellman attackers, and proving theorems of the form "a fast attack in complicated-security-model implies a fast attack in simplified-security-model." The reader might wonder why I am not using one of these simplified notions.

Example: Bentahar in [10], improving an algorithm by Muzereau, Smart, and Vercauteren in [48] based on an idea by Maurer in [44], showed that one can evaluate discrete logarithms on typical elliptic curves using roughly 2^{13} calls to a reliable oracle for the function $(mQ, nQ) \mapsto mnQ$. Bentahar then repeated the standard conjecture that computing discrete logarithms on a typical 256-bit elliptic curve costs at least 2^{128} (never mind the question of exactly what "cost" means), and deduced the conjecture that computing $(mQ, nQ) \mapsto mnQ$ costs at least 2^{115}. Why, then, should one make a conjecture regarding the difficulty of computing $(mQ, nQ) \mapsto mnQ$, rather than a simplified conjecture regarding the difficulty of computing discrete logarithms?

Answer: A standard conjecture says that computing $(mQ, nQ) \mapsto mnQ$ costs at least 2^{128}. This conjecture is quantitatively stronger than anything that can be obtained by applying Bentahar's theorem to a simplified conjecture.

Similar comments apply to other theorems of this type; see, e.g., [39, Section 3.2]. Often the theorems are so weak that they say nothing about any real-world system. To focus attention on the security properties that applications actually need, I have chosen to make a complicated but strong conjecture about security, rather than a simplified but weak conjecture.

Generic discrete logarithms by the rho and kangaroo methods. The attacker can expand Curve25519($\underline{n}$, $\underline{9}$) into a point (x, y) on $E(\mathbf{F}_{p^2})$, namely the nth multiple of the base point $(9, \ldots)$. The attacker can then use Pollard's rho method or Pollard's kangaroo method to compute the discrete logarithm of this point, namely n. The main cost in either method is the cost of performing a huge number of additions of elliptic-curve points; both methods are almost perfectly parallelizable, with negligible communication costs. See [63], [55], [61], and [60].

The number of additions here is about the square root of the length of the n interval: in this case, about 2^{125}. The computation can finish after far fewer additions, but the success chance is at most (and conjecturally at least) about $a^2/2^{251}$ after a additions.

How many elliptic-curve additions can an attacker perform? The traditional estimate is roughly 2^{70} elliptic-curve additions: a modern CPU costs about 2^6 dollars; a modern CPU cycle is about 2^{-31} seconds; each elliptic-curve addition in the rho or kangaroo method costs about 2^{10} CPU cycles for roughly 2^2 field multiplications that each cost 2^8 cycles; the attacker is willing to spend a year, i.e., 2^{25} seconds; the attacker can afford to spend 2^{30} dollars.

I don't agree with the traditional estimate. I agree that modern circuitry takes about 2^{-21} seconds for a single rho/kangaroo step; but it is a huge error to assume that this circuitry costs as much as 2^6 dollars. One can fit many parallel rho/kangaroo circuits into the same amount of circuitry as a modern CPU. A reasonable estimate for "many" is 2^{10}; see [28] for a fairly detailed chip design, and [28, Section 5.2] for the estimate. By switching to this chip, the attacker can perform roughly 2^{80} elliptic-curve additions. The attacker has an excellent chance of computing a 160-bit discrete logarithm, but only about a 2^{-90} chance of computing a 251-bit discrete logarithm.

Of course, one must adjust these estimates as chip technology improves. It is not enough to account for increases in cycle speed and for decreases in chip cost; one must also account for increases in chip size. However, the Curve25519 security level will remain comfortable for the foreseeable future.

Batch discrete logarithms. Silverman and Stapleton observed, and Kuhn and Struik proved in [41, Section 4] assuming standard conjectures, that the rho method can compute u discrete logarithms using about $\sqrt{u}$ times as much effort as computing a single discrete logarithm.

For example, given public keys Curve25519($\underline{n_1}$, $\underline{9}$), …, Curve25519($\underline{n_u}$, $\underline{9}$), the attacker can discover most of the secret keys $\underline{n_1}, \ldots, \underline{n_u}$ using only about $2^{125}\sqrt{u}$ additions, i.e., about $2^{125}/\sqrt{u}$ additions per key.

This does not mean, however, that one of the keys will be found within the first $2^{125}/\sqrt{u}$ additions. On the contrary: the attacker is likely to wait for 2^{125} additions before finding the first key, then another $2^{125}(\sqrt{2}-1)$ additions before finding the second key, etc. Curve25519 is at a comfortable security level where finding the first key is, conjecturally, far out of reach, so the reduced cost of finding subsequent keys is not a threat. The attacker can perform only $2^{125}\epsilon$

additions for small ϵ, so the attacker's chance of success—of finding *any* keys—is only about ϵ^2.

Generic discrete logarithms are often claimed to be about as difficult as brute-force search for a half-size key. But brute-force search computes a batch of u keys with about the *same* effort as computing a single key. Furthermore, brute-force search has probability roughly $u\epsilon$ of finding some key after the first ϵ of the computation, whereas discrete logarithms have only an ϵ^2 chance. Evidently generic discrete logarithms are *more* difficult than brute-force search for a half-size key: $u\epsilon$ is much larger than ϵ^2, except in the extreme case where u and ϵ are both close to 1.

Small-subgroup attacks. If the subgroup of $E(\mathbf{F}_{p^2})$ generated by the base point $(9, \ldots)$ has non-prime order then the attacker can use the Pohlig-Hellman method to save time in computing discrete logarithms. See, e.g., [5, Section 19.3].

This attack fails against Curve25519. The order of the base point is a prime, namely $2^{252} + 27742317777372353535851937790883648493$.

An active attacker has more options. Say there is a point $(x, y) \in E(\mathbf{F}_{p^2})$ of order b, with $x \in \mathbf{F}_p$ and with b not very large. The attacker can issue a public key $\underline{x}$. The legitimate user will then authenticate and encrypt data under $H(\mathrm{Curve25519}(\underline{n}, \underline{x})) = H(X_0(n(x, y))) = H(X_0((n \bmod b)(x, y)))$; the attacker can compare the results to all possibilities for $n \bmod b$, presumably determining $n \bmod b$.

The active attack also fails against Curve25519. The group $\{\infty\} \cup (E(\mathbf{F}_{p^2}) \cap (\mathbf{F}_p \times \mathbf{F}_p))$ has size $8p_1$, where $p_1 = 2^{252} + \cdots$ is the prime number displayed above. The "twist" group $\{\infty\} \cup (E(\mathbf{F}_{p^2}) \cap (\mathbf{F}_p \times \sqrt{2}\mathbf{F}_p))$ has size $2(p+1) - 8p_1 = 4p_2$, where p_2 is the prime $2^{253} - 55484635554744707071703875581767296995$. Consequently, the only possibilities for b below 2^{252} are $1, 2, 4, 8$. Secret keys $\underline{n}$ by definition have $n \bmod 8 = 0$ and thus $n \bmod b = 0$.

History: Lim and Lee in [42] pointed out active attacks on Diffie-Hellman in the group $\mathbf{F}_p^*$. They recommended in [42, Section 4] that, rather than taking the time to test that public keys are in a particular subgroup of prime order q, one choose a prime p such that "each prime factor of $(p-1)/2q$ is larger than q." Biehl, Meyer, and Müller in [14, Section 4.1] pointed out analogous attacks on elliptic curves when public keys are represented as pairs (x, y); they did not propose any workaround other than testing keys. In a November 2001 `sci.crypt` posting I wrote "You can happily skip both the y transmission and the square root. In fact, if both the curve and its twist have nearly prime order, then you can even skip square testing."

Other attacks. The kangaroo method actually searches simultaneously for $n/8$ and $p_1 - n/8$ in an interval. The range of $n/8$ is $\{2^{251}, \ldots, 2^{252} - 1\}$, so either $n/8$ or $p_1 - n/8$ is in the range $\{(p_1 + 1)/2, \ldots, 2^{252} - 1\}$. However, p_1 is only marginally above 2^{252}, so this range has length only marginally below 2^{251}.

More generally, when a group G has an easily computed automorphism φ of small order b, one can apply the kangaroo method to the orbits of φ, using only about $\sqrt{\#G/b}$ steps rather than $\sqrt{\#G}$ steps. See, e.g., [5, Section 19.5.5]. But my elliptic curve has no structure of this type other than negation. In fact, it has no complex endomorphisms of small norm. To prove this, compute the trace $t = p + 1 - 8p_1$, and observe that $t^2 - 4p$ is not a small multiple of a square: it is divisible once by the prime 8312956054562778877481, for example.

My elliptic curve also resists the transfer attacks surveyed in [30, Chapter 22]. The primes p_1 and p_2 do not equal the field characteristic p. The order of p modulo p_1 is not small: in fact, it is $(p_1 - 1)/6$. The order of p modulo p_2 is not small: in fact, it is $p_2 - 1$. Weil descent simply splits $E(\mathbf{F}_{p^2})$ into the subgroup $E(\mathbf{F}_p)$, of order $8p_1$, and the twist, of order $4p_2$; there are no proper subfields of $\mathbf{F}_p$ to exploit.

4 Fast arithmetic modulo $2^{255} - 19$

This section explains one way to use common CPU instructions, specifically floating-point instructions, to quickly multiply and add in the field $\mathbf{F}_p$ where $p = 2^{255} - 19$. I will focus on the Pentium M for concreteness, but the same techniques work well for a wide variety of CPUs. This section also discusses the choice of field structure and the choice of prime.

In this section, "floating-point" is abbreviated "fp."

Representing integers modulo $2^{255} - 19$. Define R as the ring of polynomials $\sum_i u_i x^i$ where u_i is an integer multiple of $2^{\lceil 25.5i \rceil}$. One way to see that R is a ring is to observe that it is the intersection of the subrings $\mathbf{Z}[x]$ and $\overline{\mathbf{Z}}[2^{25.5}x]$ of $\overline{\mathbf{Z}}[x]$, where $\overline{\mathbf{Z}}$ is the ring of algebraic integers in $\mathbf{C}$.

Elements of R represent elements of $\mathbf{Z}/(2^{255} - 19)$: each polynomial represents its value at 1. Often a polynomial is chosen to meet two restrictions:

- The polynomial degree is small, to limit the number of coefficients that need to be multiplied as part of polynomial multiplication. Specifically, **reduced-degree** polynomials have degree at most 9.
- Each coefficient u_i is a small multiple of $2^{\lceil 25.5i \rceil}$, to limit the effort of multiplying coefficients. Specifically, **reduced-coefficient** polynomials have $u_i/2^{\lceil 25.5i \rceil} \in \left\{-2^{25}, -2^{25}+1, \ldots, -1, 0, 1, \ldots, 2^{25}-1, 2^{25}\right\}$.

To summarize: A reduced-degree reduced-coefficient polynomial is a polynomial $u_0 + u_1 x + \cdots + u_9 x^9$ with $u_0/2^0$, $u_1/2^{26}$, $u_2/2^{51}$, $u_3/2^{77}$, $u_4/2^{102}$, $u_5/2^{128}$, $u_6/2^{153}$, $u_7/2^{179}$, $u_8/2^{204}$, $u_9/2^{230}$ all in $\left\{-2^{25}, -2^{25}+1, \ldots, -1, 0, 1, \ldots, 2^{25}-1, 2^{25}\right\}$. This polynomial represents the integer $u_0 + u_1 + \cdots + u_9$.

Note that integers are not converted to a unique "smallest" representation until the end of the Curve25519 computation. Producing reduced representations is generally much faster than producing "smallest" representations.

Representing coefficients inside CPUs. The Pentium M has eight "fp registers," each of which holds a real number $2^e f$ for integers e and f with $f \in \left\{-2^{64}, \ldots, 2^{64}\right\}$ and with e in an adequate range for all the computations discussed here. My computations hold polynomial coefficients in fp registers to the extent possible, as in [11, Section 4].

The Pentium M has many more "L1-cache doublewords" that can hold $2^e f$ with f limited to the range $\{-2^{53}, \ldots, 2^{53}\}$; e.g., reduced coefficients. To perform arithmetic on numbers in L1-cache doublewords, the Pentium M must take time to copy ("load") the numbers into registers; but this is not a big problem, because these loads can be overlapped with arithmetic if they are not too frequent.

Why split 255-bit integers into ten 26-bit pieces, rather than nine 29-bit pieces or eight 32-bit pieces? Answer: The coefficients of a polynomial product do not fit into the Pentium M's fp registers if pieces are too large. The cost of handling larger coefficients outweighs the savings of handling fewer coefficients. The overall time for 29-bit pieces is sufficiently competitive to warrant further investigation, but so far I haven't been able to save time this way. I'm sure that 32-bit pieces, the most common choice in the literature, are a bad idea.

Of course, the same question must be revisited for each CPU. The Pentium 1, Pentium MMX, Pentium Pro, Pentium II, Pentium III, Pentium 4, Athlon, and Athlon XP work well with 26-bit pieces; on the Athlon 64 and Opteron, 32-bit pieces might be slightly better. On the UltraSPARC and PowerPC, fp registers use $\{-2^{53}, \ldots, 2^{53}\}$ rather than $\{-2^{64}, \ldots, 2^{64}\}$, and I recommend twelve 22-bit pieces. The UltraSPARC and PowerPC can overlap fp additions with fp multiplications, so I expect them to end up with comparable cycle counts to the Pentium M despite the larger number of pieces.

Given that there are 10 pieces, why use radix $2^{25.5}$ rather than, e.g., radix 2^{25} or radix 2^{26}? Answer: My ring R contains $2^{255}x^{10} - 19$, which represents 0 in $\mathbf{Z}/(2^{255} - 19)$. I will reduce polynomial products modulo $2^{255}x^{10} - 19$ to eliminate the coefficients of x^{10}, x^{11}, etc. With radix 2^{25}, the coefficient of x^{10} could not be eliminated. With radix 2^{26}, coefficients would have to be multiplied by $2^5 \cdot 19$ rather than just 19, and the results would not fit into an fp register.

Using floating-point operations. The Pentium M has circuits for three fast operations on numbers stored in fp registers: sum, difference, and product. These are exact operations if the results fit into the 64-bit fp precision; otherwise the results are rounded to the nearest fp numbers.

The Pentium M can perform, at best, one fp operation per cycle. About 92% of the cycles in my Curve25519 computation (589825 out of 640838) are occupied by fp operations. One can understand the cycle counts fairly well by simply counting the fp operations. Similar comments apply to other CPUs, although the details depend on the CPU.

Warning: Writing an fp program in the C programming language, and feeding the result to a C compiler, often produces machine language that takes 3 or more Pentium M cycles for each fp operation. Further discussion of this phenomenon is outside the scope of this paper. My Curve25519 software is actually written in `qhasm`, a new programming language designed for high-speed computations.

Beware that a few CPUs have input-dependent fp timings. An old example is the Sun microSPARC-IIep. A newer example is the IBM PowerPC RS64 IV, which takes an extra cycle to multiply by 0. Fast constant-time computations on these CPUs need extra effort.

Adding integers modulo $2^{255} - 19$. If two integers are represented by two polynomials u and v then the sum of the two integers is represented by $u + v$. Similarly, the difference of the two integers is represented by $u - v$.

If u and v are reduced-degree reduced-coefficient polynomials then computing $u+v$ (or $u-v$) involves 10 additions (or subtractions) of fp numbers. Note that the sum is reduced-degree but usually not reduced-coefficient. In a long chain of sums one would occasionally have to take extra time to reduce the coefficients. This is never necessary in the Curve25519 computation: every sum (and difference) is used solely as input to products, as Appendix B illustrates.

Statistics: Each addition or subtraction takes 10 fp operations. There are 8 additions and subtractions, totalling 80 fp operations, in each iteration of the Curve25519 main loop. There are 2040 additions and subtractions, totalling 20400 fp operations, in the entire Curve25519 computation.

Multiplying integers modulo $2^{255} - 19$. If two integers are represented by polynomials u and v then their product is represented by the polynomial product uv. If u and v are reduced-degree reduced-coefficient polynomials, or sums of two such polynomials, then computing uv in the simplest way involves 100 fp multiplications and 81 fp additions; I am experimenting with other polynomial-multiplication algorithms and expect to end up with slightly better results. The product uv is then replaced by a reduced-degree reduced-coefficient polynomial:

- The coefficients of $x^{10}, x^{11}, \ldots, x^{18}$ in uv are eliminated by reduction modulo $2^{255}x^{10} - 19$. For example, the coefficient of x^{18} is multiplied by $19 \cdot 2^{-255}$ and added to the coefficient of x^8. Each reduction involves 1 fp multiplication and 1 fp addition.
- The "high" part of each coefficient is subtracted from that coefficient and added ("carried") to the next coefficient. The high part is, by definition, the nearest multiple of the power of 2 for the next coefficient. One carry involves 4 fp additions: 2 to identify the high part (by a rounded addition and then subtraction of a large constant), 1 to subtract, and 1 to add.

Starting from uv, I carry from x^8 to x^9, then from x^9 to x^{10}; then I eliminate coefficients of $x^{10}, x^{11}, \ldots, x^{18}$; then I carry from x^0 to x^1, from x^1 to x^2, $\ldots$, from x^7 to x^8, and once more from x^8 to x^9. Note that the coefficient of x^9 is a multiple of 2^{230}, and is between -2^{254} and 2^{254} after subtraction of its original high part, so the final carry from x^8 to x^9 produces reduced coefficients. Overall there are 18 fp operations to eliminate 9 coefficients, and 44 fp operations for 11 carries. There are many other reasonable carry sequences; on some CPUs it might be a good idea to have two parallel carry chains, decreasing latency at the expense of an extra carry.

Squaring is easier than general multiplication, because polynomial squaring is easier than general polynomial multiplication. Overall a squaring eliminates $9^2 + 9$ coefficient multiplications at the expense of 9 initial coefficient doublings; note that doubling coefficients at the beginning is slightly better than doubling products later. Multiplication by a small constant is also easier than general multiplication, because the constant is represented by a polynomial of degree 0.

Statistics: Each multiplication by a small constant takes 55 fp operations. Each squaring takes 162 fp operations. Each general multiplication takes 243 fp operations. Each iteration of the Curve25519 main loop has 1 multiplication by a small constant, using 55 fp operations; 4 squarings, using 648 fp operations; and 5 general multiplications, using 1215 fp operations; in total 10 multiplications, using 1918 fp operations. The Curve25519 computation has 255 multiplications by small constants, using 14025 fp operations; 1274 squarings, using 206388 fp operations; and 1286 general multiplications, using 312498 fp operations; in total 2815 multiplications, using 532911 fp operations.

Note that the squaring-to-multiplication floating-point-operation ratio is only $162/243 = 2/3$, far below the 0.8 ratio often used in the literature for estimating the costs of elliptic-curve operations.

Selecting integers. Consider the problem of computing $x[b]$, where $x[0], x[1]$ are integers modulo $2^{255} - 19$ and b is an input-dependent bit. Using b as an array index—without taking extra time for preloads, interrupt elimination, etc.—could allow hyperthreading attacks and other cache-timing attacks; see [12, Sections 8–15]. I instead compute $x[b]$ as $(1-b)x[0] + bx[1]$. Similarly, if I need to compute the pair $(x[b], x[1-b])$, I compute $(x[0] - b(x[0] - x[1]), x[1] + b(x[0] - x[1]))$.

Statistics: Each iteration of the Curve25519 main loop has 2 fp operations inside computing b and $1 - b$; 2 paired selections, taking 80 fp operations; and 2 more selections, taking 60 more fp operations. The total is 142 fp operations. The entire Curve25519 computation spends 36210 fp operations, about 6% of the total, on selection. Of course, these operations could be eliminated if timing attacks were not a concern.

Why this field? CPUs include fast integer-multiplication circuits (usually buried inside fp-multiplication circuits aimed at the large fp market) but not circuits for fast multiplication of polynomials modulo 2. Characteristic-2 fields allow several other speedups—see, e.g., [35, Section 3.4] and [25, Section 15.1]—but I can't see any way for them to set speed records on existing CPUs.

"Optimal extension fields," such as degree-10 extensions of prime fields of size around 2^{26}, are advertised in [7] and [6] as allowing faster multiplication and much faster inversion, perhaps so fast as to make affine-coordinate elliptic-curve computations faster than projective-coordinate elliptic-curve computations. My current assessment is that these fields have some slight advantages: there are no carry chains, so operations are easier to reorder; there are 10 reductions modulo a prime, rather than 11 carries, although one reduction is usually slightly more expensive than one carry; inversion is faster, although not fast enough to make affine coordinates worthwhile; and, most importantly, degree 9 might fit into 64-bit fp. Unfortunately, these fields have a huge disadvantage: even if they are slightly faster on some CPUs, they are much slower on other CPUs. A 255-bit integer can be split into 4 or 8 or 10 or 12 pieces to accommodate the capabilities of different processors; an "optimal extension field" is tied to a particular number of pieces.

So I selected a prime field. Prime fields also have the virtue of minimizing the number of security concerns for elliptic-curve cryptography; see, e.g., [29] and [22].

I chose my prime $2^{255}-19$ according to the following criteria: primes as close as possible to a power of 2 save time in field operations (as in, e.g, [9]), with no effect on (conjectured) security level; primes slightly below $32k$ bits, for some k, allow public keys to be easily transmitted in 32-bit words, with no serious concerns regarding wasted space; $k=8$ provides a comfortable security level. I considered the primes $2^{255}+95$, $2^{255}-19$, $2^{255}-31$, $2^{254}+79$, $2^{253}+51$, and $2^{253}+39$, and selected $2^{255}-19$ because 19 is smaller than $31, 39, 51, 79, 95$.

5 Fast Curve25519 computation

This section explains fast x-coordinate point addition on my elliptic curve $y^2 = x^3 + 486662x^2 + x$; explains fast x-coordinate scalar multiplication, i.e., fast computation of Curve25519; and compares this curve to other elliptic curves.

Recall that Section 2 defines two x-coordinate functions. One function X_0 maps ∞ to 0; the other function X maps ∞ to ∞. Curve25519 is defined using X_0, but inside the computation it is convenient to use X until the last moment.

Addition. Montgomery in [47, Section 10.3.1] published formulas to compute $X(2Q)$ given $X(Q)$, and to compute $X(Q+Q')$ given $X(Q), X(Q'), X(Q-Q')$, assuming that $Q \neq \infty$, $Q' \neq \infty$, $Q-Q' \neq \infty$, $Q+Q' \neq \infty$. It turns out that Montgomery's formulas also work for ∞, provided that $Q-Q' \notin \{\infty, (0,0)\}$, so the Curve25519 computation can avoid checking for ∞. See Appendix B of this paper.

Montgomery's formulas represent each X value as a fraction x/z, replacing divisions with multiplications. Montgomery commented that, when d is large, one can perform d divisions in $\mathbf{F}_p$ at about the same cost as $4d$ multiplications in $\mathbf{F}_p$, so dividing x by z may be a good idea when there are many separate elliptic-curve computations to perform at once; I have not implemented this option yet.

The formula for $X(2Q)$ involves 2 squarings, 1 multiplication by $121665 = (486662-2)/4$, and 2 more multiplications. The formula for $X(Q+Q')$ involves 2 squarings and 3 more multiplications when z_1 in Theorem B.2, the denominator of $X(Q-Q')$, is known to be 1; otherwise it involves 2 squarings and 4 more multiplications. The Curve25519 computation always has $z_1 = 1$.

Scalar multiplication. Montgomery suggested using his formulas to obtain $X(nQ+Q), X(nQ), X(Q)$ given $X(\lfloor n/2 \rfloor Q+Q), X(\lfloor n/2 \rfloor Q), X(Q)$: if n is even then $nQ = 2\lfloor n/2 \rfloor Q$ and $nQ+Q = (\lfloor n/2 \rfloor Q+Q) + (\lfloor n/2 \rfloor Q)$; if n is odd then $nQ+Q = 2(\lfloor n/2 \rfloor Q+Q)$ and $nQ = (\lfloor n/2 \rfloor Q+Q) + (\lfloor n/2 \rfloor Q)$. Either case involves one doubling and one addition.

The formulas, repeated k times, produce $X(nQ+Q), X(nQ), X(Q)$ with k doublings and k additions starting from $X(\lfloor n/2^k \rfloor Q+Q), X(\lfloor n/2^k \rfloor Q), X(Q)$. I compute $X(nQ)$ for any $n \in 2^{254} + 8\{0, 1, \ldots, 2^{251}-1\}$ with 255 doublings and

255 additions starting from $X(Q), X(0), X(Q)$. The first and last few iterations could be simplified.

The final $X(nQ)$, like other X values, is represented as a fraction x/z. I compute $X_0(nQ) = xz^{p-2}$ using a straightforward sequence of 254 squarings and 11 multiplications. This is about 7% of the Curve25519 computation. An extended-Euclid inversion of z, randomized to protect against timing attacks, might be faster, but the maximum potential speedup is very small, while the cost in code complexity is large.

Theorems B.1 and B.2 justify the above procedure if $X_0(Q) \neq 0$. The same formulas also work for $X_0(Q) = 0$: every computed fraction has denominator 0, so the final output is 0 as desired.

Other addition chains. Montgomery pointed out that one can replace the addition chain $\{\lfloor n/2^k \rfloor\} \cup \{\lfloor n/2^k \rfloor + 1\}$ with any differential addition chain (any "Lucas chain"), i.e., any addition chain where each sum is already accompanied by a difference. One can find such a chain with only about 384 elements, as discussed in [59, Section 5]. On the other hand, most of the additions then require $z_1 \neq 1$ in Theorem B.2, costing extra multiplications in $\mathbf{F}_p$. It is also not clear how easily these addition chains can be protected against cache-timing attacks. Further investigation is required.

A more common strategy is to drop the difference requirement, compensate by computing more coordinates of each multiple of Q (Jacobian coordinates, for example, or Chudnovsky coordinates), and use an addition chain with only about 320 elements. See, e.g., [17] or [4]. Unfortunately, even if A is selected so that $y^2 = x^3 + Ax^2 + x$ is isomorphic to a curve $y^2 = x^3 - 3x - a_6$, each doubling in known coordinate systems takes at least 8 field multiplications, and each general addition takes even more. All of my experiments with this strategy have ended up using more field operations, more floating-point operations, and more cycles than the x-coordinate strategy.

One can save a large fraction of the time for computing Curve25519($\underline{n}, \underline{q}$) when q is fixed—in particular, for computing public keys Curve25519($\underline{n}, \underline{9}$)—by precomputing various multiples of $(q, \ldots)$. An essentially optimal algorithm, published by Pippenger in [52] in 1976, computes u public keys with only about $256/\lg 8u$ additions per key. This speedup is negligible in the Diffie-Hellman context (and is not provided by my current software), since each key is used many times; but the speedup is useful for other applications of elliptic curves.

Why this curve? I chose the curve shape $y^2 = x^3 + Ax^2 + x$, as suggested by Montgomery, to allow extremely fast x-coordinate point operations. Curves of this shape have order divisible by 4, requiring a marginally larger prime for the same conjectured security level, but this is outweighed by the extra speed of curve operations. I selected $(A-2)/4$ as a small integer, as suggested by Montgomery, to speed up the multiplication by $(A-2)/4$; this has no effect on the conjectured security level.

To protect against various attacks discussed in Section 3, I rejected choices of A whose curve and twist orders were not $\{4 \cdot \text{prime}, 8 \cdot \text{prime}\}$; here $4, 8$ are minimal since $p \in 1 + 4\mathbf{Z}$. The smallest positive choices for A are 358990, 464586, and 486662. I rejected $A = 358990$ because one of its primes is slightly *smaller* than 2^{252}, raising the question of how standards and implementations should handle the theoretical possibility of a user's secret key matching the prime; discussing this question is more difficult than switching to another A. I rejected 464586 for the same reason. So I ended up with $A = 486662$.

Special curves with small complex automorphisms have potential benefits, as discussed in [31], and are worth further investigation, but so far I have not succeeded in saving time using them.

References

1. — (no editor), *17th annual symposium on foundations of computer science*, IEEE Computer Society, Long Beach, California, 1976. MR 56:1766. See [52].
2. Kazimierz Alster, Jerzy Urbanowicz, Hugh C. Williams (editors), *Public-key cryptography and computational number theory: proceedings of the international conference held in Warsaw, September 11–15, 2000*, Walter de Gruyter, Berlin, 2001. ISBN 3–11–017046–9. MR 2002h:94001. See [60].
3. Adrian Antipa, Daniel Brown, Alfred Menezes, René Struik, Scott Vanstone, *Validation of elliptic curve public keys*, in [21] (2003), 211–223. MR 2171928. Citations in this paper: §1.
4. Roberto M. Avanzi, *Aspects of hyperelliptic curves over large prime fields in software implementations*, in [36] (2004), 148–162. Citations in this paper: §1, §5.
5. Roberto M. Avanzi, *Generic algorithms for computing discrete logarithms*, in [19] (2005), 477–494. MR 2162735. Citations in this paper: §3, §3.
6. Roberto M. Avanzi, Preda Mihăilescu, *Generic efficient arithmetic algorithms for PAFFs (processor adequate finite fields) and related algebraic structures (extended abstract)*, in [43] (2004), 320–334. Citations in this paper: §4.
7. Daniel V. Bailey, Christof Paar, *Efficient arithmetic in finite field extensions with application in elliptic curve cryptography*, Journal of Cryptology **14** (2001), 153–176. ISSN 0933–2790. Citations in this paper: §1, §4.
8. Mihir Bellare (editor), *Advances in cryptology—CRYPTO 2000: proceedings of the 20th Annual International Cryptology Conference held in Santa Barbara, CA, August 20–24, 2000*, Lecture Notes in Computer Science, 1880, Springer-Verlag, Berlin, 2000. ISBN 3–540–67907–3. MR 2002c:94002. See [14].
9. Andreas Bender, Guy Castagnoli, *On the implementation of elliptic curve cryptosystems*, in [16] (1990), 186–192. MR 91d:11154. Citations in this paper: §4.
10. Kamel Bentahar, *The equivalence between the DHP and DLP for elliptic curves used in practical applications, revisited* (2005). URL: `http://eprint.iacr.org/2005/307`. Citations in this paper: §3.
11. Daniel J. Bernstein, *The Poly1305-AES message-authentication code*, in [32] (2005), 32–49. URL: `http://cr.yp.to/papers.html#poly1305`. ID `0018d9551b5546d97c340e0dd8cb5750`. Citations in this paper: §4.
12. Daniel J. Bernstein, *Cache-timing attacks on AES* (2005). URL: `http://cr.yp.to/papers.html#cachetiming`. ID `cd9faae9bd5308c440df50fc26a517b4`. Citations in this paper: §1, §4.
13. Daniel J. Bernstein, *Salsa20 specification* (2005). URL: `http://cr.yp.to/snuffle.html`. Citations in this paper: §3.

14. Ingrid Biehl, Bernd Meyer, Volker Müller, *Differential fault attacks on elliptic curve cryptosystems (extended abstract)*, in [8] (2000), 131–146. URL: `http://lecturer.ukdw.ac.id/vmueller/publications.php`. Citations in this paper: §1, §3.
15. Colin Boyd (editor), *Advances in cryptology—ASIACRYPT 2001: proceedings of the 7th international conference on the theory and application of cryptology and information security held on the Gold Coast, December 9–13, 2001*, Lecture Notes in Computer Science, 2248, Springer-Verlag, Berlin, 2001. ISBN 3–540–42987–5. MR 2003d:94001. See [59].
16. Gilles Brassard (editor), *Advances in cryptology—CRYPTO '89*, Lecture Notes in Computer Science, 435, Springer-Verlag, Berlin, 1990. ISBN 0–387–97317–6. MR 91b:94002. See [9].
17. Michael Brown, Darrel Hankerson, Julio López, Alfred Menezes, *Software implementation of the NIST elliptic curves over prime fields* (2000); see also newer version [18]. URL: `http://www.cacr.math.uwaterloo.ca/techreports/2000/corr2000-56.ps`. Citations in this paper: §1, §5.
18. Michael Brown, Darrel Hankerson, Julio López, Alfred Menezes, *Software implementation of the NIST elliptic curves over prime fields*, in [49] (2001), 250–265; see also older version [17]. MR 1907102.
19. Henri Cohen, Gerhard Frey (editors), *Handbook of elliptic and hyperelliptic curve cryptography*, CRC Press, 2005. ISBN 1–58488–518–1. See [5], [24], [25], [30].
20. Yvo Desmedt (editor), *Advances in cryptology—CRYPTO '94*, Lecture Notes in Computer Science, 839, Springer-Verlag, Berlin, 1994. See [44].
21. Yvo Desmedt, *Public Key Cryptography—PKC 2003, 6th international workshop on theory and practice in public key cryptography, Miami, FL, USA, January 6–8, 2003, proceedings*, Lecture Notes in Computer Science, 2567, Springer, Berlin, 2003. ISBN 3–540–00324–X. See [3].
22. Claus Diem, *The GHS attack in odd characteristic*, Journal of the Ramanujan Mathematical Society **18** (2003), 1–32. MR 2004a:14030. URL: `http://www.math.uni-leipzig.de/~diem/preprints`. Citations in this paper: §4.
23. Whitfield Diffie, Martin Hellman, *New directions in cryptography*, IEEE Transactions on Information Theory **22** (1976), 644–654. ISSN 0018–9448. MR 55:10141. URL: `http://cr.yp.to/bib/entries.html#1976/diffie`. Citations in this paper: §3.
24. Christophe Doche, Tanja Lange, *Arithmetic of elliptic curves*, in [19] (2005), 267–302. MR 2162729. Citations in this paper: §A.
25. Christophe Doche, Tanja Lange, *Arithmetic of special curves*, in [19] (2005), 355–387. MR 2162731. Citations in this paper: §4.
26. Kenny Fong, Darrel Hankerson, Julio López, Alfred Menezes, *Field inversion and point halving revisited* (2003); see also newer version [27]. URL: `http://www.cacr.math.uwaterloo.ca/techreports/2003/tech_reports2003.html`. Citations in this paper: §1.
27. Kenny Fong, Darrel Hankerson, Julio López, Alfred Menezes, *Field inversion and point halving revisited*, IEEE Transactions on Computers **53** (2004), 1047–1059; see also older version [26]. ISSN 0018–9340.
28. Jens Franke, Thorsten Kleinjung, Christof Paar, Jan Pelzl, Christine Priplata, Martin Simka, Colin Stahlke, *An efficient hardware architecture for factoring integers with the elliptic curve method*, Workshop Record of SHARCS 2005 (2005), 51–62. URL: `http://www.best.tuke.sk/simka/pub.html`. Citations in this paper: §3, §3.
29. Gerhard Frey, *How to disguise an elliptic curve (Weil descent)* (1998). URL: `http://www.cacr.math.uwaterloo.ca/conferences/1998/ecc98/slides.html`. Citations in this paper: §4.

30. Gerhard Frey, Tanja Lange, *Transfer of discrete logarithms*, in [19] (2005), 529–543. MR 2162738. Citations in this paper: §3.
31. Robert P. Gallant, Robert J. Lambert, Scott A. Vanstone, *Faster point multiplication on elliptic curves with efficient endomorphisms*, in [38] (2001), 190–200. MR 2003h:14043. Citations in this paper: §5.
32. Henri Gilbert, Helena Handschuh (editors), *Fast software encryption: 12th international workshop, FSE 2005, Paris, France, February 21–23, 2005, revised selected papers*, Lecture Notes in Computer Science, 3557, Springer, 2005. ISBN 3–540–26541–4. See [11].
33. Darrel Hankerson, Julio Lopez Hernandez, Alfred Menezes, *Software implementation of elliptic curve cryptography over binary fields* (2000); see also newer version [34]. URL: `http://www.cacr.math.uwaterloo.ca/techreports/2000/corr2000-42.ps`. Citations in this paper: §1.
34. Darrel Hankerson, Julio Lopez Hernandez, Alfred Menezes, *Software implementation of elliptic curve cryptography over binary fields*, in [40] (2000), 1–24; see also older version [33].
35. Darrel Hankerson, Alfred Menezes, Scott Vanstone, *Guide to elliptic curve cryptography*, Springer, New York, 2004. ISBN 0–387–95273–X. MR 2054891. Citations in this paper: §4.
36. Marc Joye, Jean-Jacques Quisquater (editors), *Cryptographic hardware and embedded systems—CHES 2004: 6th international workshop, Cambridge, MA, USA, August 11–13, 2004, proceedings*, Lecture Notes in Computer Science, 3156, Springer, 2004. ISBN 3–540–22666–4. See [4].
37. Burton S. Kaliski Jr. (editor), *Advances in cryptology—CRYPTO '97: 17th annual international cryptology conference, Santa Barbara, California, USA, August 17–21, 1997, proceedings*, Lecture Notes in Computer Science, 1294, Springer, 1997. ISBN 3–540–63384–7. MR 99a:94041. See [42].
38. Joe Kilian (editor), *Advances in cryptology: CRYPTO 2001, 21st annual international cryptology conference, Santa Barbara, California, USA, August 19–23, 2001, proceedings*, Lecture Notes in Computer Science, 2139, Springer, 2001. ISBN 3–540–42456–3. MR 2003d:94002. See [31].
39. Neal Koblitz, Alfred J. Menezes, *Another look at "provable security"* (2004). URL: `http://www.cacr.math.uwaterloo.ca/~ajmeneze/publications/provable.pdf`. Citations in this paper: §3.
40. Çetin Kaya Koç, Christof Paar, *Cryptographic hardware and embedded systems—CHES 2000: Proceedings of the 2nd International Workshop held in Worcester, MA, USA, August 2000*, Lecture Notes in Computer Science, Springer, 2000. ISBN 3–540–42521–7. See [34].
41. Fabian Kuhn, Rene Struik, *Random walks revisited: extensions of Pollard's rho algorithm for computing multiple discrete logarithms*, in [64] (2001), 212–229. URL: `http://www.distcomp.ethz.ch/publications.html`. Citations in this paper: §3.
42. Chae Hoon Lim, Pil Joong Lee, *A key recovery attack on discrete log-based schemes using a prime order subgroup*, in [37] (1997), 249–263. URL: `http://dasan.sejong.ac.kr/~chlim/english_pub.html`. Citations in this paper: §3, §3.
43. Mitsuru Matsui, Robert Zuccherato (editors), *Selected areas in cryptography: 10th annual international workshop, SAC 2003, Ottawa, Canada, August 14–15, 2003, revised papers*, Lecture Notes in Computer Science, 3006, Springer, 2004. ISBN 3–540–21370–8. See [6].
44. Ueli M. Maurer, *Towards the equivalence of breaking the Diffie-Hellman protocol and computing discrete logarithms*, in [20] (1994), 271–281. URL: `http://www.crypto.ethz.ch/~maurer/publications.html`. Citations in this paper: §3.

45. Alfred Menezes, *Another look at HMQV* (2005). URL: `http://eprint.iacr.org/2005/205`. Citations in this paper: §2.
46. Victor S. Miller, *Use of elliptic curves in cryptography*, in [65] (1986), 417–426. MR 88b:68040. Citations in this paper: §1.
47. 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://cr.yp.to/bib/entries.html#1987/montgomery`. Citations in this paper: §5.
48. A. Muzereau, Nigel P. Smart, Frederik Vercauteren, *The equivalence between the DHP and DLP for elliptic curves used in practical applications*, LMS Journal of Computation and Mathematics **7** (2004), 50–72. URL: `http://www.lms.ac.uk/jcm/7/lms2003-034/`. Citations in this paper: §3.
49. David Naccache (editor), *Topics in cryptology—CT-RSA 2001: Proceedings of the Cryptographers' Track at the RSA Conference held in San Francisco, CA, April 8–12, 2001*, Lecture Notes in Computer Science, 2020, Springer, 2001. ISBN 3–540–41898–9. MR 2003a:94039. See [18].
50. Dag Arne Osvik, Adi Shamir, Eran Tromer, *Cache atacks and countermeasures: the case of AES (extended version)* (2005). URL: `http://www.wisdom.weizmann.ac.il/~tromer/`. Citations in this paper: §1.
51. Colin Percival, *Cache missing for fun and profit* (2005). URL: `http://www.daemonology.net/hyperthreading-considered-harmful/`. Citations in this paper: §1.
52. Nicholas Pippenger, *On the evaluation of powers and related problems (preliminary version)*, in [1] (1976), 258–263; newer version split into [53] and [54]. MR 58:3682. URL: `http://cr.yp.to/bib/entries.html#1976/pippenger`. Citations in this paper: §5.
53. Nicholas Pippenger, *The minimum number of edges in graphs with prescribed paths*, Mathematical Systems Theory **12** (1979), 325–346; see also older version [52]. ISSN 0025–5661. MR 81e:05079. URL: `http://cr.yp.to/bib/entries.html#1979/pippenger`.
54. Nicholas Pippenger, *On the evaluation of powers and monomials*, SIAM Journal on Computing **9** (1980), 230–250; see also older version [52]. ISSN 0097–5397. MR 82c:10064. URL: `http://cr.yp.to/bib/entries.html#1980/pippenger`.
55. John M. Pollard, *Kangaroos, Monopoly and discrete logarithms*, Journal of Cryptology **13** (2000), 437–447. ISSN 0933–2790. Citations in this paper: §3.
56. John Proos, Christof Zalka, *Shor's discrete logarithm quantum algorithm for elliptic curves* (2003). URL: `http://www.cacr.math.uwaterloo.ca/techreports/2003/tech_reports2003.html`. Citations in this paper: §1.
57. Nigel P. Smart, *A comparison of different finite fields for use in elliptic curve cryptosystems* (2000); see also newer version [58]. URL: `http://www.cs.bris.ac.uk/Publications/pub_info.jsp?id=1000458`.
58. Nigel P. Smart, *A comparison of different finite fields for elliptic curve cryptosystems*, Computers and Mathematics with Applications **42** (2001), 91–100; see also older version [57]. MR 2002c:94033. Citations in this paper: §1.
59. Martijn Stam, Arjen K. Lenstra, *Speeding up XTR*, in [15] (2001), 125–143. MR 2003h:94049. Citations in this paper: §5.
60. Edlyn Teske, *Square-root algorithms for the discrete logarithm problem (a survey)*, in [2] (2001), 283–301. MR 2003c:11156. URL: `http://www.cacr.math.uwaterloo.ca/~eteske/publications.html`. Citations in this paper: §3.

61. Edlyn Teske, *Computing discrete logarithms with the parallelized kangaroo method* (2001); see also newer version [62]. URL: `http://www.cacr.math.uwaterloo.ca/techreports/2001/tech_reports2001.html`. Citations in this paper: §3.
62. Edlyn Teske, *Computing discrete logarithms with the parallelized kangaroo method*, Discrete Applied Mathematics **130** (2003), 61–82; see also older version [61]. MR 2004h:11112.
63. Paul C. van Oorschot, Michael Wiener, *Parallel collision search with cryptanalytic applications*, Journal of Cryptology **12** (1999), 1–28. ISSN 0933–2790. URL: `http://members.rogers.com/paulv/papers/pubs.html`. Citations in this paper: §3.
64. Serge Vaudenay, Amr M. Youssef (editors), *Selected areas in cryptography: 8th annual international workshop, SAC 2001, Toronto, Ontario, Canada, August 16–17, 2001, revised papers*, Lecture Notes in Computer Science, 2259, Springer, 2001. ISBN 3–540–43066–0. MR 2004k:94066. See [41].
65. Hugh C. Williams (editor), *Advances in cryptology: CRYPTO '85*, Lecture Notes in Computer Science, 218, Springer, Berlin, 1986. ISBN 3–540–16463–4. See [46].

A Appendix: rings, fields, and curves

This appendix reviews elliptic curves at the level of generality of Theorem 2.1. See [24, Chapter 13] for much more information about elliptic curves.

The base field. Let p be a prime number with $p \ge 5$. Define $\mathbf{F}_p$ as the set $\{0, 1, \ldots, p-1\}$. Define a binary operation $+$ on $\mathbf{F}_p$ as addition mod p. Define a binary operation $\cdot$ on $\mathbf{F}_p$ as multiplication mod p. Define a unary operation $-$ on $\mathbf{F}_p$ as negation mod p.

$\mathbf{F}_p$ is a commutative ring under $0, 1, -, +, \cdot$. This means that it satisfies every $0, 1, -, +, \cdot$ identity satisfied by $\mathbf{Z}$; e.g., the identity $a(b+c+1) = ab + ac + a$. Furthermore, because p is prime, $\mathbf{F}_p$ is a field: every nonzero element of $\mathbf{F}_p$ has a reciprocal in $\mathbf{F}_p$.

Squares in the base field. Squaring is a 2-to-1 map on the nonzero elements of $\mathbf{F}_p$, so there are exactly $(p-1)/2$ non-squares in $\mathbf{F}_p$. Find the smallest $\delta \in \{1, 2, \ldots, p-1\}$ such that δ is not a square in $\mathbf{F}_p$.

Fermat's little theorem implies that $\alpha^{(p-1)/2} = 1$ if α is a nonzero square in $\mathbf{F}_p$; $\alpha^{(p-1)/2} = -1$ if α is a non-square in $\mathbf{F}_p$; and $\alpha^{(p-1)/2} = 0$ if $\alpha = 0$. Consequently, if α is a non-square in $\mathbf{F}_p$, then α/δ is a nonzero square in $\mathbf{F}_p$.

The extension field. Define $\mathbf{F}_{p^2}$ as the set $\mathbf{F}_p \times \mathbf{F}_p$. Define a unary operation $-$ on $\mathbf{F}_{p^2}$ by $-(c, d) = (-c, -d)$. Define a binary operation $+$ on $\mathbf{F}_{p^2}$ by $(a, b) + (c, d) = (a + c, b + d)$. Define a binary operation $\cdot$ on $\mathbf{F}_{p^2}$ by $(a, b) \cdot (c, d) = (ac + \delta bd, ad + bc)$.

$\mathbf{F}_{p^2}$ is a commutative ring under $0, 1, -, +, \cdot$. Furthermore, each nonzero $(a, b) \in \mathbf{F}_{p^2}$ has a reciprocal $(a/(a^2 - \delta b^2), -b/(a^2 - \delta b^2)) \in \mathbf{F}_{p^2}$.

The injection $a \mapsto (a, 0)$ from $\mathbf{F}_p$ to $\mathbf{F}_{p^2}$ is a ring morphism: it preserves $0, 1, -, +, \cdot$. Thus $(a, 0)$ is abbreviated a without risk of confusion. The element $(0, 1)$ of $\mathbf{F}_{p^2}$ is abbreviated $\sqrt{\delta}$; it satisfies $\sqrt{\delta}^2 = (\delta, 0) = \delta$.

The elliptic curve. Let A be an integer such that $A^2 - 4 \bmod p$ is not a square in $\mathbf{F}_p$. Define $E(\mathbf{F}_{p^2})$ as $\{\infty\} \cup \{(x, y) \in \mathbf{F}_{p^2} : y^2 = x^3 + Ax^2 + x\}$.

Define a unary operation $-$ on $E(\mathbf{F}_{p^2})$ as follows: $-\infty = \infty$; $-(x, y) = (x, -y)$. Define a binary operation $+$ on $E(\mathbf{F}_{p^2})$ as follows:

- $\infty + \infty = \infty$.
- $\infty + (x, y) = (x, y)$.
- $(x, y) + \infty = (x, y)$.
- $(x, y) + (x, -y) = \infty$.
- If $y \neq 0$ then $(x, y) + (x, y) = (x'', y'')$ where $\lambda = (3x^2 + 2Ax + 1)/2y$, $x'' = \lambda^2 - A - 2x = (x^2 - 1)^2/4y^2$, and $y'' = \lambda(x - x'') - y$. Here / refers to division in $\mathbf{F}_{p^2}$.
- If $x' \neq x$ then $(x, y) + (x', y') = (x'', y'')$ where $\lambda = (y' - y)/(x' - x)$, $x'' = \lambda^2 - A - x - x'$, and $y'' = \lambda(x - x'') - y$.

Standard (although lengthy) calculations show that $E(\mathbf{F}_{p^2})$ is a commutative group under $\infty, -, +$. This means that every $0, -, +$ identity satisfied by $\mathbf{Z}$ is also satisfied by $E(\mathbf{F}_{p^2})$ when 0 is replaced by ∞.

Note that the following three sets are subgroups of $E(\mathbf{F}_{p^2})$:

- $\{\infty, (0, 0)\}$. Indeed, $\infty + \infty = \infty$; $(0, 0) + (0, 0) = \infty$; and $(0, 0) + \infty = (0, 0)$.
- $\{\infty\} \cup (E(\mathbf{F}_{p^2}) \cap (\mathbf{F}_p \times \mathbf{F}_p))$. Indeed, if $x, y, x', y' \in \mathbf{F}_p$ then the quantities λ, x'', y'' defined above are in $\mathbf{F}_p$.
- $\{\infty\} \cup (E(\mathbf{F}_{p^2}) \cap (\mathbf{F}_p \times \sqrt{\delta}\mathbf{F}_p))$. This time λ is a ratio of an element of $\mathbf{F}_p$ and an element of $\sqrt{\delta}\mathbf{F}_p$, and is therefore an element of $\sqrt{\delta}\mathbf{F}_p$, producing $x'' \in \mathbf{F}_p$ and $y'' \in \sqrt{\delta}\mathbf{F}_p$.

Note also that if $x^3 + Ax^2 + x = 0$ in $\mathbf{F}_p$ then $x = 0$. (Otherwise $A^2 - 4 = (x - 1/x)^2$ in $\mathbf{F}_p$, so $A^2 - 4 \bmod p$ is a square in $\mathbf{F}_p$, contradiction.) In other words, $(x, 0) \notin E(\mathbf{F}_{p^2})$ if $x \neq 0$.

Proof of Theorem 2.1. Let n be an integer. Let q be an element of $\mathbf{F}_p$. Define $\alpha = q^3 + Aq^2 + q$. Define $X_0 : E(\mathbf{F}_{p^2}) \to \mathbf{F}_{p^2}$ as follows: $X_0(\infty) = 0$; $X_0(x, y) = x$.

I will show that there are exactly two $Q \in E(\mathbf{F}_{p^2})$ such that $X_0(Q) = q$, that both of them have the same value of $X_0(nQ)$, and that the value is in $\mathbf{F}_p$. Here nQ means the nth multiple of Q under the above group operations on $E(\mathbf{F}_{p^2})$.

Case 1: $\alpha = 0$. Then $q = 0$. The only square root of 0 in $\mathbf{F}_{p^2}$ is 0, so $\{Q \in E(\mathbf{F}_{p^2}) : X_0(Q) = q\}$ is exactly the group $\{\infty, (0, 0)\}$. Thus each $Q \in E(\mathbf{F}_{p^2})$ with $X_0(Q) = q$ has $nQ \in \{\infty, (0, 0)\}$; i.e., $X_0(nQ) = 0$.

Case 2: α is a nonzero square in $\mathbf{F}_p$. Select a square root r. Now $q \neq 0$, and the only square roots of $q^3 + Aq^2 + q$ in $\mathbf{F}_{p^2}$ are $\pm r$, so $\{Q \in E(\mathbf{F}_{p^2}) : X_0(Q) = q\} = \{(q, r), (q, -r)\}$. Define $s = X_0(n(q, r))$. The group $\{\infty\} \cup (E(\mathbf{F}_{p^2}) \cap (\mathbf{F}_p \times \mathbf{F}_p))$ contains (q, r), so it contains $n(q, r)$, so $s \in \{0, 1, 2, 3, \ldots, p - 1\}$. Furthermore $n(q, -r) = n(-(q, r)) = -n(q, r)$, so $X_0(n(q, -r)) = X_0(n(q, r)) = s$. Thus $X_0(nQ) = s$ for all $Q \in E(\mathbf{F}_{p^2})$ such that $X_0(Q) = q$.

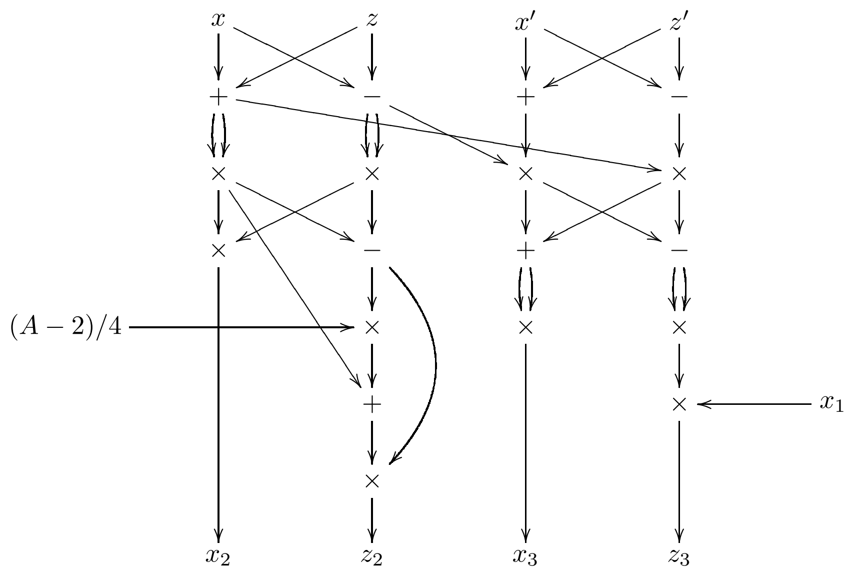
Case 3: α is a non-square in $\mathbf{F}_p$. Then α/δ is a nonzero square in $\mathbf{F}_p$. Select a square root r. Now $q \neq 0$, and the only square roots of $q^3 + Aq^2 + q$ in $\mathbf{F}_{p^2}$ are $\pm r\sqrt{\delta}$, so $\{Q \in E(\mathbf{F}_{p^2}) : X_0(Q) = q\} = \{(q, r\sqrt{\delta}), (q, -r\sqrt{\delta})\}$. Define

$s = X_0(n(q, r\sqrt{\delta}))$. The group $\{\infty\} \cup (E(\mathbf{F}_{p^2}) \cap (\mathbf{F}_p \times \sqrt{\delta}\mathbf{F}_p))$ contains $(q, r\sqrt{\delta})$, so it contains $n(q, r\sqrt{\delta})$, so $s \in \{0, 1, 2, 3, \ldots, p-1\}$. Furthermore $n(q, -r\sqrt{\delta}) = n(-(q, r\sqrt{\delta})) = -n(q, r\sqrt{\delta})$, so $X_0(n(q, -r\sqrt{\delta})) = X_0(n(q, r\sqrt{\delta})) = s$. Thus $X_0(nQ) = s$ for all $Q \in E(\mathbf{F}_{p^2})$ such that $X_0(Q) = q$. $\square$

B Appendix: Montgomery's double-and-add formulas

This appendix states Montgomery's x-coordinate double-and-add formulas, and proves that the formulas work whenever $Q - Q' \notin \{\infty, (0,0)\}$.

The following diagram summarizes Montgomery's formulas in the case $z_1 = 1$. As in Theorems B.1 and B.2, x/z and x'/z' are the x-coordinates of points Q, Q'; x_2/z_2 is the x-coordinate of $2Q$; x_1 is the x-coordinate of $Q - Q'$; and x_3/z_3 is the x-coordinate of $Q + Q'$.



One can see at a glance that there are 4 squarings, 1 multiplication by $(A-2)/4$, and 5 other multiplications; and that there are 8 additions/subtractions, none of which produce input to another addition/subtraction.

Theorem B.1. *Let p be a prime number with $p \geq 5$. Let A be an integer such that $A^2 - 4$ is not a square modulo p. Define E as the elliptic curve $y^2 = x^3 + Ax^2 + x$ over the field $\mathbf{F}_p$. Define $X : E(\mathbf{F}_{p^2}) \to \{\infty\} \cup \mathbf{F}_{p^2}$ as follows: $X(\infty) = \infty$; $X(x, y) = x$. Fix $x, z \in \mathbf{F}_p$ with $(x, z) \neq (0, 0)$. Define*

$$\begin{aligned}
x_2 &= (x^2 - z^2)^2 = (x-z)^2(x+z)^2,\\
z_2 &= 4xz(x^2 + Axz + z^2)\\
&= ((x+z)^2 - (x-z)^2)\left((x+z)^2 + \frac{A-2}{4}((x+z)^2 - (x-z)^2)\right).
\end{aligned}$$

Then $X(2Q) = x_2/z_2$ *for all* $Q \in E(\mathbf{F}_{p^2})$ *such that* $X(Q) = x/z$.

Here x/z means the quotient of x and z in $\mathbf{F}_p$ if $z \neq 0$; it means ∞ if $x \neq 0$ and $z = 0$; it is undefined if $x = z = 0$.

Proof. Case 1: $z = 0$. Then $x_2 = x^4 \neq 0$ and $z_2 = 0$. Also $X(Q) = x/0 = \infty$ so $Q = \infty$ so $2Q = \infty$ so $X(2Q) = \infty = x_2/0 = x_2/z_2$.

Case 2: $z \neq 0$ and $x = 0$. Then $x_2 = z^4 \neq 0$ and $z_2 = 0$. Also $X(Q) = 0/z = 0$ so $Q = (0, 0)$ so $2Q = \infty$ so $X(2Q) = \infty = x_2/0 = x_2/z_2$.

Case 3: $z \neq 0$ and $x \neq 0$. Then $Q = (x/z, y)$ for some $y \in \mathbf{F}_{p^2}$ satisfying $y^2 = (x/z)^3 + A(x/z)^2 + (x/z)$ and thus $4y^2z^4 = 4(x^3z + Ax^2z^2 + xz^3) = z_2$. The non-squareness of $A^2 - 4$ implies that $y \neq 0$; hence $z_2 \neq 0$. Also $X(2Q) = ((x/z)^2 - 1)^2/4y^2$ by definition of doubling; thus $z_2X(2Q) = z^4((x/z)^2 - 1)^2 = (x^2 - z^2)^2 = x_2$. $\square$

Theorem B.2. *In the context of Theorem B.1, fix* $x, z, x', z', x_1, z_1 \in \mathbf{F}_p$ *with* $(x, z) \neq (0, 0)$, $(x', z') \neq (0, 0)$, $x_1 \neq 0$, *and* $z_1 \neq 0$. *Define*

$$x_3 = 4(xx' - zz')^2 z_1 = ((x - z)(x' + z') + (x + z)(x' - z'))^2 z_1,$$
$$z_3 = 4(xz' - zx')^2 x_1 = ((x - z)(x' + z') - (x + z)(x' - z'))^2 x_1.$$

Then $X(Q+Q') = x_3/z_3$ *for all* $Q, Q' \in E(\mathbf{F}_{p^2})$ *such that* $X(Q) = x/z$, $X(Q') = x'/z'$, *and* $X(Q - Q') = x_1/z_1$.

Proof. Case 1: $Q = Q'$. Then $X(Q - Q') = X(\infty) = \infty$, so $z_1 = 0$, contradiction.

Case 2: $Q = \infty$. Then $z = 0$ and $x \neq 0$; also $X(Q - Q') = X(-Q') = X(Q')$, so $x_1/z_1 = x'/z'$, so $x' \neq 0$ and $z' \neq 0$. Finally $x_3 = 4(xx')^2 z_1$ and $z_3 = 4(xz')^2 x_1$ so $x_3/z_3 = (x'/z')^2 z_1/x_1 = x'/z' = X(Q') = X(Q + Q')$.

Case 3: $Q' = \infty$. Then $z' = 0$ and $x' \neq 0$; also $X(Q - Q') = X(Q)$, so $x_1/z_1 = x/z$, so $x \neq 0$ and $z \neq 0$. Finally $x_3 = 4(xx')^2 z_1$ and $z_3 = 4(zx')^2 x_1$ so $x_3/z_3 = (x/z)^2 z_1/x_1 = x/z = X(Q) = X(Q + Q')$.

Case 4: $Q = -Q'$. Then $X(Q') = X(Q)$ so $x/z = x'/z'$ so $xz' = zx'$ so $z_3 = 0$.

Suppose that $x_3 = 0$. Then $(x - z)(x' + z') + (x + z)(x' - z') = 0$ and $(x-z)(x'+z')-(x+z)(x'-z') = 0$, so $(x-z)(x'+z') = 0$ and $(x+z)(x'-z') = 0$. If $x+z \neq 0$ then $x' - z' = 0$ so $x' + z' = 2x' \neq 0$ so $x - z = 0$; i.e., $X(Q) = 1$ and $X(Q') = 1$. Otherwise $x = -z$ so $x - z = 2x \neq 0$ so $x' = -z'$; i.e., $X(Q) = -1$ and $X(Q') = -1$. Either way $X(Q - Q') = X(2Q) = (X(Q)^2 - 1)^2/\cdots = (1 - 1)^2/\cdots = 0$ by definition of doubling, so $x_1 = 0$, contradiction.

Thus $x_3 \neq 0$, and $x_3/z_3 = \infty = X(\infty) = X(Q + Q')$.

Case 5: $Q \neq \infty$; $Q' \neq \infty$; $Q \neq Q'$; and $Q \neq -Q'$. Then $z \neq 0$, $z' \neq 0$, and $x/z \neq x'/z'$, so $z_3 \neq 0$. Find $y, y' \in \mathbf{F}_{p^2}$ such that $Q = (x/z, y)$ and $Q' = (x'/z', y')$. Write $\alpha = x'/z' - x/z$ and $\beta = A + x/z + x'/z'$. Then $X(Q+Q') = ((y' - y)/\alpha)^2 - \beta$ and $X(Q - Q') = ((-y' - y)/\alpha)^2 - \beta$ by definition of $Q \pm Q'$, so $X(Q + Q')X(Q - Q') = \beta^2 - 2\beta((y')^2 + y^2)/\alpha^2 + ((y')^2 - y^2)^2/\alpha^4$. Substitute $y^2 = (x/z)^3 + A(x/z)^2 + (x/z)$ and $(y')^2 = (x'/z')^3 + A(x'/z')^2 + (x'/z')$ and simplify to see that $X(Q+Q')X(Q-Q') = (xx' - zz')^2/(xz' - x'z)^2$; this is what Montgomery did. Finally $X(Q+Q') = (xx' - zz')^2 z_1/(xz' - x'z)^2 x_1 = x_3/z_3$. $\square$