ON ELLIPTIC CURVE CRYPTOGRAPHIC PROTOCOLS

CHRISTIAN ZHOU-ZHENG

Abstract. The group structure of elliptic curves over finite fields gives rise to the elliptic curve discrete logarithm problem, a one-way problem that lends itself exceptionally well to cryptography. This paper will explain the theory and implementation behind some elliptic curve-based modifications to public key cryptosystems based on the multiplicative group of a finite field.

1. Introduction

Existing classical cryptosystems are primarily based around the structure of the multiplicative group of a finite field $\mathbb{F}_p^\times$. In conjunction with the generalization of existing cryptosystems to any finite abelian group, H. W. Lenstra's 1987 discovery of integer factoring algorithm based on elliptic curves immediately provoked the study of cryptosystems based on the group structure of the points on an elliptic curve over a finite field. Koblitz and Miller were among the first to propose such systems immediately following Lenstra's announcement, and the first proofs-of-concept were implemented shortly after. Practically any cryptosystem based on the discrete logarithm problem—that is, nearly all public key cryptosystems at the time—could be instead converted to be based on the elliptic curve discrete logarithm problem for increased security and efficiency.

We begin by introducing the mathematical background for elliptic curves over finite fields and their underlying groups, before introducing the principal problems of the discrete logarithm and elliptic curve discrete logarithm. We then compare and contrast two cryptographic protocols, the Diffie-Hellman key exchange system and the Digital Signature Algorithm, when based on the discrete logarithm problem as opposed to the elliptic curve discrete logarithm problem. We conclude with some remarks on general comparisons between elliptic curve cryptography and other classical cryptography, and provide appendices with further details about real-world implementations, shortcomings, and concerns.

2. Preliminaries

Analogously to how classical cryptography takes advantage of the group structure of the multiplicative group of integers modulo p, $(\mathbb{Z}/p\mathbb{Z})^\times \simeq \mathbb{F}_p^\times$, elliptic-curve cryptography takes advantage of the group structure inherent to elliptic curves over finite fields. Thus we define an elliptic curve.

Definition 2.1. An elliptic curve $E(\mathbb{F}_q)$ defined over a finite field $\mathbb{F}_q$ is the set of points $(x, y) \in \mathbb{F}_q^2$ that satisfy the equation $y^2 = x^3 + ax + b$, with $a, b \in \mathbb{F}_q$, along with a "point at infinity" $\mathcal{O}$ that makes this set an abelian group under point addition. Furthermore, $\Delta = -16(4a^3 + 27b^2)$ must not equal 0.

Date: December 1, 2023.

For our purposes, we suppose $\mathbb{F}_q$ is not of characteristic 2 or 3, because the curve equation over fields of those characteristics cannot be written in the short Weierstrass form $y^2 = x^3 + ax + b$ as above. The last condition is required because $\Delta = 0$ implies the defining cubic curve has repeated roots and therefore at least one singular point, which in turn implies the curve is singular and can be transformed into either the form $y^2 = x^3$ or $y^2 = x^3 - x^2$ with an appropriate change of variable. This in turn implies the set of curve points is isomorphic to a finite field—either $\mathbb{F}_p^+$ or $\mathbb{F}_{p^2}^\times$, respectively. Not only does this cause the loss of the unique structure which elliptic curves provide, it also completely nullifies any benefit over classical cryptography, which already uses $\mathbb{F}_p^\times$.

The elliptic curve group operation, point addition, can be geometrically thought of as drawing the secant line between two points $P_1, P_2 \in E(\mathbb{F}_q)$ (or the tangent line at P_1, in the case of point doubling) and negating the third intersection point of this line with the curve $E(\mathbb{F}_q)$. The negative of a point in $E(\mathbb{F}_q)$ is the point with the same x-coordinate but negated y-coordinate.

Mathematically, if we have $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, then $P_3 = (x_3, y_3)$ is given by the equations

$$x_3 = m^2 - x_1 - x_2 \qquad y_3 = -y_1 + m(x_1 - x_3), \tag{2.1}$$

where m is defined as

$$m = \frac{y_2 - y_1}{x_2 - x_1} \text{ if } P_1 \neq P_2 \qquad m = \frac{3x_1^2 + a}{2y_1} \text{ if } P_1 = P_2. \tag{2.2}$$

We are particularly interested in the case in which $P_1 = P_2$, known as point doubling, as it allows tractable calculation of higher multiples of a point. Point multiplication by n is denoted nP, which can be computed in $O(\log n)$ operations of point additions and doublings.

Algorithm 2.2. A simple recursive algorithm for point multiplication, the double-and-add method, is as follows:

(1) Begin with multiplier n and point P.
(2) If n is odd, add P and subtract 1 from n.
    If n is even, double P and halve n.
(3) Repeat Step 2 until $n = 0$.

Example. Finding $11P$ can be done in 3 doublings and 2 point additions as follows:

$$11P = 2(5P) + P = 2(2(2P) + P) + P$$

Point multiplication forms the backbone of elliptic-curve cryptography, as it allows the retrieval of a point nP seemingly unrelated to the original point P from specifying only one parameter, the multiplier n. While nP can be computed in reasonable time, it is very difficult to compute n and P given nP; this is known as the *elliptic curve discrete logarithm problem*, or ECDLP, formally stated as follows:

Problem 2.3. Given points $P_1, P_2 \in E(\mathbb{F}_q)$, find an integer n such that $P_2 = nP_1$.

Example. On the curve $E(\mathbb{F}_{97}) \coloneqq y^2 = x^3 + 5x + 3$ and starting point $P = (1, 3)$, we get $22P = (9, 96)$—which has no obvious relationship to 22 or P whatsoever! Furthermore, as shown in Figure 1, adding P to this to yield $23P = (63, 28)$ has no visible relationship to 23, P, or $22P$ either! There are very few hints to glean any useful information from Q.

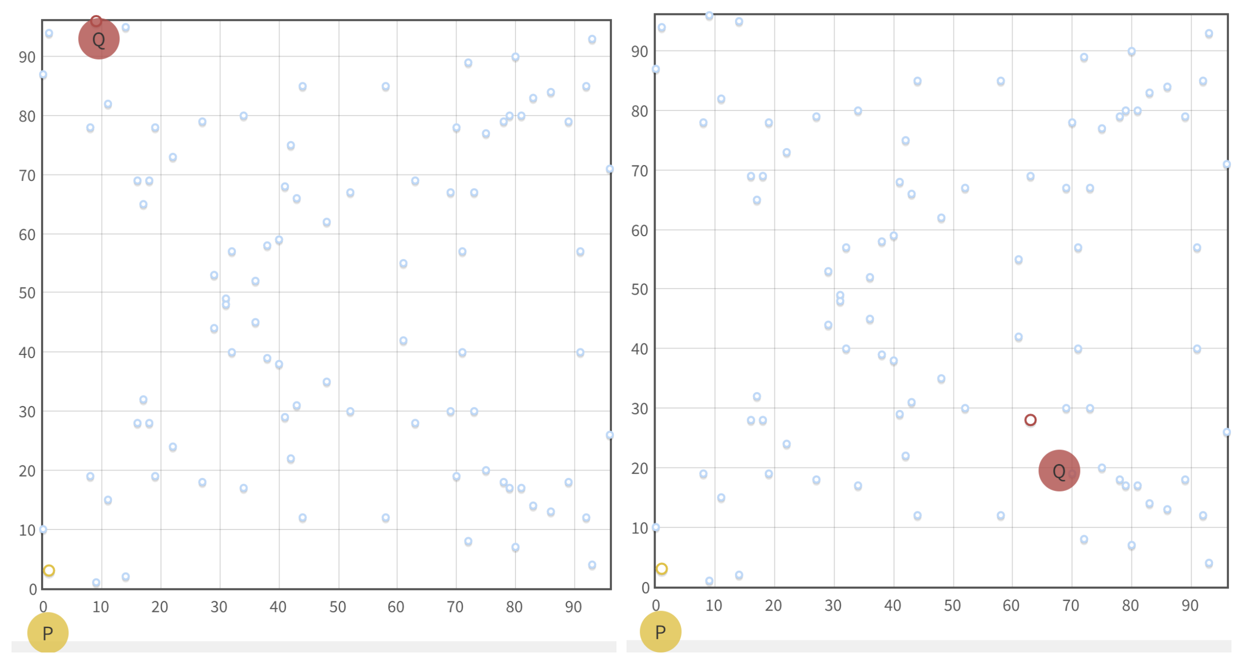

Figure 1. $\mathbb{F}_{97}^2$ is graphed with points on $E(\mathbb{F}_{97})$ shown. P denotes the starting point, $(1, 3)$. On the left, $Q = 22P$, and on the right, $Q = 23P$.

The astute reader may notice that the ECDLP is reminiscent of the general discrete logarithm problem over a finite group G, stated as follows:

Problem 2.4. Given elements $a, b \in G$, find an integer n such that $b = a^n$.

However, the ECDLP is remarkably more difficult to crack than the generic DLP, since the index calculus attacks commonly used to break the DLP fail over well-chosen elliptic curves. Thus, cryptographic systems based on the DLP can be easily modified to base their security on the ECDLP, as we will now examine.

Remark. The index calculus method for finding discrete logarithms involves finding the discrete logarithms of a set of small primes called the factor base, and then computing the discrete logarithm of the desired number in terms of the discrete logarithms of the factor base. This fails for elliptic curves purely because there is no notion of a ”prime element” in the set of points on an elliptic curve.

3. The Diffie-Hellman System

3.1. **Finite Field Diffie-Hellman.** The Diffie-Hellman key exchange system is one of the oldest public-key protocols for establishing a shared secret key over an insecure channel. It is based on the eponymous *Diffie-Hellman problem*, or DHP, stated as follows:

Problem 3.1. Given a generator $g \in G$ and the values g^a and g^b, what is the value of g^{ab}?

Typically G will be $\mathbb{F}_p^\times \simeq (\mathbb{Z}/p\mathbb{Z})^\times$, hence the name *finite field* Diffie-Hellman. The most efficient known way to solve the DHP is to solve Problem 2.4, the DLP, to find $\log_g g^x = x$ and raise g^y to that power once found. Assume all computation is done modulo p.

Curiously, the intractability of solving this problem, and the inexistence of an easier solution, is taken as an *assumption* in cryptographic circles (albeit one that has stood the test of three decades and counting). In fact, no proof yet exists that either the DLP or DHP are *actually* hard problems, which would imply that P $\neq$ NP!

Algorithm 3.2 (Finite Field Diffie-Hellman)**.**

(1) Decide the domain parameters (g, p) over the public channel.
  - g: the integer base.
  - p: the prime modulus for $\mathbb{F}_p^\times$.

(2) Determine the private keys a, b. Do not share these.
(3) Generate the public keys $A = g^a$, $B = g^b$ and share over the public channel.
(4) Obtain shared secret key s from the public keys $s = A^b = B^a$.

Remark on Security. The only public information when executing Algorithm 3.2 is as follows:

- g, the integer base.
- p, the prime modulus.
- A, Alice's public key.
- B, Bob's public key.

Crucially, neither private key, a or b, is known to the attacker. With only the information given, they must solve the DHP to determine the shared secret. The easiest way to do so is to attempt to retrieve a private key from one of the public keys, which in turn requires solving the DLP to find $\log_g A = \log_g ag = a$ or $\log_g B = \log_g bg = b$.

3.2. **Elliptic Curve Diffie-Hellman.** A clear analog to the DHP holds in the context of elliptic curves, the *elliptic curve Diffie-Hellman problem*, or ECDHP:

Problem 3.3. Given a point $P \in E(\mathbb{F}_q)$ and points aP and bP, what is the value of abP?

It is very convenient to translate the implementation of Algorithm 3.2 into the language of elliptic curves and the ECDHP; in fact, the analogous elliptic curve algorithm is practically word-for-word identical after the relevant substitutions have been made. Assume all computation is done in $E(\mathbb{F}_q)$.

Algorithm 3.4 (Elliptic Curve Diffie-Hellman)**.**

(1) Decide the domain parameters (q, a, b, G, n, h) over the public channel.
  - q: the order of $\mathbb{F}_q$.
  - a, b: the coefficients defining $E(\mathbb{F}_q)$.
  - G, a point generating a cyclic subgroup of $E(\mathbb{F}_q)$.
  - n, the order of G or the smallest integer such that $nG = \mathcal{O}$.
  - h, the cofactor, defined by $h = \frac{1}{n}|E(\mathbb{F}_q)|$.

(2) Determine the private keys $c, d \in [1, n-1]$. Do not share these.
(3) Generate the public keys $C = cG$, $D = dG$ and share over the public channel.
(4) Obtain shared secret key S from the public keys $S = dC = cD$.
(5) Convert shared secret key S from a curve point to a usable form, as detailed in Appendix A.2.

Remark on Security. The only public information when executing Algorithm 3.4 is as follows:

- q, the order of $\mathbb{F}_q$.
- a and b, the coefficients defining $E(\mathbb{F}_q)$.

- G, the generator of the used cyclic subgroup of $E(\mathbb{F}_q)$.
- C and D, the public keys.

n and h are also publicly available, which are purely computational aids and provide no opportunity for attack. Once again, neither private key is known to the attacker, so they must solve the ECDHP to determine the shared secret; the easiest known way to do so is to attempt to retrieve a private key from one of the public keys, which requires solving the ECDLP to find $\log_G C = \log_G cG = c$ or $\log_G D = \log_G dG = d$.

4. The Digital Signature Algorithm

4.1. **Standard Digital Signature Algorithm.** The Digital Signature Algorithm, or DSA, is a digital signature scheme used to verify the authenticity of documents purported to have been created by a particular person. It does not rely on any particularly named problem like the Diffie-Hellman system does. DSA is actually comprised of three separate algorithms: a key generation algorithm, a signature generation algorithm, and a signature verification algorithm.

Algorithm 4.1 (DSA Key Generation)**.**

(1) Choose a cryptographic hash function[1] H with output length m bits.
(2) Choose a prime q of over m bits, and another prime p such that $q \mid p-1$.
(3) Find a generator g for the subgroup of order q in $\mathbb{F}_p^\times$ by computing $g = h^{(p-1)/q} \pmod{p}$ for random elements $h \in \mathbb{F}_p^\times$ until $g \neq 1$.
(4) Save the domain parameters (H, p, q, g) for distribution.
(5) For each individual user:
    (a) Choose a random integer $x \in [1, q-1]$. This is the private key; do not share it.
    (b) Compute $y = g^x \pmod{p}$. This is the public key; share it!

Remark. DSA key generation essentially defines a hash function, a subgroup of $\mathbb{F}_p^\times$ of order q, and a generator for this group. Each user then gets a random element of this subgroup as a private key, and a corresponding public key in $\mathbb{F}_p^\times$ that sets up the DLP for later.

Algorithm 4.2 (DSA Signature Generation)**.**
Input: m, the message; x, the private key.
Output: A signature (r, s) for the message m from the user.

(1) Define the reduction map $f : \mathbb{F}_p^\times \to \mathbb{F}_q$ defined by $x \mapsto x \pmod{q}$.
(2) Choose a random integer $k \in [1, q-1]$.
(3) Compute $r = f(g^k \pmod{p})$ If $r = 0$, choose new k.
(4) Compute $s = f\left(\frac{H(m)+xr}{k}\right)$ If $s = 0$, choose new k.
(5) Share (r, s) as the signature for the message m.

Remark. Adding the $H(m)$ term to the xr term makes the signature dependent on both the message and the user's private key in a way that is difficult to reverse but easy to check, as we will see next.

[1]A one-way function that takes an arbitrarily sized input and produces a random fixed-length numerical string as output.

Algorithm 4.3 (DSA Signature Verification)**.**
Input: m, the message; y, the public key; (r, s), the signature.
Output: Validity of the signature.

(1) Verify that $r, s \in [1, q-1]$, as they must be.
(2) Compute $u_1 = \frac{H(m)}{s} \pmod q$.
(3) Compute $u_2 = \frac{r}{s} \pmod q$.
(4) Compute $v = (g^{u_1} y^{u_2} \pmod p) \pmod q$. The signature is valid if and only if $v = r$.

Remark on Security. It can be mathematically proven that the above seemingly-arbitrary-looking algorithm accurately verifies correctly generated signatures.

First, from Algorithm 4.1, we defined $g = h^{(p-1)/q} \pmod p$. By Fermat's Little Theorem, this implies that $g^q \equiv h^{p-1} \equiv 1 \pmod p$. Thus g has order q, since q is prime.

In Algorithm 4.2, we computed

$$s = \frac{H(m) + xr}{k} \pmod q.$$

Therefore, solving for k gives

$$k \equiv \frac{H(m) + xr}{s} \equiv \frac{H(m)}{s} + \frac{xr}{s} \pmod q.$$

Since g has order q, we can raise it to the power of k and drop the (mod q) from above:

$$g^k \equiv g^{\frac{H(m)}{s} + \frac{xr}{s}} \equiv g^{\frac{H(m)}{s}} g^{\frac{xr}{s}} \pmod p.$$

Because $g^x = y$ by Algorithm 4.1, we can rewrite the latter term as $y^{\frac{r}{s}}$. Clearly now defining u_1 and u_2 as in Algorithm 4.3, this gives

$$g^k \equiv g^{u_1} y^{u_2} \pmod p.$$

In Algorithm 4.2, we defined $r = (g^k \pmod p) \pmod q$. Thus taking modulo q of both sides yields

$$r = (g^{u_1} y^{u_2} \pmod p) \pmod q$$

and, as we defined v to be exactly the right side of that equation in Algorithm 4.3, we have the equality that $r = v$. ∎

Past the additions, multiplications, and exponentiation, which are complicated but easy to undo for an attacker, an attacker aiming to uncover x—the private key—from the signature (r, s) must solve the DLP in $\mathbb{F}_p^\times$ to determine y and then what x is from $\log_g y = \log_g g^x = x$. Thus the DSA is, at its core, based on the DLP, and so we can modify it for use with the ECDLP.

4.2. **Elliptic Curve Digital Signature Algorithm.**

Now that we have seen the standard Digital Signature Algorithm in action, we can compare and contrast it against its elliptic curve analog.

Algorithm 4.4 (ECDSA Key Generation)**.**

(1) Choose a cryptographic hash function H with output length m bits.
(2) Choose a finite field $\mathbb{F}_q$ and an elliptic curve $E(\mathbb{F}_q)$ over this field.
(3) Choose a point G of prime order p on this curve.

(4) Save the domain parameters $(H, \mathbb{F}_q, E(\mathbb{F}_q), p, G, h)$ for distribution. h is defined as in Algorithm 3.4.
(5) For each individual user:
   (a) Choose a random integer $x \in [1, p-1]$. This is the private key; do not share it.
   (b) Compute $Y = xG$. This is the public key; share it!

Remark. ECDSA key generation also defines a hash function, a group of order p, and a generator for this group. The similarities to DSA should be clear; setting the public key $Y = xG$ sets up the ECDLP for later.

Algorithm 4.5 (ECDSA Signature Generation)**.**
Input: m, the message; x, the private key.
Output: A signature (r, s) for the message m from the user.

(1) Define the reduction map $f : E(\mathbb{F}_p) \to \mathbb{F}_p$ defined by $P \mapsto x(P) \pmod p$, where $x(P)$ is the x-coordinate of the point P.
(2) Choose a random integer $k \in [1, p-1]$.
(3) Compute $r = f(kG)$. If $r = 0$, choose new k.
(4) Compute $s = f\left(\frac{H(m)+xr}{k}\right)$. If $s = 0$, choose new k.
(5) Share (r, s) as the signature for the message m.

Remark. Note that this is practically identical to Algorithm 4.2, with the choice of reduction map being the only major difference. We can look at the x-coordinate only because $x(nP)$ depends only on $x(P)$; the x-coordinate is the easiest piece of critical information to compute, and stores the bulk of the data in practical elliptic curve cryptographical applications.

Algorithm 4.6 (ECDSA Signature Verification)**.**
Input: m, the message; Y, the public key; (r, s), the signature.
Output: Validity of the signature.

(1) Verify that $r, s \in [1, p-1]$, as they must be.
(2) Compute $u_1 = \frac{H(m)}{s} \pmod p$.
(3) Compute $u_2 = \frac{r}{s} \pmod p$.
(4) Compute $V = u_1 G + u_2 Y$. The signature is valid if and only if $r = f(V)$.

Remark on Security. Like the DSA, we can prove that the above algorithm accurately verifies correctly generated signatures. The reader is encouraged to pause here and attempt to derive the analogous proof from the proof of Algorithm 4.3.

First, from Algorithm 4.4, we have that $Y = xG$. Thus we have

$$V = u_1 G + u_2 x G.$$

By distributivity over addition, this is equivalent to

$$V = (u_1 + u_2 x)G.$$

From the definitions of u_1 and u_2 as in Algorithm 4.6, this expands to

$$V = \left(\frac{H(m) + xr}{s}\right) G.$$

From the definition of s in Algorithm 4.5, this becomes

$$V = \left(\frac{H(m) + xr}{H(m) + xr} \cdot k\right) G$$

which cancels to yield $V = kG$. Thus $f(V) = f(kG) = r$ as per Algorithm 4.5. ■

Once again, an attacker attempting to find the private key x from the signature (r, s) must solve the ECDLP in $E(\mathbb{F}_q)$ to determine $\log_G Y = \log_G xG = x$.

5. Comparisons

Elliptic curve cryptography (ECDH, ECDSA) provides the same level of security as asymmetric cryptography (resp. FFDH, DSA) with smaller key sizes; see Table 1 for the numbers. This reduces latency for elliptic curve cryptography users, as operations can be performed more quickly; it also reduces the amount of information transmitted, especially when using point compression to reduce bandwidth further (again, see Appendix A.2).

In general, the best known algorithms to crack the ECDLP have a complexity of about $O(\sqrt{n})$, where n is the order of the field. Thus for n bits of security, 2^n, we need a field with $(2^n)^2 = 2^{2n}$ elements and thus a key of length $2n$.

On the other hand, over $\mathbb{F}_p^\times$, the general number field sieve can be exploited to factor field entries and solve the DLP, so a much longer key is needed to make cracking classical cryptographical schemes computationally intractable.

Symmetric Key Length (Bit Security)	Asymmetric Key Length (i.e. RSA, DSA, etc.)	Elliptic Curve Key Length (i.e. ECDSA)
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	512

Table 1. Security comparison between elliptic curve-based and other cryptographic schemes.

Elliptic curve-based cryptosystems have been adapted from a variety of other classical cryptosystems like the Massey-Omura and El-Gamal systems, and they are being increasingly adopted in many modern cryptographic applications for their efficiency and security: LINE messaging app uses elliptic curve Diffie-Hellman for end-to-end encryption, and the Signal Protocol used by chat apps including Signal, WhatsApp, and Skype uses ECDH for post-compromise security—ensuring future messages are secure even if a previous message was compromised. Cryptocurrencies such as Bitcoin and Ethereum use ECDSA to sign transactions.

Unfortunately, elliptic curve cryptography is susceptible to attacks by quantum computers using Shor's algorithm for quantum integer prime factorization. While elliptic curve cryptography sees widespread use today, it may become obsolete in the near future in the face of leaps in quantum computing technology, and cryptosystems based on other intractable mathematical problems will rise in its place—only time will tell.

Appendix A. Notes on General Implementation

A.1. **Choice of Field.** There is a finite field $\mathbb{F}_q$ if and only if q is a power of a prime, and this field is unique up to isomorphism. Thus we could theoretically use any field with power-of-prime order, but we are restricted in cryptographic applications for practical reasons. Specifically, we are restricted in q to $q = p$ an odd prime, the prime fields, or $q = 2^m$ for some $m \geq 1$, the binary fields. This only serves to forbid fields with $q = p^m$ for $p \neq 2$.

The reasoning behind this is that power-of-prime fields have no practical reason for being implemented. Binary fields are naturally easy to implement due to computers' binary nature, and prime fields' elements, being simple integers, are also convenient to work with. Other power-of-prime fields are too inconvenient to implement to warrant their usage.

A.2. **Elliptic Curve Point Conversion.** When given a point $(x, y) \in E(\mathbb{F}_q)$, it is often necessary to convert it to a suitable data format and vice versa. Typically the output will be in the form of an octet string, comprised of octets of two hexadecimal digits each. There are two primary methods of converting curve points: with or without point compression.

In either case, the point at infinity $\mathcal{O}$ is converted to the string 00_{16}. Field elements of prime fields are converted using standard integer-to-octet-string conversion, while field elements of binary fields are converted by considering the coefficient string as a bit string and using bit-string-to-octet-string conversion.

Algorithm A.1 (With Point Compression)**.** Point compression works on the basis that there are at most two y-coordinates for points at any given x-coordinate; therefore, the only important information from the y-coordinate is its parity $\tilde{y}$. In a prime field, $\tilde{y} = y \pmod 2$, but in a binary field, if the x-coordinate is 0, $\tilde{y} = 0$; otherwise, set $\tilde{y}$ to the constant term z_0 of the binary polynomial z such that $z = \frac{y}{x}$.

Then the leading octet of the string is set to either 02_{16} or 03_{16} if $\tilde{y} = 0$ or 1, respectively. The x-coordinate is translated from a field element to an octet string and appended to the indicator octet to yield the converted octet string.

Algorithm A.2 (Without Point Compression)**.** Without point compression, the leading octet of the string is set to 04_{16} to denote that point compression is not being used. Then the x-coordinate and y-coordinate are translated from field elements to octet strings and concatenated to yield the converted octet string.

Converting data to an elliptic curve point can then be done by setting the leading octet to indicate whether point compression is being used and reversing the above algorithms.

A.3. **SEC Recommended Curves.** The Standards for Efficient Cryptography Group, or SECG, has several standards recommended for implementation of elliptic curve cryptography. They recommend the following for prime and binary fields:

A.3.1. *Prime Fields.* $\lceil \log_2 p \rceil \in \{192, 224, 256, 384, 521\}$ and elements should be integers.

A.3.2. *Binary Fields.* $m \in \{163, 233, 239, 283, 409, 571\}$ and elements should be of the set of binary polynomials of degree $m - 1$:

$$\left\{a_{m-1}x^{m-1} + \cdots + a_1x^1 + a_0 : a_i \in \{0, 1\}\right\}$$

with addition and multiplication done with respect to a reduction polynomial, a specially chosen irreducible binary polynomial of degree m for each field.

Appendix B. Failure Methods

It is worth mentioning a few ways in which these algorithms can fail or be exploited in practical applications.

B.1. **Choice of Curve.**

Some curves are unsuitable for use in cryptography. In particular, the set of nonsingular points on a singular curve $E(\mathbb{F}_q)$—one with repeated roots, or $\Delta = 0$ from Definition 2.1—form a group isomorphic to $\mathbb{F}_q$. Technically these are not *elliptic* curves, rather general cubic curves, but the same can happen over proper elliptic curves; the ECDLP can be lifted to the DLP over a corresponding finite field in some cases, rendering these curves insecure.

B.2. **ECDH Invalid Curve Attacks.**

ECDH is susceptible to attacks in which one party maliciously chooses invalid curve points, and when the other party does not verify that these points are indeed invalid, the malicious party is able to derive the other party's private key given enough time.

The general idea is that the attacker takes the elliptic curve equation and varies the constant parameter b to generate groups with arbitrary order, possibly divisible by small primes. For instance, suppose the attacker has a parameter b_3 such that the corresponding elliptic curve group has order divisible by 3; then, there exists a point P on this curve that generates a subgroup of order 3. There are only 3 possibilities for a multiple xP of this point, so by sending this to the other party and checking whether a valid response is received, the attacker can potentially determine xP and thus $x \pmod 3$. Doing this for enough small primes allows the attacker to apply the Chinese Remainder Theorem to compute the much larger private key x.

B.3. **DSA True Randomness.**

In both the DSA and ECDSA, the random integer k must be *truly* random, and no two messages can be signed with the same value of k. To see why, suppose the user with private key x signs two messages, m_1 and m_2, with the same value of k in ECDSA (equivalent for DSA with different math). This results in the signatures (r_1, s_1) and (r_2, s_2), where:

$$\begin{aligned} r_1 &= r_2 = f(kG) \\ s_1 &= \frac{H(m_1) + xr}{k} \pmod p \\ s_2 &= \frac{H(m_2) + xr}{k} \pmod p. \end{aligned}$$

Rearranging the last two equations for k and setting them equal, we have that

$$\frac{H(m_1) + xr}{s_1} = k = \frac{H(m_2) + xr}{s_2} \pmod p.$$

Therefore we have

$$xr(s_1 - s_2) = s_1 H(m_2) - s_2 H(m_1) \pmod q \implies x = \frac{s_1 H(m_2) - s_2 H(m_1)}{r(s_1 - s_2)} \pmod q.$$

Since everything on the right side of the last equation is known to the attacker, this allows for easy retrieval of the private key if the same value of k is reused.