J. Luke Scott

Information Security Basics
MiCS Program
University of Luxembourg

Final Project
Friday 15th June, 2012

Implementation of the SWU Elliptic-Curve Algorithm

1 Introduction

Elliptic curves are two-dimensional equations in the form $y^2 = x^3 + ax + b$. To be an elliptic curve, the discriminant of this equation must not be zero (mathematically: $-16(4a^3 + 27b^2) \neq 0$).

Graphing an elliptic curve reveals visually some of its representative traits (Figure 1). First, they are symmetric across the x axis. Any x value greater than x_{min} will correspond to exactly two points on the curve. Second, any line that intersects the curve will do so either at one point (this happens *only* when a vertical line intersects the curve at x_{min}), at two points (when a vertical line intersects at some other point), or at three points (under any other conditions).

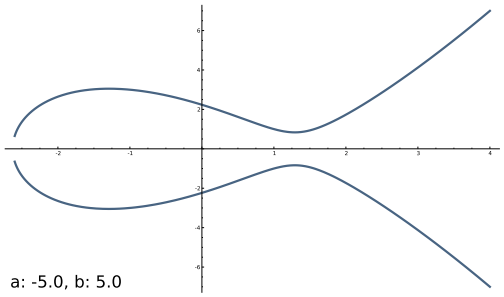

Figure 1: Simple example of an Elliptic Curve.

Other traits will be explained below; but for now, an appropriate introduction is to say that the points on an elliptic curve can be used as a set from which to build Abelian groups.

The rest of this paper is organized as follows: in section 2 and 3, I introduce using elliptic curves as Abelian groups and how point addition works; in section 4 I discuss how the properties of Abelian groups allow elliptic curves to be used for cryptography, and what kind of mathematical challenges and tricks arise, in section 5 I describe the SWU algorithm and its uses, and finally, section 6 concludes with a demonstrated use case for the SWU algorithm, using the NIST Curve P-192 published in [7].

2 Elliptic Curve Operations

Point addition, identified by the "dot" operator in Abelian groups ($\bullet$) is the most basic operation on elliptic curves. Graphically, adding points p and q means drawing a line through p and q, finding where this line intersects the curve at a third point, and then mirroring this point over the x axis. Figure 2 shows a line going from p through $2p$ to the point where it intersects the curve, which is then mirrored over the x axis to find the result $3p$.

Symmetrical points (points whose x values are the same) will have a vertical line through them, which will never meet the curve at a third point. Similarly, the tangent line through the point where the curve crosses the y axis will be vertical and will not cross the curve at a second point. In these cases, the point of intersection and the result of the addition operation is considered to be the "Point At Infinity", which functions as an identity element for the group (section 3).

Point doubling, not surprisingly, is adding a point to itself. Since there is no second point q to draw a line through, the line used when doubling p is considered to be the tangent at point p. Figure 2 shows point p being "doubled" by extending its tangent line to $-2p$ and then mirroring that point to the result $2p$.

Point multiplication, intuitively, is adding a point to itself n times. $2p = (p \bullet p)$, $5p = (p \bullet p \bullet p \bullet p \bullet p)$, etc. Point multiplication is a function applied to a point and a scalar, not two points. It is a mapping $P \times \mathbb{Z} \rightarrow P$, where $P \subseteq E_{a,b}(\mathbb{F}_q)$, not $P \times P \rightarrow P$.

3 Elliptic Curves as Abelian Groups

Elliptic Curves are Abelian Groups, as they exhibit all the necessary properties:

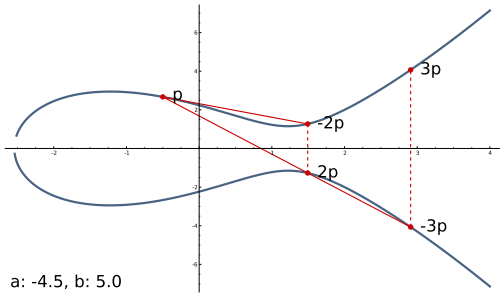

Figure 2: Point addition on an Elliptic Curve.

Group Operation: Point Addition The "Dot" (•) operation in Abelian groups can be any well-defined binary operation performed on members of a class whose result is another element of the same class. In the case of Elliptic Curves, this operation is "Point Addition" (section 2), performed on two points and returning a third point.

Closure The closure requirement in Abelian groups states that the result of the • operation applied to two members of a given group will always return a member of the same group. In Elliptic Curves, the closure property means that adding two points on a given curve will always return a point on the same curve. Of course, as described in sections 3 and 3, the set "points on the curve" must include the point at infinity.

Commutativity The • operation is commutative; that is, $p_1 \bullet p_2 = p_2 \bullet p_1$.

Associativity When performing two • operations on three points in series, order does not matter. The result of $p_1 \bullet (p_2 \bullet p_3)$ is equal to $(p_1 \bullet p_2) \bullet p_3$.

Identity Element Elliptic curves have an additive identity element, known as the "Point at Inifinity" (∞). Adding this to any other point on a curve will result in the same point on the curve.

Inverse: P + -P = I Similar to the additive identity property described above, there exists an Inverse Element p' for each point p, such that $p \bullet p' = \infty$. In the case of Elliptic Curves, this point p' is the "negative" of p. In more practical terms, this means that p' is p reflected across the x axis; $p' = (p_x, -p_y)$ where $p = (p_x, p_y)$.

4 Using Elliptic Curves for Cryptography

The cryptographic premises of Elliptic Curve Cryptography (ECC) are that elliptic curves can be used as Abelian groups, and that the point addition and multiplication operations provide security through the "trap door" mechanism (section 4.2).

In ECC, performing efficient addition and multiplication operations is a key consideration. A convenient algorithm for doing point multiplication is the double-and-add algorithm:

```
def doubleAndAdd(p, n):
    binaryString = bin(n)[2:]
    q = p
    for i in range(1, len(binaryString)):
        q = addPoints(q, q) # double
        if binaryString[i] == '1':
            q = addPoints(q, p) # add
    return q
```

Other efficient algorithms exist, but this is an extremely simple, intuitive technique.

However, one reason not to use the double-and-add algorithm is its vulnerability to side-channel attacks: a sophisticated attacker could gain information about the value of p or n by measuring the amount of time or energy it takes to perform the calculation. One technique which can resist this type of attack is Montgomery's Ladder [6], a modification of the double-and-add algoritm that always performs the same number of calculations and therefore uses constant power regardless of inputs.

4.1 Modular Elliptic Curve Operations

Because elliptic curve cryptography uses curves over finite fields $\mathbb{F}_q$, mathematical operations modulo q are used (Figure 3). Figure 4 and 5 show that the equivalent point addition and multiplication operations can be done on modulo (finite field) and non-modulo curves, but that the visual intuition breaks down when using finite fields.

All of the important characteristics of Abelian groups still hold, and concepts are all equivalent, with the exception that modulo operations are inherently difficult to "undo". In this paper, I use the standard continuous-curve diagrams whenever possible, even if the concept being illustrated is in practice done on

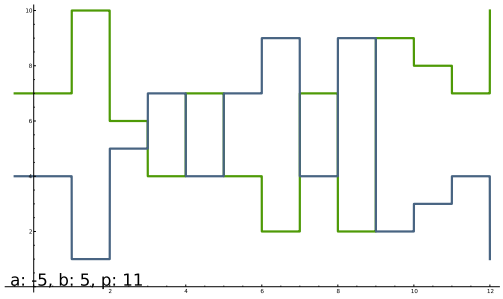

Figure 3: Elliptic Curve using integers mod p (negative values from the original non-modulo curve are shown in green).

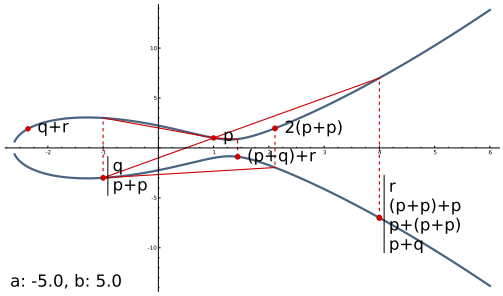

Figure 4: Point addition operations performed on a non-modulo elliptic curve.

finite fields. (Figures 6-9, etc.)

4.2 Discrete Logarithm Problem

RSA and other public-key cryptography schemes use the problem of factoring large numbers to provide security [8]. Elliptic curve cryptography uses the Discrete Logarithm problem: given the result of point addition and multiplication operations, it is prohibitively hard for an attacker to know the inputs to those operations [3]. This provides the "trap door" mechanism needed in cryptographic processes: the party performing the process knows how to generate a result in the forward direction, but an attacker cannot discover the key(s), clear text, or other inputs used to generate that result without prior knowledge.

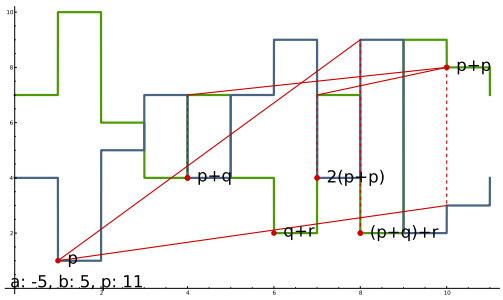

Figure 5: The same point addition operations as in Figure 4 performed on a modulo (finite field) elliptic curve.

4.3 Benefits Over Other Cryptographic Methods

A benefit of elliptic curve cryptographic methods is the key size required. For a given level of security, ECC generally requires much smaller keys than other public-key schemes [1, 5]. The security of a protocol is measured in bits, where the value represents the speed of the fastest known attack, in steps it will take an attacker to break a key. Public-key schemes are known to require larger keys than in symmetric schemes, due to the fact that, by design, part of the key data used to exchange information is public knowledge. The security of such a system is derived from mathematical properties of these combined public/private keys, whereas the security of a private key scheme is derived in large part from the fact that the key(s) are private. In private schemes, the only publicly visible information is the encrypted ciphertext itself; there is no publicly-known key to compare with some ciphertext from which to derive knowledge about the original key.

This trait of private schemes means that their key lengths are generally the same as their cryptographic strength. To achieve 80-bit security, for instance, one would require an 80-bit private key system. But the equivalent strength using public-key encryption would require much larger keys. DSA/RSA schemes require a key length of around 1,500 for the same level of security. However, the extra "cost" of the publicly-shared key as compared with private schemes is much less with ECC than with other public-key schemes such as RSA. Only around 150 bits are needed to achieve 80-bit security using ECC. (These are the recommended appropriate levels for 2012 in [5]).

4.4 What Can We Do With Elliptic Curves?

A secure key exchange can be done using only the properties of Abelian groups and a suitable hash function [2]:

- Assume a shared password pw between Alice and Bob, a shared hash function $H(x)$, and se-

cret keys k_a and k_b, where $k_a, k_b, H(x) \in \mathbb{Z}_q$.

- Alice sends $A = k_a \bullet H(pw)$ to Bob. By the closure property, $A \in \mathbb{Z}_q$.
- Bob sends $B = k_b \bullet H(pw)$ to Alice. By the closure property, $B \in \mathbb{Z}_q$.
- Alice computes $K_a = k_a \bullet B = k_a \bullet (k_b \bullet H(pw))$.
- Bob computes $K_b = k_b \bullet A = k_b \bullet (k_a \bullet H(pw))$.
- Because of the commutative property of Abelian groups, $k_a \bullet k_b = k_b \bullet k_a$.
- This leads directly to $K = K_a = K_b$, a shared private key:

$$\begin{aligned} K = K_a &= k_a \bullet B \\ &= k_a \bullet k_b \bullet H(pw) \\ &= k_b \bullet k_a \bullet H(pw) \\ &= k_b \bullet A \\ &= K_b \end{aligned}$$

Because elliptic curves can be used as Abelian groups (section 3), they can be leveraged to perform this algorithm, where the $\bullet$ operator in the algorithm is point addition as described in section 3. The algorithm as it is described above uses the group $(\mathbb{Z}_q, \bullet)$, but it could easily be replaced by $E_{a,b}(\mathbb{F}_q, \bullet)$ where $\mathbb{F}_q$ is the field of points on an elliptic curve mod p, $\bullet$ is the point addition operation, and $E_{a,b}(F)$ denotes an elliptic curve using characteristic parameters a and b over the field F. Also, the hash function $H(x)$ must be changed from $H(x) : X \rightarrow \mathbb{Z}_q$ to $H(x) : X \rightarrow E_{a,b}(\mathbb{F}_q)$ for a given a and b. If this hash function is known then the process is simple:

- Alice sends $A = k_a \bullet H(pw)$ to Bob. By the closure property, $A \in E_{a,b}(\mathbb{F}_q)$ (Figure 6).
- Bob sends $B = k_b \bullet H(pw)$ to Alice. By the closure property, $B \in E_{a,b}(\mathbb{F}_q)$ (Figure 7).
- Alice computes $k_a \bullet B = K_a \in E_{a,b}(\mathbb{F}_q)$ (Figure 8).
- Bob computes $k_b \bullet A = K_b \in E_{a,b}(\mathbb{F}_q)$ (Figure 9).
- Again, $K = K_a = K_b$, a shared private key. But now, $K \in E_{a,b}(\mathbb{F}_q)$.

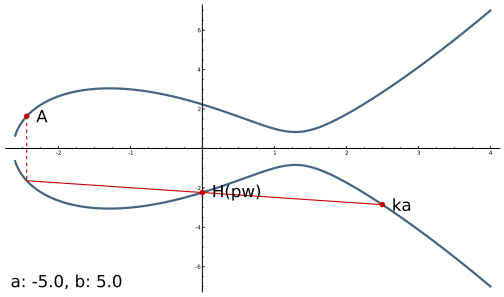

Figure 6: Alice sends $A = k_a.H(pw)$ to Bob

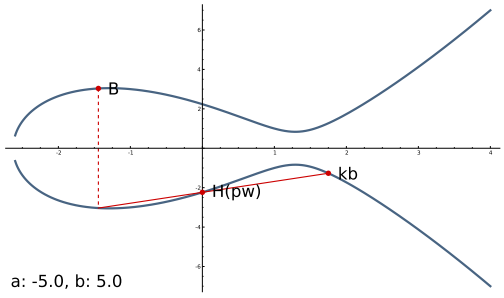

Figure 7: Bob sends $B = k_b.H(pw)$ to Alice

There is one caveat here: Figures 6-9 show the point addition operations being performed on a curve over $\mathbb{R}$ because of its visual accessibility, but in practice, the field $\mathbb{F}_q$ is used.

Other uses for elliptic curves are similar to those used on normal finite fields:

A public-key encryption scheme, Integrated Encryption Scheme (IES), on elliptic curves is similar to the RSA protocol: a prime number is used for a group modulo operand, a group order n, generator G, and hash function are known for the group, plus the characteristic parameters a and b are used to define the curve over the finite field.

Elliptic curves can be used for digital signatures as well using the ECDSA algorithm [10].

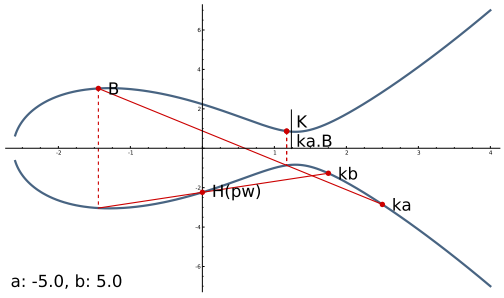

Figure 8: Alice computes $K = k_a.(k_b.H(pw))$

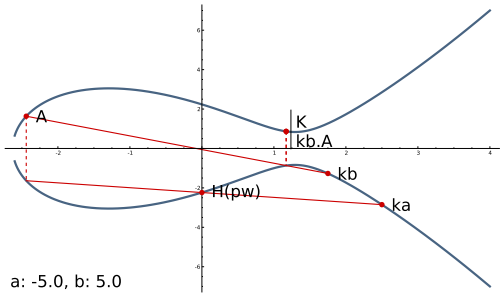

Figure 9: Bob computes $K = k_b.(k_a.H(pw))$

5 The SWU Algorithm

In section 4.4 I mentioned that, given a hash value $H(x) \in E_{a,b}(\mathbb{F}_q)$, a secret key exchange could be performed. But how do we get a hash function $H(x) : X \rightarrow E_{a,b}(\mathbb{F}_q)$, with X some class that is useful to us, such as $\mathbb{Z}_q$, or better yet, clear text?

Here I describe the Shallue-Woestijne-Ulas (SWU) algorithm, first published in [9]. It is exactly this hash function that I have described the need for: $SWU(x) : \mathbb{F}_q \rightarrow E_{a,b}(\mathbb{F}_q)$. Here, I will describe the SWU algorithm using the mapping $\mathbb{F}_q \rightarrow E_{a,b}(\mathbb{F}_q)$, although it can be used to hash any scalar or plaintext value into an elliptic curve by using the result of an appropriate pre-hash function $H(x) : X \rightarrow \mathbb{F}_q$, where X is an arbitrary class of values, as the input to the SWU algorithm.

For instance, hashing a plaintext passphrase requires a function $H(T) : T \rightarrow E_{a,b}(\mathbb{F}_q)$, where T might be the set of all ASCII or Unicode strings. Assembling such a hash function is as simple as using the following hash function composition:

$$SWU(SHA1(x)) : T \rightarrow \mathbb{F}_q \rightarrow E_{a,b}(\mathbb{F}_q).$$

The SWU algorithm is based on Simplified Ulas Maps:

$$U(t)^2 = -g\big(X_2(t)\big) \cdot g\big(X_3(t)\big),$$

where, for some $a, b \neq 0$:

$$g(x) = x^3 + ax + b,$$

$$X_2(t) = \frac{-b}{a}\Big(1 + \frac{1}{t^4 - t^2}\Big),$$

$$X_3(t) = -t^2 X_2(t),$$

and

$$U(t) = t^3 g(X_2(t)).$$

Because $U(t)^2 = -g\big(X_2(t)\big) \cdot g\big(X_3(t)\big)$, and because -1 cannot be a square, we know that either $g\big(X_2(t)\big)$ or $g\big(X_3(t)\big)$ must be a square. By simply finding which one has a square root mod p, and then calculating the appropriate $\sqrt{g(f(t))}$, we are able to create a unique point on a curve using irreversible operations.

5.1 Quadratic Residue

Determining whether a value has a square root mod q, then, becomes important. Thankfully, it is a simple matter of employing the Legendre Symbol calculation first published in [4] – I use the notation l_x for brevity:

$$l_x = \Big(\frac{x}{q}\Big) = x^{\left(\frac{p-1}{2}\right)}$$

For a prime q, $q \neq 2$, $l_x = 1$ if $x \neq 0$ and is a square modulo q, $l_x = 0$ if $x = 0$ modulo q, and $l_x = -1$ if $x \neq 0$ and is not a square modulo q. So by calculating l_x for $x = g\big(X_2(t)\big)$ and $x = g\big(X_3(t)\big)$, we will know which of the two has a square root modulo q.

Once we know which $g\big(f(t)\big)$ has a square root modulo q, we can use it to create a point $\Big(f(t), \sqrt{g\big(f(t)\big)}\Big)$ on $E_{a,b}(\mathbb{F}_q)$.

5.2 Square Roots mod q

We have determined what value two potential hashed points would have, and we have determined which of the two points we will use. One final step in creating this point is to actually calculate the square root modulo q of $g\big(f(t)\big)$.

Conveniently, when our prime number q is equal to 3 mod 4, we can compute a square root mod q very easily:

$$\sqrt{x} \bmod q = x^{\left(\frac{(q+1)}{4}\right)}.$$

5.3 Putting it Together

By incorporating this final technique into the hashing algorithm, we arrive at:

$$SWU(t) = \Big(X_2(t), g\big(X_2(t)\big)^{\frac{(q+1)}{4}}\Big)$$

if $g(X_2(t))$ is a square, or

$$SWU(t) = \left(X_3(t), g\left(X_3(t)\right)^{\frac{(q+1)}{4}}\right)$$

otherwise (because $g\left(X_3(t)\right)$ is a square if $g\left(X_2(t)\right)$ is not).

6 Application and Conclusion

For this paper, I have written code to demonstrate the use of the SWU algorithm to perform a simple key exchange (see the Appendix, section 7). By using a plaintext passphrase as the only privately shared information between the two sides before the exchange, as well as the common knowledge outlined in section 4.4, I have demonstrated that the algorithm does indeed allow two parties to arrive at a common shared secret point on an elliptic curve suitable for use in ECC cryptographic functions.

At a lower level of detail, one also sees that this script demonstrates the steps of first using SHA-1 to map from clear text to $\mathbb{Z}_q$, and using the SWU algorithm as a map from $\mathbb{Z}_q$ to $E_{a,b}\mathbb{F}_{q'}$. The second step, incidentally, includes an inherent re-assignment of $t \in \mathbb{Z}_q$ (where q is the maximum value of a SHA-1 hash) to $(t \bmod q') \in \mathbb{Z}_{q'}$ (where q' is that defined for NIST curve P-192) and therefore $(t \bmod q') \in \mathbb{F}_{q'}$ – the domain of the elliptic curve. Also one can see in the example output that the SWU algorithm decides which of two potential results to return: that derived from X_2 or X_3. Using this generated shared secret, Alice and Bob are able to exchange encrypted messages using a symmetric protocol.

In conclusion, I have shown the underlying mathematical theory of Elliptic Curve Cryptography based on Abelian Groups, I have explained the advantage that ECC has over other cryptographic methods, and I have demonstrated the use of the SWU algorithm for generating points on an elliptic curve and subsequently a shared secret between two parties.

7 Appendix

Figure 11 shows the execution output of a Python script demonstrating a SPEKE session in which a private shared key is generated. Figure 12 shows the source code for the sample script. (All source code will be submitted along with this paper).

References

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

[2] D.P. Jablon. Strong password-only authenticated key exchange. *ACM SIGCOMM Computer Communication Review*, 26(5):5–26, 1996.

[3] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[4] A.M. Legendre. *Essai sur la theorie des nombres*. Chez Duprat, 1808.

[5] A.K. Lenstra and E.R. Verheul. Selecting cryptographic key sizes. *Journal of cryptology*, 14(4):255–293, 2001.

[6] P.L. Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987.

[7] National Institute of Standards and Technology (NIST). Federal information processing standards (fips) pub 186-3. *Digital Signature Standard (DSS), Issued June*, 2009.

[8] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[9] A. Shallue and C. van de Woestijne. Construction of rational points on elliptic curves over finite fields. *Algorithmic number theory*, pages 510–524, 2006.

[10] S Vanstone. Responses to nist's proposal. *Communications of the ACM*, 35:50–52, 1992.

```
def swuHash(self, t):
    self.out.indent("SWU Hash Algorithm starting...")
    alpha = -(t * t)
    alphaDenominator = mod(alpha ** 2 + alpha, self.p)
    x2 = -(self.b / self.a) * (1 + 1 / alphaDenominator)
    h2 = mod(x2 ** 3 + self.a * x2 + self.b, self.p)
    exponent = (self.p + 1) / 4

    if self.legendre(h2) == 1:
        self.out.put("Found g(X2(t)) is a perfect square.")
        self.out.unIndent()
        return (x2, h2 ** exponent)

    self.out.put("Found g(X3(t)) is a perfect square.")
    x3 = alpha * x2
    h3 = x3 ** 3 + self.a * x3 + self.b
    self.out.unIndent()
    return (x3, h3 ** exponent)

def legendre(self, x):
    # Legendre symbol:
    # 1 if x is Quadratic Residue mod p
    # 0 if x == 0 mod p
    # -1 otherwise
    exponent = ((self.p - 1) / 2)
    return x ** exponent
```

Figure 10: Listing of the SWU algorithm written in Python.

```
Loading Sage...
Starting /home/luke/Projects/cryptoFinalProject/python/SpekeSwu.py...

Alice and Bob are both generating h(pw) from passphrase: 'a cow jumped over the moon'...
    integer representation of sha1(pw):
                          370209994692993959654855452181727349094605495589
    SWU Hash Algorithm starting...
        Found g(X2(t)) is a perfect square.
    h(pw) = SWU(sha1(pw)):                  (3.2e+57, 2.1e+57)

Alice is generating a private key using a random int 730...
    SWU Hash Algorithm starting...
        Found g(X3(t)) is a perfect square.
    Alice's private key:                    (2.5e+57, 1.3e+57)
    EllipticCurve.addPoints
        p1:                                     (2.5e+57, 1.3e+57)
        p2:                                     (3.2e+57, 2.1e+57)
        returning:                              (2.1e+56, 4.5e+56)
    Alice generated public key A = a.h(pw): (2.1e+56, 4.5e+56)

Bob is generating a private key using a random int 1202...
    SWU Hash Algorithm starting...
        Found g(X2(t)) is a perfect square.
    Bob's private key:                      (1.2e+56, 2.5e+57)
    EllipticCurve.addPoints
        p1:                                     (1.2e+56, 2.5e+57)
        p2:                                     (3.2e+57, 2.1e+57)
        returning:                              (4e+57, 1.4e+57)
    Bob generated public key B = b.h(pw):   (4e+57, 1.4e+57)

Bob and Alice are exchanging keys...
    EllipticCurve.addPoints
        p1:                                     (2.5e+57, 1.3e+57)
        p2:                                     (4e+57, 1.4e+57)
        returning:                              (4.1e+57, 4.4e+57)
    Alice generated Ka = a.(b.(h(pw))):     (4.1e+57, 4.4e+57)
    EllipticCurve.addPoints
        p1:                                     (1.2e+56, 2.5e+57)
        p2:                                     (2.1e+56, 4.5e+56)
        returning:                              (4.1e+57, 4.4e+57)
    Bob generated Kb = b.(a.(h(pw))):       (4.1e+57, 4.4e+57)
Keys match! Alice and Bob now have a new shared secret!
Done.
```

Figure 11: Output of a sample SPEKE session showing Alice and Bob generating a shared private key based on two randomly-generated private keys and a shared passphrase.

```
#!/usr/bin/sage
from EllipticCurve import EllipticCurveModRing

from hashlib import sha1
from random import randint, seed

class SpekeSwu(EllipticCurveModRing):
    KNOWN_PASSPHRASE = "a cow jumped over the moon"

    # NIST Curve P-192 (ref: FIPS PUB 186-3)
    p = 6277101735386680763835789423207666416083908700390324961279
    n = 6277101735386680763835789423176059013767194773182842284081
    a = -3
    b = 0x64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1
    Gx = 0x188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012
    Gy = 0x07192b95ffc8da78631011ed6b24cdd573f977a11e794811

    def createEllipticCurve(self):
        seed()
        self.out.indent("\nAlice and Bob are both generating h(pw) from passphrase: '%s'..." %
                        self.KNOWN_PASSPHRASE)

        sha1Hash = sha1()
        sha1Hash.update(self.KNOWN_PASSPHRASE)
        sha1HashInt = int(sha1Hash.hexdigest(), 16)
        self.printNumber("integer representation of sha1(pw)", sha1HashInt)

        self.h_pw = self.swuHash(sha1HashInt)
        self.printPoint("h(pw) = SWU(sha1(pw))", self.h_pw)
        self.out.unIndent()

        t = randint(0, 10000)
        self.out.indent("\nAlice is generating a private key using a random int %d..." % t)
        self.alice = self.swuHash(t)
        self.printPoint("Alice's private key", self.alice)
        self.a_h_pw = self.addPoints(self.alice, self.h_pw)
        self.printPoint("Alice generated public key A = a.h(pw)", self.a_h_pw)
        self.out.unIndent()

        t = randint(0, 10000)
        self.out.indent("\nBob is generating a private key using a random int %d..." % t)
        self.bob = self.swuHash(t)
        self.printPoint("Bob's private key", self.bob)
        self.b_h_pw = self.addPoints(self.bob, self.h_pw)
        self.printPoint("Bob generated public key B = b.h(pw)", self.b_h_pw)
        self.out.unIndent()

        self.out.indent("\nBob and Alice are exchanging keys...")
        self.ka = self.addPoints(self.alice, self.b_h_pw)
        self.printPoint("Alice generated Ka = a.(b.(h(pw)))", self.ka)

        self.kb = self.addPoints(self.bob, self.a_h_pw)
        self.printPoint("Bob generated Kb = b.(a.(h(pw)))", self.kb)
        self.out.unIndent()

        if self.ka[0] == self.kb[0] and self.ka[1] == self.kb[1]:
            self.out.put("Keys match! Alice and Bob now have a new shared secret!")
        else:
            self.out.put("ERROR: something went wrong.")

if __name__ == "__main__":
    SpekeSwu().run()
```

Figure 12: Source code for SPEKE sample.