Modern elliptic curve cryptography

Ivo Kubjas

1 Introduction

Elliptic curve cryptography has raised attention as it allows for having shorter keys and ciphertexts. For example, to obtain similar security levels with 2048 bit RSA key, it is necessary to use only 256 bit keys using over elliptic curve cryptography.

Additionally, developments in the index calculus method for solving a discrete logarithm problem increases the sizes of the keys to keep the security requirements. However, these methods do not apply to points on the elliptic curves, allowing better estimations for security levels over longer period of time.

Up to now, only a small family of elliptic curves have been widely used for cryptographic purposes. However, as the construction of these curves have been lead by large intelligence agencies, skepticism has been rising against using these families for different purposes as the may contain unknown backdoors.

These concerns can be understood as getting arithmetic right on these families is difficult and developer can add security bugs inadvertently.

In this report we study different elliptic curve formulas, concentrating on Edward curve. Furthermore, we study the feasibility of implementing such curves on embedded hardware devices and discuss optimization methods for obtaining reasonable performance rates.

2 Construction of elliptic curve groups

Let $K$ be a finite field.

Definition 1 ([7]). An elliptic curve $E$ over a field $K$ is defined by an equation

$$E : y^2 + a_1 xy + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6,$$

(1)

where $a_1, a_2, a_3, a_4, a_6 \in K$ and $\Delta \neq 0$, where $\Delta$ is the discriminant of $E$ and is defined as follows:

$$\Delta = -d_2^2 d_8 - 8 d_4^3 - 27 d_6^2 + 9 d_2 d_4 d_6,$$

$$d_2 = a_1^2 + 4 a_2,$$

$$d_4 = 2 a_4 + a_1 a_2,$$

$$d_6 = a_3^2 + 4 a_6,$$

$$d_8 = a_1^2 a_6 + 4 a_2 a_6 - a_1 a_3 a_4 + a_2 a_5 - a_4^2.$$
If $L$ is any extension field of $K$, then the set of $L$-rational points on $E$ is the set:

$$E(L) = \{(x,y) \in L \times L : y^2 + a_1xy + a_3y - x^3 - a_2x^2 - a_4x - a_6 = 0\} \cup \{\infty\},$$

where $\infty$ is the point at infinity.

From now on, we denote the points on the curve with capital letters and the field elements with smaller letters.

**Remark 1.** As this formula is complete only if the coefficients $a_1, a_2, a_3, a_4, a_6$ are elements of $K$, then it is common to also write $E/K$. The field $K$ is called the underlying field of $E$.

If $E$ is defined over $K$, then the same equation is correctly defined for any extension field of $K$.

**Remark 2.** The condition $\Delta \neq 0$ ensures that the elliptic curve does not have points where there are more than one tangent lines. If there are several tangent lines, then as we will see later, the addition would not be uniquely determined.

### 3 Elliptic curve formulas

#### 3.1 Short Weierstrass curve

Given the elliptic curve formula (1), if the coefficients are set as $a_1 = 0$, $a_2 = 0$, $a_3 = 0$, $a_4 = a$ and $a_6 = b$, we obtain the curve formula

$$E : y^2 = x^3 + ax + b.$$  \hfill (2)

This form is called short Weierstrass form.

Furthermore, in the FIPS 186-4 [11], the parameter $a$ is fixed as $a \equiv -3 \mod p$ if the underlying field is $\mathbb{F}_p$. This choice for $a$ allows for efficient implementation of point arithmetic. The standard refers to IEEE Standard 1363-2000 [8], but the latter standard does not present any comparison between the effects and instead refers to a paper by Chudnovsky et Chudnovsky [6]. In [6], a proof was given that this restriction allows a 11% decrease in the number of operations for adding two points.

Equation formulas of this type are used in different standardized curves.

#### 3.1.1 Arithmetic on short Weierstrass curves

The point addition on elliptic curves over the finite fields is similar to the point addition of the elliptic curves over the reals. The addition of the points $P$ and $Q$ is performed as follows:

1. Draw a tangent line through $P$ and $Q$.
2. Find the intersection of the tangent line and the elliptic curve and take the point $R$ on the intersection.
<table>
<thead>
<tr>
<th>Name</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIST P-192</td>
<td>2455155546008943817740293915197451784769108058161191238065</td>
</tr>
<tr>
<td>NIST P-224</td>
<td>189582862855666080004086685444932641550468096867932107578</td>
</tr>
<tr>
<td>NIST P-256</td>
<td>4105836372515214212932612978004726840911444101599372555483</td>
</tr>
<tr>
<td>NIST P-384</td>
<td>27580193559959705787784901184038904809305690585636156852142</td>
</tr>
<tr>
<td>NIST P-521</td>
<td>1093849038073734274511112390766805669936207598951683748994</td>
</tr>
</tbody>
</table>

Figure 1: Coefficient \( b \) for different standardized short Weierstrass curves. Curves denoted in bold are suggested by NSA for securing SECURE and TOP SECURE level documents. As NIST P-521 does not provide any reasonable additional security, then its use is not recommended.

3. Mirror the point \( R \) over the \( x \)-axis to obtain \( P + Q \).

In the cases where the tangent line is not uniquely defined (the points to be added coincide) or when the tangent line does not intersect the curve elsewhere (the \( x \)-coordinates of the points coincide), it is not possible to use this definition. In general, if \( P = (x_1, y_1) \) and \( Q = (x_2, y_2) \), then:

- if \( P = \infty \), then \( P + Q = Q \)
- if \( Q = \infty \), then \( P + Q = P \)
- if \( x_1 = x_2 \) and \( y_1 = -y_2 \), then \( P + Q = \infty \)
- if \( P \neq Q \), then \( \lambda = \frac{y_2 - y_1}{x_2 - x_1} \) and
  \[ P + Q = \left( \lambda^2 - x_1 - x_2, \lambda(x_1 - x_3) - y_1 \right) \]
- if \( P = Q \), then \( \lambda = \frac{3x_1^2 + a}{2y_1} \) and
  \[ P + Q = \left( \lambda^2 - x_1 - x_2, \lambda(x_1 - x_3) - y_1 \right) \]

The algorithm for adding two points on the short Weierstrass curve over a finite field is defined in [12]. In Algorithm [1], we present the algorithm in a simplified form. This simplified form follows the general construction of the initial algorithm but discards the representation-specific operations. The inputs are points \( S = (S_x, S_y, S_z) \). If \( S_z = 0 \), then \( S_x = S_y = 1 \) and the point \( S \) is said to be infinity (\( \infty \)). Otherwise, the affine coordinates of the point are \((S_x/S_z, S_y/S_z)\).
Algorithm 1 Algorithm for point addition

1: procedure ECADD(S, T)
2:     if $S_z = 0$ then
3:         return $T$
4:     else if $T_z = 0$ then
5:         return $S$
6:     else if $S_x = T_x$ then
7:         if $S_y = T_y$ then
8:             return DOUBLE($S$)
9:         else
10:             return $\infty$
11:     end if
12: else
13:     return GENERALADD($S, T$)
14: end if
15: end procedure

The GENERALADD procedure performs optimized addition of two points in the case when the points are different and they are not the points at infinity. The DOUBLE procedure doubles the input in an optimized way. As the latter algorithms use specific representation of the points, then we won’t describe them in detail.

However, there are several drawbacks in regards to ECADD and GENERALADD:

- neither of the functions check if the points which are added are on the defined curve. Furthermore, neither of them even includes the parameter $b$ of the curve in the procedures and so it is not possible to construct any tests which consider the parameter. In [5] it was described, how it allows for extracting information about the secret key.

- both of the functions contain conditional checks, thus it is possible to gain side information about the values used in the addition by measuring the time required for performing the operations.

3.2 Koblitz curve

Fixing $a_1 = 1$, $a_2 = a$, $a_3 = 0$, $a_4 = 0$, $a_6 = 1$ in the general curve formula (1), we obtain a formula

$$E : y^2 + xy = x^3 + ax^2 + 1,$$

(3)

which is also called Koblitz form of an elliptic curve formula. This form is used in the standards if the underlying fields is an extension field of $\mathbb{F}_2$. Because these forms are not widely used, we do not specify the parameters explicitly, as the parameters are easily found in [11].
3.3 Edward curve

The Edward curve is defined as

\[ E : ax^2 + y^2 = 1 + dx^2y, \]

(4)

with \( d \not\in \{0,1\} \).

There are not many elliptic curve groups which are defined by Equation (4) but Curve25519 is very widely known [3]. The addition formula for points \( X = (x_1, x_2) \) and \( Y = (y_1, y_2) \) is

\[ X + Y = \left( \frac{x_1y_2 + x_2y_1}{1 + dx_1x_2y_1y_2}, \frac{x_2y_2 - ax_1y_1}{1 - dx_1x_2y_1y_2} \right) \]

(5)

Compared to Algorithm 1, the addition formula is straightforward, constant time and depends on the curve parameters. Because the addition is rather straightforward, then there exists smaller chance for faulty implementation. Furthermore, if \( d \) is non-square, then the values \( 1 \pm dx_1x_2y_1y_2 \) are not zero for any \( X \) and \( Y \).

3.4 Montgomery curve

The Montgomery formula for defining elliptic curve is:

\[ E : by^2 = x^3 + ax^2 + x, \]

(6)

such that \( b(a^2 - 4) \neq 0 \).

The addition formula for the points \( X \) and \( Y \) is

\[ (x_1, x_2) + (y_1, y_2) = \left( \frac{b(y_1x_2 - x_1y_2)^2}{x_1y_1(y_1 - x_1)^2}, \frac{(2x_1 + y_1 + a)(y_2 - x_2)}{y_1 - x_1}, \frac{b(y_2 - x_2)^3}{(y_1 - x_1)^3} - x_2 \right) \]

(7)

Even though the addition has similar properties as for points on Edward curve, the computational complexity is larger.

3.5 Scalar multiplication

As the elliptic curve forms an additive group, then it is natural to define a scalar multiplication of a scalar and a point. The scalar multiplication is analogous to exponentiation over multiplicative groups.

The scalar multiplication leads to a difficult problem similar to solving a discrete logarithm problem in a multiplicative group.

For multiplication, it is possible to use two algorithms. The first algorithm is a generalization of square-and-multiply algorithm for the points on an elliptic curve, called double-and-add algorithm. The algorithm is given in Algorithm 2.

NSA recommends [12] a slightly modified version of the algorithm, where the addition is hidden using another subtraction which is performed when addition...
Algorithm 2 Double-and-add scalar multiplication algorithm

1: procedure ScalMult(x, T)
2:  \( V = \infty \)  \( \triangleright \) \( \infty \) is the identity
3:  Let \( x = (x_0, \ldots, x_n) \) be the binary representation
4:  for \( i = 0, i \leq n \) do
5:      if \( x_i = 0 \) then
6:         \( V = \text{ECADD}(V, T) \)
7:      end if
8:  \( T = \text{DOUBLE}(T) \)
9:  end for
10: return \( V \)
11: end procedure

is not performed. Then, depending on the branch taken, the intermediate value
is copied or not. We see, that the steps taken during the scalar multiplication
depends on the intermediate values and thus is not constant time.

Another approach is to use the Montgomery ladder [10], which can be mod-
ified to perform constant time operations [9]. The algorithm is given in Algo-

Algorithm 3 Montgomery ladder

1: procedure ScalMult(x, T)
2:  \( V_0 = \infty, V_1 = T \)
3:  Let \( x = (x_0, \ldots, x_n) \)
4:  for \( i = 0; i \leq n \) do
5:      \( V_{1-x_i} = V_0 + V_1 \)
6:      \( V_{x_i} = \text{DOUBLE}(V_{x_i}) \)
7:  end for
8: return \( V_0 \)
9: end procedure

Even though the number of steps is constant, low-level hardware features de-
terminate the exact timing information, which must be mitigated using platform-
specific tools.

4 Conversion between different curve formulas

In [4], transformations are given to represent curves through other formulas
when necessary conditions are satisfied.

If we take
\[
x = bu - a/3 \\
y = bv,
\]
in Montgomery curve equation 6, we obtain

\[ v^2 = x^3 + 3 - \frac{a^2}{3b^2}u + \frac{2a^3 - 9a}{27b^3} , \]

which corresponds to the short Weierstrass formula.

Also, if we take

\[
\begin{align*}
x &= u/v \\
y &= \frac{u - 1}{u + 1},
\end{align*}
\]

in Edward curve equation 4, we obtain

\[
\frac{4}{1 - d}v^2 = u^3 + \frac{2(1 + d)}{1 - d}u^2 + u,
\]

which corresponds to the Montgomery formula. The Montgomery formula can now further be converted to short Weierstrass formula.

5 ANSI X9.62 representation of points on Weierstrass curve

As only the Weierstrass formula has been specified in the standards, then only the points on such curves are defined for being transferred in canonical form. For complete reference, we guide the readers to ANSI X9.62 standard [2]. We have extracted the relevant definitions:

```plaintext
SubjectPublicKeyInfo ... = SEQUENCE {
   algorithm AlgorithmIdentifier,
   subjectPublicKey BIT STRING
}

AlgorithmIdentifier ... = SEQUENCE {
   algorithm ALGORITHM,
   parameters ECParameters
}

ECParameters ... = SEQUENCE {
   -- Elliptic curve parameters
   version INTEGER,
   fieldID FieldID,
   curve Curve,
   base ECPair, -- Base point G
   order INTEGER, -- Order n of the base point
   cofactor INTEGER OPTIONAL, -- The integer h = #E(Fq)/n
}```
6 JavaCard platform

JavaCard is a smartcard development platform where the operating system of the smart card executes a highly modified Java Virtual Machine to run the applets. Since the inception, the platform has gained considerable traction due to ease-of-use and development.

However, the JVM on the cards is highly constrained. For example, from the standard library only the Exceptions of following libraries are implemented:

- java.io
- java.lang
- java.rmi

The platform supports some platform-specific libraries:

- javacard.framework
- javacard.framework.service
- javacard.security
These libraries contain the helper functions for interfacing with the card-specific security zones (e.g. PIN storage, key storage, hardware-accelerated cryptographic operations). Even though the libraries are supported by the JavaCard development kit, support for every possible functionality depends on each specific card model. Using the JCAAlgTest applet and corresponding application [14], it is possible to generate a compatibility report. A database of collected results is available from [13], but during writing the report it was still in infant stage and possessed only a handful of the reports.

The standard for the JavaCard supports the following encryption ciphers (enumerated by their constant names in the library):

- DES CBC-mode
- DES ECB-mode
- RSA
- AES CBC-mode (128, 192, 256 bit modes)
- AES ECB-mode (128, 192, 256 bit modes)
- Korea seed CBC-mode
- Korea seed ECB-mode

Also, the following signature schemes are supported:

- DES MAC4-mode
- DES MAC8-mode
- RSA SHA1 (ISO9795, PKCS1 and RFC2409 paddings)
- RSA SHA224512 (PKCS1 padding)
• RSA MD5 (ISO9795, PKCS1 and RFC2409 paddings)
• RSA RIPEMD160 (ISO9795, PKCS1 and RFC2409 paddings)
• ECDSA SHA
• AES MAC-mode

7 Feasibility study

As the elliptic curve cryptography allows for shorter keys and more efficient implementations, they are being more widely used in practice. However, currently only Weierstrass and Koblitz formulas are standardized and being used in practice, even though the arithmetic is difficult to implement. This is also probably the reason, why there exists only few hardware implementations.

Our goal was to study, how feasible would be to implement Curve25519 purely in software on JavaCard. As the environment on JavaCards is very restricted, then the target was to implement a scalar multiplication operation for Edward curves as this is enough to perform a Diffie-Hellmann key agreement protocol.

Due to the fact that the JavaCard exemplar in our possession allowed only 16-bit integer operations, we had to implement large number library. We implemented 256-bit integers, which were based on arrays of sixteen 16-bit integers. For the addition and multiplication of such integers, we used text-book implementations with computational complexities $O(n)$ and $O(n^2)$ respectively. The subtraction operation is based on addition, adding the inverse of minuend.

The implementation of right shift was straightforward. The division is done using the short division [15] algorithm. As the division algorithm also returns the remainder, then the same operation is used for modulo operation. The modulo exponentiation is implemented using square-and-multiply algorithm.

On top of 256-bit integers we implemented the Edwards addition and general scalar multiplication using double-and-multiply algorithm.

7.1 Performance comparison of arithmetic on different curves

The code of the application is available at MATH.UT.EE server from location
/HOME/IVOKUB/REPO/JC.

Due to very naive implementation, the operations on the JavaCard were very slow. Single 256-bit integer multiplication took around 5 seconds and addition took 0.075 seconds. Using these values, we estimated the time for a single scalar multiplication to take nearly 735 hours. Even though we tried to perform this operation, the card threw an exception after around 15 minutes of computation. We guess that because the virtual machine does not perform garbage collection, we ran out of memory as temporary variables may be allocated during inline mathematical operations.
The exact clock speed of the used JavaCard is not published, but in general they are clocked around 5MHz. Thus, even with more capable devices with clock speeds around 10-20MHz, the computation time would decrease only several times.

7.2 Improvements

Even though the initial estimation for a single scalar multiplication was 735 hours, there are several possible optimization methods.

Firstly, finding an inverse is done using modular exponentiation, but it can also be done using extended Euclid’s algorithm. This optimization would decrease the computation hundredfold. Thus, the total running time for single scalar multiplication would be around 7 hours.

Furthermore, as the modulo operation is done very frequently and internally a division operation is done to find the value, we could use known methods for fast modulo operations, which could reduce the running time thousandfold.

Additionally, if we would use Karatsuba multiplication, we could reduce the running time 30 times, reaching the one second threshold.

As the available memory is very sparse and the memory model does not allow freeing already allocated memory, we had to use temporary variables in a memory area, which does not perform at full capability. Optimizing memory usage allows us to use memory in the fastest bus, allowing to reduce the running time by several factors.

7.3 Alternative implementation

Bernstein has made available [1] a highly optimized routines for Curve25519 operations which use the structure of the underlying field of the curve for very fast operations. We believe that this implementation would allow for reasonable computation time even for software implementation. The reference implementation requires however access to 64-bit integers, which must be implemented in software.

8 Conclusion

A standardization process is in way to standardize Curve25519 for use in cryptographic protocols. We implemented a text-book variant of Curve25519 on smart card to test the feasibility of using it until proper hardware implementations arrive. Even though our initial estimations do not suggest using it on current JavaCards, we believe that using optimization on different levels it is possible to achieve acceptable running speeds.

References


