El Gamal over ECC using Lopez-Dahab (LD) projective coordinates

Jared Garey & Cristian Tapiero

Abstract

This paper describes the process we took to prove the Lopez-Dahab (LD) point doubling algorithm works correctly within a full El Gamal cryptosystem. We first fully designed the system in Singular. We implement the designed using Verilog and show simulation results for our design at many levels. And we use Singular again to transform points from LD projective coordinates to affine coordinates to show equivalence.

Introduction

Elliptic Curve Cryptography is a more efficient use of bits when compared to Feisel cypher-based cryptosystems. There are more complex math operations, making it more difficult to design hardware. These operations include addition, multiplication, squaring, and the more costly operation inversion. To avoid any division or inverse operations the math is done in a projective coordinate system instead of the default affine coordinate system. The Lopez-Dahab paper introduces an altered algorithm from the traditional projective coordinate system algorithms we learned in class, and looks at the performance benefits. The only altered algorithm is for the point doubling operation. We will implement the LD point doubling algorithm in a full El Gamal Elliptic Curve Cryptosystem, as well as other portions of the cryptosystem we did not design in class. We will show that the resulting points are indeed the same when using the default LD point doubling algorithm.

Notations / Abbreviations

LD – Lopez-Dahab, who are the authors of the paper covering the new algorithm

ECC – Elliptical Curve Cryptography or Cryptosystem

In the equations, a capital X is equal to a root α in the primitive polynomial. A lowercase x is a variable for the equation.

We often switch the elliptical curve point notation between binary (110, 010), decimal (6, 2), and polynomial (X^2+X, X).

Contributions / Division of labor

We both individually read the paper on LD, and verified our Singular and Verilog code. We worked together on our project proposal and plans for what we wanted to do, as well as debugging several specific bugs in our code.

Christian put together the Singular and Verilog code, pulling from homework 3. He wrote most of the simulations. He also did the key generation (in singular), encryption (in singular and Verilog), and decryption (in singular and Verilog).

Jared wrote the point generation testbench and singular code to translate a point from the projective to the affine coordinate system. He also wrote the final paper, including gathering extra data and cleaning up project files.

Project Description and Results

Singular Coding and Design

We started our cryptosystem design by choosing an elliptic curve E_{CC} and primitive polynomial P(X). We chose to use k=3 for this design. Instead of increasing the bit length, we chose to focus on implementing a full cryptosystem and kept the bit length shorter.

$$E_{CC} = y^2 + xy + x^3 + X^2x^2 + 1$$
$$P(X) = X^3 + X^2 + 1$$

Generating our alpha (root) values from the primitive polynomial, we get the following.

```
printing values of alpha:
X0 is: 1
X1 is: (X)
X2 is: (X2)
X3 is: (X2+1)
X4 is: (X2+X+1)
X5 is: (X+1)
X6 is: (X2+X)
X7 is: 1
```

We will use (X^6, 1) for our generator point in both singular and Verilog code. We also tried and verified the other generator point (X,1). Just as in homework 3, we also generate all the points on the curve by substituting in x=all values of α into E_{CC}, then factoring the resulting polynomial. This will tell us what all the possible points are on the curve, with 2 points for each value of alpha.

The first unique task for this project was to design the LD algorithm in Singular. P_1 is the generator point. P_2 is generated by doing the point doubling algorithm (P+P). This is the operation that is unique for the Lopez-Dahab algorithm. The rest of the points are generated the same, which is point addition (P+Q).

The LD point doubling algorithm and proof are given in the Lopez-Dahab paper. The formula we implemented is given below.

$$\begin{cases} x_2 = x_1^2 + \dfrac{b}{x_1^2} \ , \\ y_2 = \dfrac{b}{x_1^2} + ax_2 + (y_1^2 + b) \cdot (1 + \dfrac{b}{x_1^4}) \ . \end{cases}$$

Below are the generated points, verifying both algorithms give us the same result.

```
SCHROEPPEL
--------------------------
Generating All the points with primitive element P: (X^6,1)
Point 1:
P = ((X2+X), 1)
Point 2:
P = ((X2+X+1), (X))
Point 3:
P = ((X+1), 0)
Point 4:
P = ((X2), (X2+X))
Point 5:
P = ((X2+1), (X2+X))
Point 6:
P = ((X), 1)
Point 7:
P = (0, 1)
Point 8:
P = ((X), (X+1))
Point 9:
P = ((X2+1), (X+1))
Point 10:
P = ((X2), (X))
Point 11:
P = ((X+1), (X+1))
Point 12:
P = ((X2+X+1), (X2+1))
Point 13:
P = ((X2+X), (X2+X+1))
Point 14:
Point at infinity
P = (0, 0)
```

```
LOPEZ DAHAB POINT DOUBLING
--------------------------
Generating All the points with primitive element P: (X^6,1)
Point 1:
P = ((X2+X), 1)
Point 2:
P = ((X2+X+1), (X))
Point 3:
P = ((X+1), 0)
Point 4:
P = ((X2), (X2+X))
Point 5:
P = ((X2+1), (X2+X))
Point 6:
P = ((X), 1)
Point 7:
P = (0, 1)
Point 8:
P = ((X), (X+1))
Point 9:
P = ((X2+1), (X+1))
Point 10:
P = ((X2), (X))
Point 11:
P = ((X+1), (X+1))
Point 12:
P = ((X2+X+1), (X2+1))
Point 13:
P = ((X2+X), (X2+X+1))
Point 14:
Point at infinity
P = (0, 0)
```

To further verify the point doubling algorithm, we did repeated point doubling. These points match what is generated using point addition. This verification was more useful to us because it is far less operations to get to further points, which is what will be done in a full cryptosystem. We experimented several more point doubling operations that what is shown here.

```
printing only doubled points on the field using DL
Point 2= ((X2+X+1), (X))
Point 4= ((X2), (X2+X))
Point 8= ((X), (X+1))
```

We also briefly looked at the repeated point doubling algorithm given in the LD paper, but did not implement it. This would be interesting to look into and compare efficiency for if more work is ever done.

INPUT: $P = (x, y) \in E \quad i \geq 2$.
OUTPUT: $Q = 2^i P$.

Set $V \leftarrow x^2$, $D \leftarrow V$, $W \leftarrow y$, $T \leftarrow b$.
for $k = 1$ **to** $i - 1$ **do**
  Set $V \leftarrow V^2 + T$.
  Set $W \leftarrow D \cdot T + V \cdot (aD + W^2 + T)$.
  if $k \neq i - 1$ **then**
    $V \leftarrow V^2$, $D \leftarrow D^2$, $T \leftarrow bD^2$, $D \leftarrow D \cdot V$.
  fi
od
Set $D \leftarrow D \cdot V$.
Set $M \leftarrow D^{-1} \cdot (V^2 + W)$.
Set $x \leftarrow D^{-1} \cdot V^2$.
Set $x_i \leftarrow M^2 + M + a, \quad y_i \leftarrow x^2 + M \cdot x_i + x_i$.
return $(Q = (x_i, y_i))$.

Hardware Implementation

For the Hardware design we chose to use the same Montgomery multiplier for the Galois Field multiplication that we used for homework 3. The addition in the Galois Field was simply a bitwise XOR. The implementation for point addition P+Q=R is given in the equations below (given in the LD paper).

We did not implement point addition in homework 3, so implementing this was new. It wasn't obvious if these equations given in the LD paper were equivalent to the traditional equations for projective coordinate point addition, but we think that it is equivalent. There is also an improved algorithm for when $Z_1 = 1$, but we did not implement it.

Projective Elliptic Addition

The projective form of the adding formula is

$$(X_0, Y_0, Z_0) + (X_1, Y_1, Z_1) = (X_2, Y_2, Z_2) \ ,$$

where

$$\begin{array}{lll} A_0 = Y_1 \cdot Z_0^2 \ , & D = B_0 + B_1 \ , & H = C \cdot F \ , \\ A_1 = Y_0 \cdot Z_1^2 \ , & E = Z_0 \cdot Z_1 \ , & X_2 = C^2 + H + G \ , \\ B_0 = X_1 \cdot Z_0 \ , & F = D \cdot E \ , & I = D^2 \cdot B_0 \cdot E + X_2 \ , \\ B_1 = X_0 \cdot Z_1 \ , & Z_2 = F^2 \ , & J = D^2 \cdot A_0 + X_2 \ , \\ C = A_0 + A_1 \ , & G = D^2 \cdot (F + aE^2) \ , & Y_2 = H \cdot I + Z_2 \cdot J \ . \end{array}$$

The point doubling P+P=R algorithm has changed to the equations below (given in the LD paper). With both the point addition and doubling algorithms, the number of squaring operations has increased, which provides more benefit to us because that is where the LD algorithm is used.

Projective Elliptic Doubling

$$2(X_1, Y_1, Z_1) = (X_2, Y_2, Z_2) \ ,$$

$$\begin{aligned} Z_2 &= Z_1^2 \cdot X_1^2 \ , \\ X_2 &= X_1^4 + b \cdot Z_1^4 \ , \\ Y_2 &= bZ_1^4 \cdot Z_2 + X_2 \cdot (aZ_2 + Y_1^2 + bZ_1^4) \ . \end{aligned}$$

We simulated both hardware implementations of the algorithms. For the point addition we did 3 test cases keeping the first point the same and changing the second point. The third test case is testing the case where point 2 is all zeros. Here we discovered a bug where we are getting all 0s for the output. We weren't able to fix this bug.

$$Test\ 1\text{: } (X0, Y0, Z0) = (110, 001, 001)\ and\ (X1, Y1, Z1) = (111, 010, 001)$$
$$\rightarrow (X1, Y2, Z2) = (011, 000, 001)$$

$$Test\ 2\text{: } (X0, Y0, Z0) = (110, 001, 001)\ and\ (X1, Y1, Z1) = (111, 101, 001)$$
$$\rightarrow (X1, Y2, Z2) = (110, 111, 001)$$

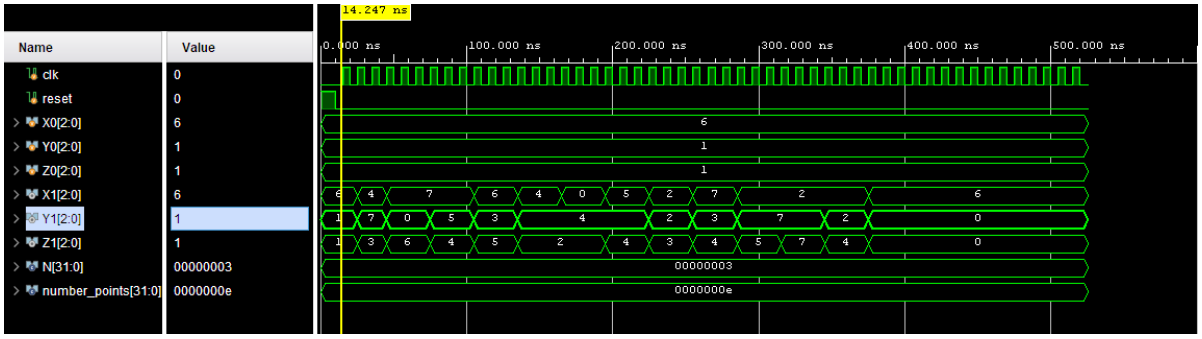
$$Test\ 3\text{: } (X0, Y0, Z0) = (110, 001, 001)\ and\ (X1, Y1, Z1) = (000, 000, 000)$$
$$\rightarrow (X1, Y2, Z2) = (000, 000, 000)$$

For the point doubling testbench, we did 2 test cases.

$$Test\ 1\text{: } (X1, Y1, Z1) = (010, 001, 001) \rightarrow (X2, Y2, Z2) = (110, 001, 100)$$

$$Test\ 2\text{: } (X1, Y1, Z1) = (111, 011, 001) \rightarrow (X2, Y2, Z2) = (101, 111, 010)$$

Below is the point generation simulation of our Verilog code. This testbench file was very similar to a singular script. We implemented a for loop with a point double for the first point, and point addition afterwards. The generator and starting point is P_0(X,Y,Z) = (6,1,1) or (X^2 + X, 1, 1). We generated points to P_{14}, which is a point intersecting the line at infinity. P_{15} should give us the starting point P_1.



This type of operation was useful so we created a statemachine to cycle through the generated points a number of times.

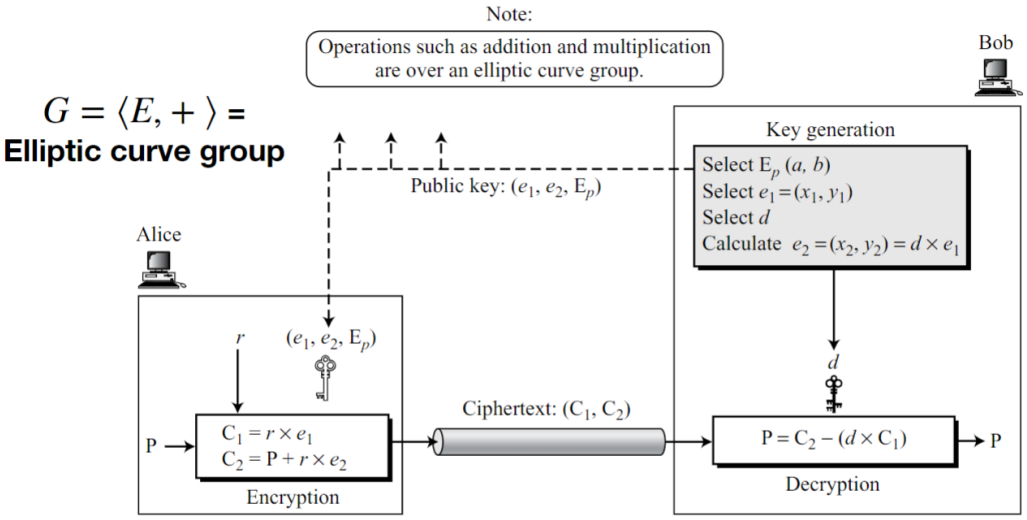
Using Singular, we translated our points from LD projective to the affine coordinate system using the equations below. These points are all the same as the calculated points using singular. One difficulty we had but didn't solve was on P_{15}, the last point. This point should be equal to the first point (the generator point), but instead our code gave us (0,0,0). This is consistent with our testbench simulations, and we still weren't able to figure out how to fix it. We had to take this into consideration later when choosing inputs for encryption and decryption to avoid getting 0s as inputs to our point addition.

$$X_{affine} = \frac{X_{proj}}{Z_{proj}}, \qquad Y_{affine} = \frac{Y_{proj}}{Z_{proj}^2}$$

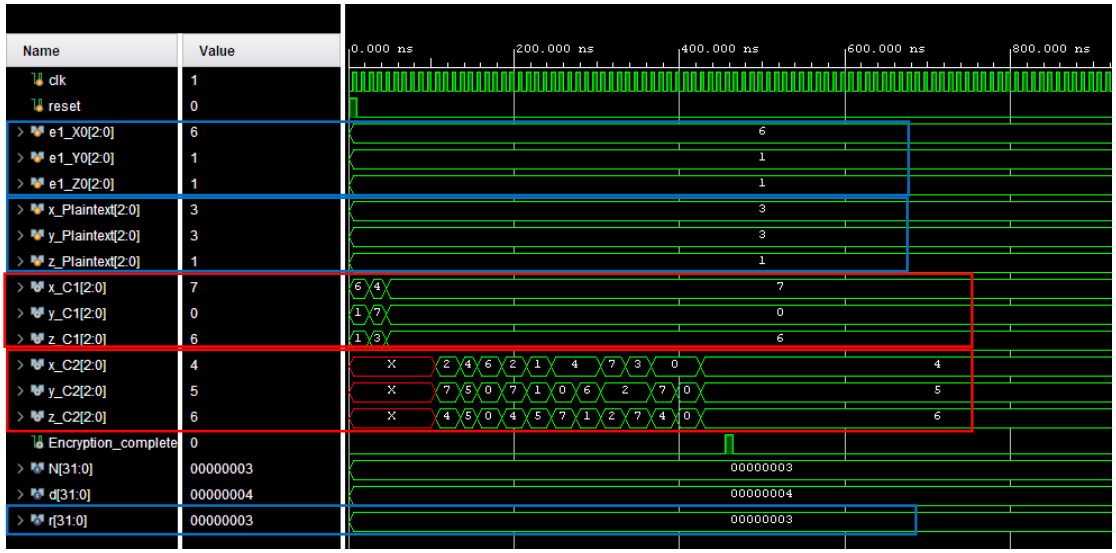
```
HARDWARE VERIFICATION for ALL points (X/Z3  and  Y/Z3^2:)
-----------------------------
proj coord point  ->  affine coord point
P1: ((X2+X),1,1)
 -> ((X2+X),1)
P2: ((X2),(X2+X+1),(X+1))
 -> ((X2+X+1),(X))
P3: ((X2+X+1),0,(X2+X))
 -> ((X+1),0)
P4: ((X2+X+1),(X2+1),(X2))
 -> ((X2),(X2+X))
P5: ((X2+X),(X+1),(X2+1))
 -> ((X2+1),(X2+X))
P6: ((X2),(X2),(X))
 -> ((X),1)
P7: (0,(X2),(X))
 -> (0,1)
P8: ((X2+1),(X2),(X2))
 -> ((X),(X+1))
P9: ((X),(X),(X+1))
 -> ((X2+1),(X+1))
P10: ((X2+X+1),(X+1),(X2))
 -> ((X2),(X))
P11: ((X),(X2+X+1),(X2+1))
 -> ((X+1),(X+1))
P12: ((X),(X2+X+1),(X2+X+1))
 -> ((X2+X+1),(X2+1))
P13: ((X),(X),(X2))
  -> ((X2+X),(X2+X+1))
    (m,0,0) means Horizantle line.
P14: ((X2+X),0,0)
  -> (infinity,0)
    (0,0,0) means point at infinity.
P15: (0,0,0)
  -> (infinity,infinity)
Expecting P14 == P0
```

Now that we have fully proved our algorithms work for all points, we implemented the El Gamal Cryptosystem (block diagram is shown below). The proof of El Gamal and how it works is described in the class lecture slides so we won't go into it here. Because we had an issue with P_{15} not being P_1, we had to avoid certain values for the keys and plaintext.

As shown in the block diagram, the receiver Bob does the key generation. We used singular to try out several values for our key generation. E_p is the elliptic curve which we didn't change, e1 is a generator point, d is like a private constant key, and e2 is derived. We chose e1 = (6,1,1), selected d = 4, and then calculated e2 = (4,6,1) using the point generator circuit and iterating d times. We used the point generation circuit to calculate e2 because it was simple and e1 is always guaranteed to be a generator point.



Below is the encryption simulation of our hardware design. For encryption by Alice, we started with a plaintext P = (3,3,1) and chose a private transmitter key of r = 3. The input to the encryption is boxed in blue, and the output is boxed in red.



Using singular, we verified that the values calculated by hardware were correct. Below is the singular script output. We used e1=(6,1) and e2 = (4,6), which is the same as our hardware because we let Z=1 to make the transformation easy. Our chosen plain text was (3,3). The script shows that C_1 and C_2 are $C_1 = (X+1, 0)$ and $C_2 = (X^2 + 1, X + 1)$. We transformed the points our hardware calculated of $C_1 = (7,0,6)$ and $C_2 = (4,5,6)$ to the affine space and verified there are the same.

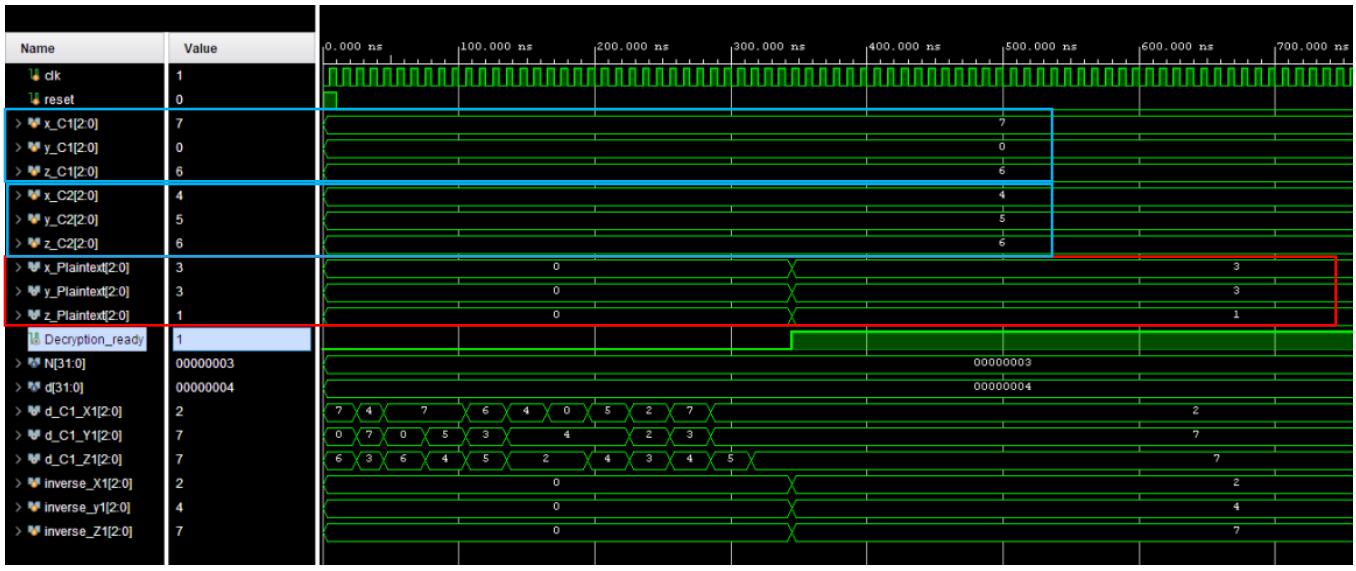
```
d = 4
e1 = ((X2+X), 1)
e2 = ((X2), (X2+X))
Plain text Value = ((X+1), (X+1))
r = 3
) ::Encrypt:26ng j (int j;

C1 and C2:
Encripted Value = ((X+1),0), ((X2+1),(X+1))
```

```
VERIFYING C1 IN HARDWARE in singular is supposed to be (A5,0)
making sure the values are the same dividing by Z3:
(X3,Y3,Z3) = ((X2+X+1),0,(X2+X))
X3/Z3 = (X+1)
Y3/Z3^2 = 0

VERIFYING C2 IN HARDWARE in singular is supposed to be (A3,A5) or (X2 +1 , X + 1)
making sure the values are the same dividing by Z3:
(X3,Y3,Z3) = ((X2),(X2+1),(X2+X))
X3/Z3 = (X2+1)
Y3/Z3^2 = (X+1)
```

For the decryption simulation by Bob, we started with a cypher text C_1=(7,0,6) and C_2=(4,5,6). We chose a private receiver key of d = 4. The input to the encryption is boxed in blue, and the output is boxed in red. At the end of the simulation the plaintext is P=(3,3,1), which is what we started with. We were successful in at least some cases to encrypt and decrypt the point.



We again verified our hardware with Singular (script output below). We performed the math operations individually first, then did the decryption and got the plain text of P = (3,3). We transformed the point our hardware calculated of P = (3,3,1) to the affine space and verified that it is indeed the original plaintext P = (3,3).

```
C1 times d
[1]:
(X2+X+1)
[2]:
(X2+1)
------
C1 inverse
[1]:
(X2+X+1)
[2]:
(X)
---------
Decripted Value = ((X+1), (X+1))
```

```
VERIFYING PLAIN TEXT BACK IN HARDWARE  (X +1 , X + 1)
making sure the values are the same dividing by Z3:
(X3,Y3,Z3) = ((X+1),(X+1),1)
X3/Z3 = (X+1)
Y3/Z3 = (X+1)
```

Concepts learned

Most of what we learned for this project came from studying the LD paper. This algorithm for point doubling is very simple and easy to implement. It focuses on improving just one aspect of projective coordinate system arithmetic: point doubling. It increases the number of field multiplications in order to only slightly reduce the number of field inversions. This was interesting because they didn't start designing an algorithm from the ground up, but instead improved an already existing algorithm.

We learned that when analyzing performance of an algorithm, you have to take into consideration how common your algorithm will be used. The paper brings up in the beginning that if the cost ratio of inversion to multiplication is very small (< 3), then using projective coordinates isn't going to be as efficient. This stuck out because even though it's unlikely one would use an elliptical curve where this is true, it's still possible. And inversion isn't impossible in hardware, just very costly.

We also learned from implementing the El Gamal cryptosystem that it's important to test and treat edge cases. Our design isn't very useful because we aren't doing something correct with the infinity value. To avoid this, we determined we needed to keep our key generation numbers small.

When implementing hardware encryption and decryption you have to be careful with which algorithms you use. We initially implemented a point generation state machine that would take an input point and use it to generate another point N away. This was a mistake because we forgot that the point generation circuit had to use a generator point, and would give incorrect values when given other points.

There are some things we still don't understand. Why does e1 have to be a generator point? Why is e1 a public key if it is a generator point? Does making e1 public make the cryptosystem any less secure? And why is the point addition implementation in hardware different than what is in our lecture slides? We feel we could make guesses on some of these, but are not confident with the limited time we've spent with this material.

Conclusion

The Lopez-Dahab paper states that their new projective coordinates and algorithm are simple to implement. We showed this is indeed true. We enjoyed learning a new algorithm, but even more enjoyed understanding the El Gamal cryptosystem architecture even more. We were not 100% successful because

we had an error in our point addition algorithm in both Singular and Hardware where the system stopped working when an input point had 0 in the X or Y coordinate. But we were able to work around that and prove the encryption and decryption works by carefully testing and choosing many input parameters. We saw lots of opportunity for additional work as we worked. We saw additional steps can be taken for repeated doubling, and comparisons in speed and device utilization can be made for lots of different system parameters, such as the curve, generator point, and generated keys.

References

[1] J. L´opez, R. Dahab, and R. Dahab, "Improved Algorithms for Elliptic Curve Arithmetic in GF(2 n)." Springer-Verlag, 1998, pp. 201–212.

[2] P. Barrett, "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor," in Proceedings of Advances In Cryptology. London, UK, UK: Springer-Verlag, 1987, pp. 311–323. [Online]. Available: http://dl.acm.org/citation.cfm?id=36664.36688

[3] Priyak Kalla slides and book on hardware Cryptography