Illinois Wesleyan University
Digital Commons @ IWU

Honors Projects | Mathematics

2015

A Computational Study of Icart's Function

Thomas Simmons
Illinois Wesleyan University, tsimmons@iwu.edu

Recommended Citation
Simmons, Thomas, "A Computational Study of Icart's Function" (2015). *Honors Projects.* Paper 18.
http://digitalcommons.iwu.edu/math_honproj/18

This Article is brought to you for free and open access by The Ames Library, the Andrew W. Mellon Center for Curricular and Faculty Development, the Office of the Provost and the Office of the President. It has been accepted for inclusion in Digital Commons @ IWU by the faculty at Illinois Wesleyan University. For more information, please contact digitalcommons@iwu.edu.
©Copyright is owned by the author of this document.

A Computational Study of Icart's Function

Thomas Simmons

April 23, 2015

Abstract

A hash function maps some elements of a larger, initial set to elements of a smaller, resultant set. By nature, this leads to collisions and, sometimes, not all elements in the smaller set will be mapped to as a result. The set in consideration here is all points on an elliptic curve. This is a special class of curve with two variables, which takes the form here as $y^2 = x^3 + ax + b$. A hash function is useful in offering a deterministic way to map an input to a pair of x and y values that satisfy such an equation.

This paper experimentally verifies that an asymptotic result on the size of the image for Icart's hash function provided by Fouque and Tibouchi is true for small primes less than 2^{19} and for all curves of conductor less than or equal to 100. Combined with Fouque and Tibouchi's asymptotic result, this proves that the coverage of Icart's hash function is a 5/8 fraction of the points (with some error).

1 Introduction

1.1 Public-key Cryptography

Cryptography is the study of securing data. Cryptography is vital in the Information Age as it provides a way to communicate across the globe in a secure and private manner. A major advance in convenient, secure transmission was the invention of public-key cryptography. In a typical symmetric-key cipher, some of which date back thousands of years, the two parties who want to exchange messages shared a secret key that was used to encipher and decipher the message on each end. Importantly, because both the encryption and decryption process used the same key, each party involved in the secret message passing required the shared key. This poses an interesting question when considered in the modern context of the Internet: How can we share secret keys across an insecure medium? We can assume that anything sent across a network is susceptible to interception by another party. Thus, one cannot simply send a symmetric cipher key to the person with whom he or she wishes to communicate.

The solution is asymmetric (or public-key) cryptography. Typically, in these ciphers, two keys are used. A public key is distributed at large and anybody wishing to send messages to the owner can encrypt with this public key. However, unlike symmetric systems, only the private key can decrypt the ciphertext. These public-key systems rely on conjectured hard problems, such as the discrete logarithm problem and integer factorization. For an example, we can look at Diffie-Hellman-Merkle key exchange, an algorithm that solves an issue related to secret key exhange. The public-key systems that have been invented so far are slower than traditional, symmetric algorithms. It is therefore somewhat impractical to encrypt entire messages with a public key and decrypt with the private key. Rather, a public key is often used to encrypt a symmetric cipher key, where the latter key is then used with a faster, symmetric cipher for the rest of the communication.

Diffie-Hellman-Merkle key exchange addresses this need where the idea is to produce a shared secret between two or more parties, which can then be used to derive a key for symmectric cipher encrpytion/decryption. The traditional algorithm proceeds roughly as follows, where the two actors involved will be called Bob and Alice: Both parties agree on a fixed element g of the large finite field $\mathbb{F}_q$, where q is publicly known. Ideally g is a generator in $\mathbb{F}_q^*$ and need not be kept secret. Now, Alice randomly selects some element $a \in \mathbb{F}_q$, which she keeps secret, and publishes g^a. Likewise, Bob does the same for some b, producing g^b. Both parties then compute g^{ab} using their respective private integers and the other's published number. For some eavesdropper, computing g^{ab} from g^a and g^b has been conjectured to be computationally equivalent to solving the discrete logarithm problem. Thus, Alice and Bob have a shared secret which can be used to secure further communications.

1.2 Elliptic curves

Elliptic curves offer one of the most intriguing areas of research in the cryptographic world today. When used for public-key cryptosystems, especially in the case of digital signatures, elliptic curves promise high security and smaller keys than comparable public-key systems. The points on an elliptic curve with the correct properties can be used to define such cryptosystems. However, establishing points on an acceptable curve is a nontrivial task.

Per Koblitz [5], an elliptic curve is defined over a field, which can be the real numbers, the rational numbers, the complex numbers, or a finite field. In general, the long Weierstrass form for the equation of an elliptic curve in any field is $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$. Here, we let K be a field of characteristic $\neq 2{,}3$ and $x^3 + ax + b$ (where $a, b \in K$) a cubic polynomial with no multiple roots. In this setting, an elliptic curve over K is the set of points (x, y) with $x, y \in K$ that satisfies the short form Weierstrass equation $y^2 = x^3 + ax + b$, along with the identity element denoted O called the "point

at infinity". We perform this transform from the long-form equation with 5 coefficients to the short-form one via the proceedure shown in [6].

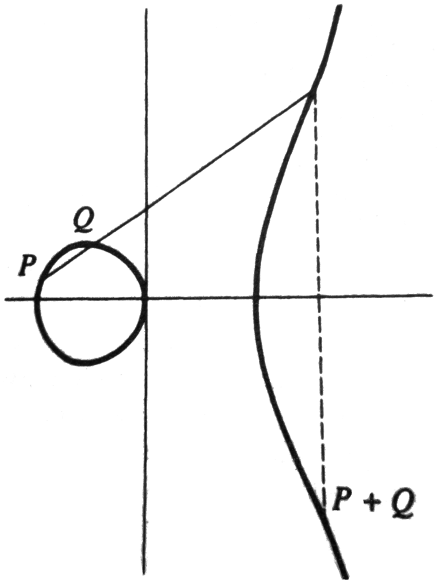

Figure 1: Elliptic curve addition [5]

Importantly, elliptic curves have an algebraic stucture that makes it possible to create analogs to various established cryptosystems. Given the points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, we define $P + Q = (x_3, y_3)$ as:

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2;$$
$$y_3 = -y_1 + \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3).$$

If $P = Q$, the formulae change:

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1;$$
$$y_3 = -y_1 + \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3).$$

Scalar multiplication is performed by repeatedly adding a point to itself a given number of times. There is also a geometric definition for this operation. See [5] §VI.1 for a description. In this geometric setting, we see that $P + -P = O = \infty$ where $-P = (x, -y)$.

We can examine the analog of DHM key exchange (explained above for the integers) to better understand how elliptic curves fit into the public- key

cryptography scene. Elliptic curves also have analogous systems for ElGamal ([5] §VI.2) and DSA [1], to name a few. Once again, this example follows the process of Bob and Alice reaching a shared secret: As in traditional DHM key exchange, Bob and Alice agree upon a field $\mathbb{F}_q$ that is known, but then define an elliptic curve E over said field (which is also known). Then, a base point B is chosen. As in the traditional scheme, B should generate a group of sufficiently large size, though requiring B to be a generator for all points on the curve necessitates a point counting algorithm and non-trivial computation. Now, Alice and Bob pick numbers a and b, respectively. Alice computes aB using point multiplication and publishes it, with Bob doing the same for bB. They can now compute abB, and thus have reached a shared secret. As in the case of traditional DHM key exchange, computing abB from aB and bB is conjectured to be infeasible until the invention of an efficient solution to the discrete logarithm problem.

In all elliptic curve cryptosystems, we want to find valid points on the curve, typically derived from user data. The trivial method for finding a point on an elliptic curve is to simply pick a random element from the field over which the curve is defined to use as an abscissa and then solve the elliptic curve equation to determine if the value has a valid corresponding ordinate (see Koblitz [5] §VI.2). The problem becomes more complicated when we need to connect user data with points on the curve. A randomized algorithm complicates the creation of public-key systems for a couple of reasons. First, using a randomized algorithm does not allow us to bound the run-time of our method. Beyond the obvious detriment of potential performance bottlenecks (which in real-world cases tends not to be a prohibitive issue), this preformance variance can lead to timing attacks on the overall protocol. Second, we want our generated points to have adequate distribution on the curve. The use of a randomized algorithm does not guarantee this property, and as such several points may need to be generated before a satisfactory one is found. Once again, slow performance can result from the repeated generation of potential keys in search of a usable one.

Instead, a deterministic function is needed to make elliptic curve cryptosystems viable. These functions are predictable in their output, as well as in their runtime. With the advent of deterministic point finding, one could efficiently map user-chosen data to and from elliptic curve points in easily repeatable ways. Such algorithms fall into the classification of "hash functions".

1.3 Hash functions

Hash functions, in general, provide a method of mapping the elements of a set of arbitrary size to a set of a fixed size. A more technical definition follows from Menezes, van Oorschot, and Vanstone [7]:

Definition 1.1. A hash function is a computationally efficient function mapping binary strings of arbitrary length to binary strings of some fixed length, called hash-values.

Here, the term "hash function" is somewhat of an abuse of notation because in practice this process would utilize two hash functions. The first would be a cryptographic hash function that takes in the arbitrary user data and produces a fixed length string within $\mathbb{F}_q$. It is this output string that is then mapped onto a curve $E(\mathbb{F}_q)$ by Icart's hash function.

One often encounters hash functions as the basis of the hash table data structure. These hash tables provide expected constant-time lookup speeds for arbitrary data stored in an array, with the worst case being a linear runtime. This is done by applying a hash function to the data (which, given that it is arbitrary, essentially comes from the infinite set) and using the returned value as an index into an array (which is of a limited size, hence the definition above). Such functions are particularly useful in the context of elliptic curves in the mapping of data to points on the curve. As previously noted, the string of fixed length is the x, y values. It is important to reiterate that we do want computational efficiency with these hash functions. As mentioned in Section 1.2, there is a randomized method to find points on a curve but this consumes time and computer resources. Rather, we want a method to efficiently map user data to curve coordinates. One such hash function from integers to elliptic curve points was created by Thomas Icart [4].

Icart's function takes elements of a finite field and maps them to a point on the elliptic curve over that field. We write this function as $I : \mathbb{F}_q \to E(\mathbb{F}_q)$, where $I(u)$ is some point on the curve. Let $I^{-1}(Q)$ denote the preimage of point Q under the function I. Multiple u values can map to the same point so $I^{-1}(Q) \subset \mathbb{F}_q$, and referring to a collision denotes such an instance where $I(u_1) = I(u_2)$.

Icart defines the function over the field $\mathbb{F}_{q^n}$ where $q > 3$ and $q^n = 2 \pmod 3$. On the elliptic curve over this field the coordinates are defined as

$$x = \left(v^2 - b - \frac{u^6}{27}\right)^{1/3} + \frac{u^2}{3}$$
$$y = ux + v$$

where

$$v = \frac{3a - u^4}{6u}.$$

Division in a finite field setting means multiplication by a number's modular multiplicative inverse. For $u = 0$ the output is fixed as the point at infinity. Icart also defines a quartic $P(u)$ in Lemma 3 ([4] §3) as follows:

Definition 1.2. Let $Q = (x, y)$ be a point on the curve E. Define $P_Q(u)$ as

$$u^4 - 6xu^2 + 6yu - 3a = 0.$$

We simply use $P(u)$ when Q is clear from context. The a value in the above equation is the a coefficient of the curve E.

Icart proves that the roots of this polynomial correspond to a particular $I^{-1}(Q)$. Hence, this gives a basis for the collisions we see. A point can be a collision for 0 to 4 u values depending on the factorization of $P(u)$.

An important question about any hash function regards coverage. We often would like to know how many of the potential outputs are actually returned by the hash function being used. Here, the coverage refers to the number of points on the elliptic curve hit by Icart's function for all possible inputs u. By the very nature of hash functions, certain inputs will map to the same output, which we call a collision. In the elliptic curve context, it is useful to know which inputs will collide on the same curve point, as well as what proportion of the total points will be mapped to by the function. Fouque and Tibouchi studied the size of the image for Icart's hash function as well as another (the Shallue-Woestijne-Ulas algorithm) [3]. For the Icart hash function they proposed the inequality $\left|N - \frac{5}{8}q\right| \leq 55q^{1/2}$, where N is the size of the image for a curve over the prime field defined by q. They proved this bound holds for all $q \geq 2^{19}$ and all curves. The purpose of this research is to extend the bound to smaller prime orders.

1.4 Organization of this paper

In this paper, we show some general, predictable behavior observed in Icart's function. These primarily concern collisions resulting from the hashing of certain values. It is also shown how to explicitly construct a curve that will produce collisions for certain inputs. After examining the patterns of collisions for Icart's function, attention is turned to the results presented by Fouque and Tibouchi. Using theoretical and computation tools, their coverage estimate is extended to include fields over smaller primes than they showed.

2 Patterns in Icart function output

2.1 Initial proposition

The original goal of this work was to increase the coverage of Icart's hash function. An ideal hash function would map every value of a finite field to a distinct point on a curve over said field. We know by Hasse's Theorem on elliptic curves that the number of points is bounded as $|N - (q + 1)| \leq 2\sqrt{q}$ [5], where N is the number of points and q is the number of elements in

the finite field. Thus, we want a bijection between field elements and curve points, but by the Hasse bound we know this is impossible. The research here attempts to increase the size of the coverage of Icart's function. This was to be done by redirecting the inputs that would normally map to the same point to distinct outputs instead. For this to work, one must know both which inputs will produce collisions and which points on the curve will not be covered by the hash function. This research tackles both pieces.

Note that curves of characteristic 2,3 are not covered here because $3 \not\equiv 2 \pmod 3$ while the field of only 2 elements is trivial and uninteresting.

2.2 The general cases

When looking at the output for several generic curves (A,B = -1,0; 1,-1; and 0,5) over small prime fields ($q = 11$, 17, 29), the first pattern that emerges is the symmetry of the output (See Appendix A). This leads to the following proposition.

Proposition *If the point (x, y) is on the curve, then so is $(x, -y)$.*

Proof. A point (x, y) on a curve satisfies the equation $y^2 = x^3 + ax + b$. By the properties of multiplication, we know $y^2 = (-y)^2$. Thus, $-y$ also satisfies the equation $(-y)^2 = x^3 + ax + b$, and so if (x, y) is on the curve so is $(x, -y)$. □

This proposition can be refined to further help determine which inputs will map to which outputs. Specifically, we note that for an input u that maps to the point (x, y), its negative will map to the inverse point.

Theorem 2.1. *If $I(u) = (x, y)$, then $I(-u) = (x, -y)$.*

Proof. Icart defines y as $y = ux + v$, where $v = (3a - u^4)/(6u)$. By the properties of multiplication, we see that $\frac{3a-(-u)^4}{6(-u)} = -v$. Given that all exponents for values u, v are even,

$$\left((-v)^2 - b - \frac{(-u)^6}{27}\right)^{1/3} + \frac{(-u)^2}{3} = \left(v^2 - b - \frac{u^6}{27}\right)^{1/3} + \frac{u^2}{3} = x$$

Thus, $y = ux + v$ and $-y = (-u)x - v$. □

Theorem 2.1 allows us to save some computational effort during tabulation. We know that for negative values of u, the Icart function will return the inverse of the point (x, y) that is mapped to by positive u. Therefore, if we were trying to produce every output of the hash function for each input from a finite field, we can effectively skip half of the inputs because the outputs are known.

When noting patterns in the output of Icart's function, the most apparent is the symmetry of the point distribution. Theorem 2.1 tells us the expected behavior of mapping $u, -u$; but, another symmetry worth noting is the number of times a point is output by the hash function. In general, when a point Q is output by the function, we can expect that the inverse, $-Q$, will be mapped to an equivalent number of times.

Theorem 2.2. *If for some point Q, $\left|I^{-1}(Q)\right| \geq 1$ then there exists a point $-Q$ where $\left|I^{-1}(-Q)\right| = \left|I^{-1}(Q)\right|$ and $\forall u \in I^{-1}(Q), -u \in I^{-1}(-Q)$. As will be shown, it can be the case that $Q = -Q$ when $y = 0$ and x satisfies certain properties.*

Proof. We know by Icart's work that $P(u)$ can have 1 to 4 roots for a valid point $Q_1 = (x, y)$, and that those roots make up the set $I^{-1}(Q_1)$. Per Theorem 2.1, we know that for each u, $-u$ will map to the point $Q_2 = (x, -y)$. Thus, for every u in $I^{-1}(Q_1)$ there exists $-u$ in $I^{-1}(Q_2)$ and $\left|I^{-1}(Q_1)\right| = \left|I^{-1}(Q_2)\right|$. □

Theorem 2.2 is the main basis for the computational leaps taken in the test code. By counting the number of u values that map to a given point, we can avoid computation for an equivalent number of u values throughout the tabulation process. Specifically, for any given prime q we need only count the u values up to $q/2$.

These theorems help to save time on the computation of all outputs for a given curve, but there are edge cases for which one must account. When performing tabulation, in most cases, we can assume $u \mapsto (x, y)$ and $-u \mapsto (x, -y)$. When $(x, y) \neq (x, -y)$, this allows us to avoid performing the calculation for $-u$ altogether. However, when this is not the case, special care must be taken to not miscount the point total.

2.3 Special cases

Special consideration must be given to the instances where $(x, y) = (x, -y)$ (specifically where $y = 0$), which means that we cannot count u and $-u$ as mapping to separate points. Note that an elliptic curve will have at most 3 points with $y = 0$, corresponding to at most 3 roots of $x^3 + ax + b$.

Theorem 2.3. *If I(c) = I(-c), c satisfies the equations $6y = 0$ and $c^4 - 6xc^2 - 3a = 0$.*

Proof. The numbers c, -c are u values that map to the same point $Q = (x, y)$ if and only if $u^2 - c^2$ divides the polynomial seen in Definition 1.2 $P(u) = u^4 - 6xu^2 + 6yu - 3a$ evenly. After performing the division, we see that $u^4 - 6xu^2 + 6yu - 3a$ can be written $(u^2 - c^2)(u^2 + c^2 - 6x) + 6yu - 3a + c^4 - 6xc^2$. Thus, c and -c map to the same point Q when $6y = 0$ and $c^4 - 6xc^2 - 3a = 0$. □

Corollary 2.4. *If* $6y \neq 0$ *or* $c^4 - 6xc^2 - 3a \neq 0$, *I(c)* $\neq$ *I(-c). Thus we note that I(c), I(-c) will tend to be different points. This is not always the case, however, as the following examples demonstrate.*

We can consider a few special classes of curves when the values $u, -u$ will collide at the same point. Per Theorem 2.3, $y = 0$ in each instance noted below and the equations were found by rearranging the general Weierstrass equation, $y^2 = x^3 + ax + b$.

First, we assume $b = 0$, which allows us to set $x(x^2 + a) = 0$. If x is assumed to be 0, the equation $c^4 - 6xc^2 - 3a = 0$ loses the x term and can be rewritten as $a = \frac{c^4}{3}$, meaning u and $-u$ will collide at the point (0,0) with the given a, b values.

e.g. Working in $\mathbb{F}_{17}$ on the curve $y^2 = x^3 + 7x$, $\{6, 7, 10, 11\} \mapsto (0,0)$.

Next we consider the instance where $x^2 + a = 0$, also written $a = -x^2$. Now the equation $c^4 - 6xc^2 - 3a = 0$ can be rewritten $c^4 - 6xc^2 + 3x^2 = 0$. Factoring this reveals $c^2 = x(3 \pm \sqrt{6})$. After rearranging to find x and considering this in terms of a, we see that curves with $a = -\left(\frac{c^2}{3 \pm \sqrt{6}}\right)^2$ will produce a collision between u and $-u$ at the point $(\sqrt{-a}, 0)$. This only works in fields where $\sqrt{6}$ has an integer solution.

e.g. Defining the curve $y^2 = x^3 + 16x$ over $\mathbb{F}_{29}$, $\{9, 20\} \mapsto (10,0)$.

Finally, we consider the case with $b \neq 0$. Here, we rewrite $c^4 - 6xc^2 - 3a = 0$ as $a = \frac{(c^4 - 6xc^2)}{3}$. Inserting this back into the Weierstrass equation and expanding gives the cubic $x^3 - 2c^2x^2 + \frac{c^4}{3}x + b = 0$. If c and b values are chosen such that the cubic can be factored, the roots can be used to solve for a. These a and b values can then be used to construct a curve where the known c value will collide with $-c$.

e.g. Over the field $\mathbb{F}_{17}$, we let $c = 9$. This results in the cubic $x^3 + 8x^2 + 11x + b = 0$. Let $b = 2$. which gives the equation a root of 4. Using 4 in the equation to determine a, the final curve is set to $y^2 = x^3 + 9x + 2$. After applying Icart's function, we note that indeed $\{9, 8\} \mapsto (4,0)$.

3 Computational verification

3.1 Structure of code

The code used to verify the Fouque-Tibouchi bound comes in two main parts. First, the Icart hash function was implemented in C++ using Victor Shoup's NTL library [9]. This program takes a prime and two curve coefficients as input and returns the number of distinct points hit by the hash function. Lists of prime numbers up to 2^{19} congruent to 2 modulo 3 (as required by Icart's function) were used to define the finite fields. El-

liptic curve coefficients were provided courtesy of John Cremona [2], who organizes curves by conductor. A full definition is beyond the scope of this paper, but the size of the conductor gives a rough sense of the size of a curve. This was a convenient way to order the test curves, where a list of all conductors less than or equal to 100 provided the rest of the input data. For a full definition of conductors see [8]. Python was used to create glue code that pulled numbers from each of these lists, passes them to the Icart program, and checks the image size against the proposed inequality.

3.2 Pseudocode

The code is embarrassingly parallel, where the prime numbers and curve data are stored in files that can be split up in any convenient manner. The conductor data is provided in long Weierstrass form (i.e. with $a_1, \ldots, a_4, a_6$) and must be processed into our two desired coefficients (see [6] for a detailing of these equations). Note again that fields of characteristic $= 2, 3$ are ignored, which allows us to perform this transform to the short form $y^2 = x^3 + ax + b$.

Thus, the code loops over a range of primes and for each prime it counts the image for a range of curves (as defined for each conductor). These ranges are adjusted as computational capacity dictates. Per curve and for a given prime, we tabulate the size of the image of Icart's function. This counting was done with a hash set data structure to ensure only distinct points were counted. The point itself was used as the key so that the data structure itself would be ensuring uniqueness. As noted throughout Section 2, tabulation was only performed for inputs 1 up to $q/2$ (inclusive, where q is our given prime).

Special care must be taken, however, when calculating the final point count. The basic idea is to simply double the size of the hash set that holds Icart output, but the code accounts for two edge cases. The first is (fittingly) noted in Section 2.3. In cases where u and $-u$ map to the same point, we only want to count it once whereas outright doubling the size of the hash set assumes $u, -u$ map to distinct points. These special cases are counted in a separate data structure (also a hash set). The second case is instances where both Q and $-Q$ are mapped to by u values less than $q/2$ (*e.g.* see Appendix A.2, points $(1, 1)$ and $(1, 16)$). To account for this, we use only x values for keys in the hash set from each output point. This then reflects the symmetry that if Q is on the curve $-Q$ will be as well (Theorems 2.1 and 2.2).

The main idea is to use point data as our key so that the hash sets only account for distinct entries. As noted earlier, the problem parallelizes easily. This also means that the two outer loops could be reversed to achieve the same effect.

Algorithm 1 Tabulate the size of the image

```
for all q in prime range do
  for all curves in conductor range do
    find short form coefficients
    for all u from 1..q/2 do
      point = Icart(u)
      if point.y is 0 and u^4 − 6(point.x)c^2 − 3a is 0 then
        specialCases.insert(point,1)
      else
        set.insert(point.x,1)
      end if
    end for
    count = set.size × 2 + specialCases.size
    if |count − (5/8)q| ≤ 55q^{1/2} then
      print The inequality holds
    end if
  end for
end for
```

3.3 Theoretical analysis

In this algorithm analysis, we treat the cost of the Icart function as a constant time $O(1)$. The hash function consists of a number of multiplications and exponentiations, but the numbers we work with ($< 2^{19}$) all fit within a single word and thus their runtime is not the significant factor in the algorithm's overall complexity.

It is mentioned above that the outer two loops could be reversed without a change in results. For this analysis, we consider the Icart function performed for each curve and then for all primes less than some bound. This is taken as the trivial bound for the count of primes less than a given integer. Icart's function restricts these to primes congruent to 2 modulo 3.

The count of curves per conductor N is denoted $c(N)$. Finally, we can consider the algorithm as a loop over each of these pieces. Thus, we have

$$\sum_{\text{curves for } N} \sum_{\text{primes} \leq x} \sum_{u \in \mathbb{F}_{p/2}} \text{cost of } I(u).$$

This gives rise to the complexity bound

$$O\left(c(N) \cdot \frac{x^2}{\log x}\right).$$

3.4 Data

For all conductors through 100 for all primes less than 2^{19}, the inequality proposed by Fouque and Tibouchi has been shown to hold. Below is an

example of the data found by the code. This is for the conductor 44 with curve coefficients 0, 1, 0, 3, −1. The primes are all congruent to 2 modulo 3 and range from 5 to 4013.

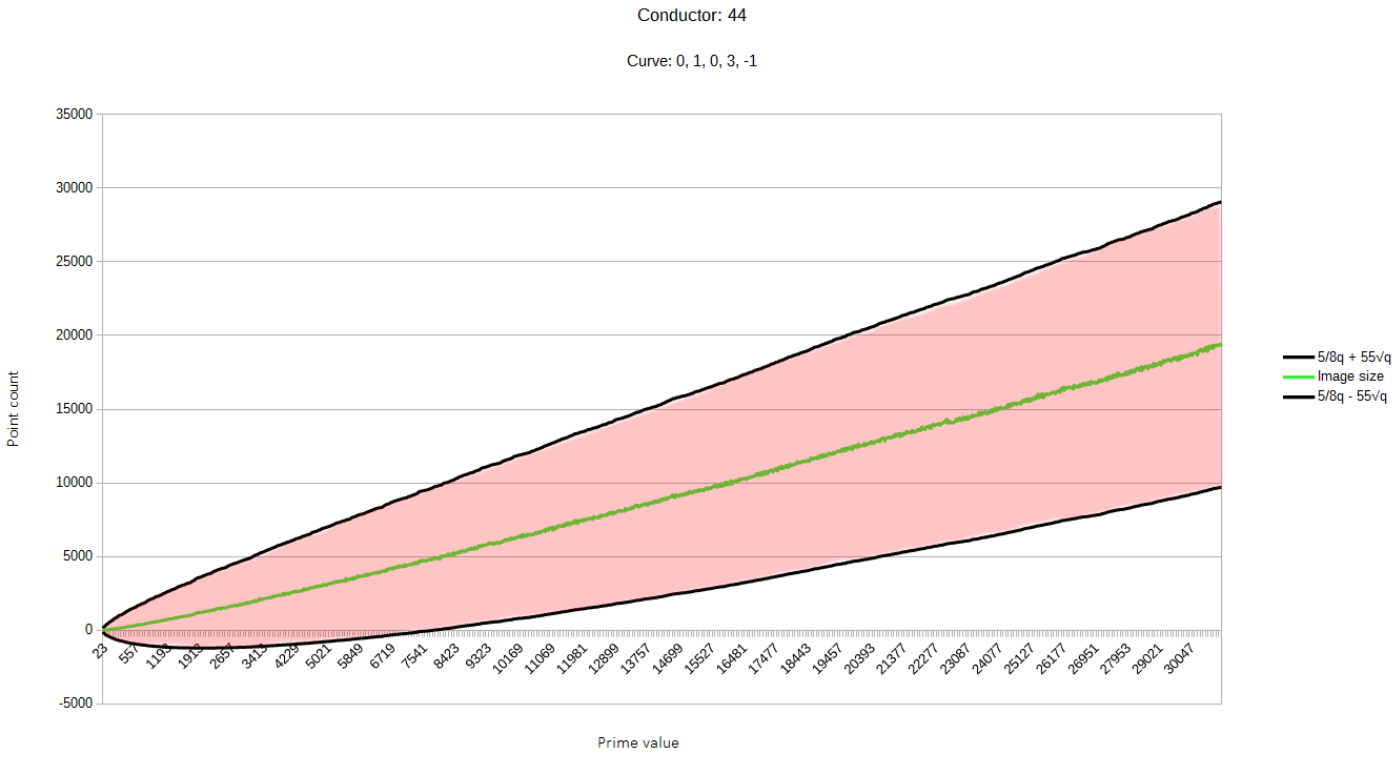

Figure 2: Image size falls within the bounds

The inequality $\left|N - \frac{5}{8}q\right| \leq 55q^{1/2}$ was tested and shown to hold for all curves given by conductor less than or equal to 100 defined over the primes less than 2^{19}.

3.5 Result Analysis

With these results, we have experimentally proven the following theorem:

Theorem 3.1. *For all curves of conductor less than or equal to 100 for all prime order finite fields $\mathbb{F}_q$ where $q < 2^{19}$, the size of the image of Icart's hash function N satisfies the inequality $\left|N - \frac{5}{8}q\right| \leq 55q^{1/2}$.*

4 Conclusion

A perfect hash function onto an elliptic curve would cover ever single point available. In practice, however, this has yet to be attained. It is useful to

know the size of the image when deciding which hash function to use for a particular use case. To that end, this research builds upon the previous work towards estimating the size of the output for Icart's function.

Further research could include verifying the bound for a greater number of conductors and for prime powers. It is also possible that the margin of error, $55q^{1/2}$, present in the Fouque-Tibouchi bound could be tightened somewhat in order to give an even better estimate of the size of the Icart function image. The data collected here shows that the error constant could be lowered as far as 3 in select cases (*i.e.* $N = \frac{5}{8}q + O(3q^{1/2})$).

Acknowledgements

I would like to thank Dr. Andrew Shallue for opening the door to elliptic curves and helping a computer scientist navigate the complex world of abstract mathematics. In addition, this research would not have been possible without the help of Dr. Mark Liffiton and his constant willingness to make our computers work for us.

References

1. Bassham III, L. E. The Elliptic Curve Digital Signature Algorithm Validation System (ECDSAVS). NIST Information Technology Laboratory, Computer Science Division, 2004.

2. Cremona, J. E. `http://homepages.warwick.ac.uk/~masgaj/ftp/data/`

3. Fouque, P.-A. and Tibouchi, M. Estimating the Size of the Image of Deterministic Hash Functions to Elliptic Curves.

4. Icart, T. How to Hash into Elliptic Curves. *Proceedings of Crypto 2009*, LNCS, vol. 5677. Springer, 2009. p. 303-316.

5. Koblitz, N. *A Course in Number Theory and Cryptography.* Springer, 1998.

6. Lemmermeyer, F. Lecture 7, Wednesday 25.02.04. `http://www.fen.bilkent.edu.tr/~franz/ta/ta07.pdf`

7. Menezes, A. J., van Oorschot, P. C., and Vanstone, S. A. *Handbook of Applied Cryptography.* CRC Press, 1997.

8. Silverman, J. H. *The Arithmetic of Elliptic Curves.* Springer, 1986.

9. Shoup, V. `http://shoup.net/ntl/`

A Example hash function images

In this appendix, sample Icart function output is given for an entire curve. This data was generated by brute force, first finding the x, y values that satisfy the given curve equation and then calculating the points given by the hash function for each u value in the given field. These two sets were then compared to find both which points were missed and how many collisions we witness.

A.1 $q = 11, a = -1, b = 0$

```
Final point count: 12
( 0,0 ) was missed by the hash function.
( 1,0 ) was missed by the hash function.
( 4,4 ) appears 1 time(s).
U val(s): 10

( 4,7 ) appears 1 time(s).
U val(s): 1

( 6,1 ) appears 2 time(s).
U val(s): 5, 7

( 6,10 ) appears 2 time(s).
U val(s): 4, 6

( 8,3 ) appears 1 time(s).
U val(s): 9

( 8,8 ) appears 1 time(s).
U val(s): 2

( 9,4 ) appears 1 time(s).
U val(s): 3

( 9,7 ) appears 1 time(s).
U val(s): 8

( 10,0 ) was missed by the hash function.
```

A.2 $q = 17, a = 1, b = -1$

```
Final point count: 18
( 0,4 ) was missed by the hash function.
```

```
( 0,13 ) was missed by the hash function.
( 1,1 ) appears 2 time(s).
U val(s): 8, 12

( 1,16 ) appears 2 time(s).
U val(s): 5, 9

( 2,3 ) appears 1 time(s).
U val(s): 6

( 2,14 ) appears 1 time(s).
U val(s): 11

( 4,4 ) appears 1 time(s).
U val(s): 10

( 4,13 ) appears 1 time(s).
U val(s): 7

( 6,0 ) appears 2 time(s).
U val(s): 4, 13

( 7,3 ) appears 1 time(s).
U val(s): 2

( 7,14 ) appears 1 time(s).
U val(s): 15

( 8,3 ) appears 2 time(s).
U val(s): 14, 16

( 8,14 ) appears 2 time(s).
U val(s): 1, 3

( 11,7 ) was missed by the hash function.
( 11,10 ) was missed by the hash function.
( 13,4 ) was missed by the hash function.
( 13,13 ) was missed by the hash function.
```

B Code

This appendix includes the code used to computationally verify the Fouque-Tibouchi bound. The code implementing Icart's function was written in C++ using the Shoup NTL package. This includes an elliptic curve point class to encapsulate both the x, y data and the functions for adding points together. The Icart class implements the hash function itself. The main file implements the algorithm found in Section 3.2.

The data was stored in two sets of files. All primes congruent to 2 (mod 3) less than 2^{19} were split across six files such that each file contained the primes within a range of 100,000 (*e.g.* `IcartPrimes3.txt` contains the primes p such that $300000 < p < 399999$). The conductors were split up in a similar manner as organized in Cremona's tables.

For the full code with accompanying documentation, in addition to a similar Java implementation, see the following:
`https://github.com/tomsimmons/ecurves`

B.1 main.cpp

The main file for the Icart hash function image counting.

```
1  // A verification of the size of Icart's image
2  //  Coded by Thomas Simmons
3
4  #include "epoint.h"
5  #include "icart.h"
6  #include <stdio.h>
7  #include <unordered_map>
8
9  using namespace std;
10
11 int main(int argc, char* args[])
12 {
13     unordered_map<uint64_t, int> points;
14     ZZ prime;
15     ZZ_p ayy, bee;
16     unordered_map<uint64_t, int> specialCases;
17
18     if (argc != 4)
19     {
20         cout << "This program takes three arguments ↩
               exactly" << endl;
21         cout << "A prime, an A coefficient, and a B ↩
               coefficient" << endl;
22         return 1;
23     }
24     else
```

```
25      {
26          prime = conv<ZZ>(args[1]);
27          ZZ_p::init(prime);
28          ayy = conv<ZZ_p>(conv<ZZ>(args[2]));
29          bee = conv<ZZ_p>(conv<ZZ>(args[3]));
30      }
31
32      // Create new Icart hash object with appropriate ↩
            parameters
33      Icart ic = Icart(&prime,&ayy,&bee);
34      // Bound the loop by p/2 (convert from ZZ to long)
35      unsigned long bound = conv<unsigned long>(conv<ZZ>(↩
            args[1]));
36      bound = (bound / 2) + 1;
37      for (uint64_t u=1; u<bound; u++)
38      {
39          //cout << "u: " << u << endl;
40          // Call hash function
41          EPoint R = ic.hash(ZZ_p(u));
42          //R.print(); cout << endl;
43
44      // Acount for special cases
45      if (IsZero(R.y) &&
46          IsZero( conv<ZZ_p>(power(conv<ZZ>(u),4)) - (conv<↩
                ZZ_p>(6)
47          * R.x * sqr(conv<ZZ_p>(u))) - (conv<ZZ_p>(3) * ayy↩
                ) ) )
48      {
49              specialCases.emplace(R.toLongLong(), 1);
50      }
51      else
52      {
53          // Blank the y coordinate because we want to count↩
                only one inverse per half-field
54          // See README to know what this even means
55          uint64_t halfPoint = R.toLongLong() & 0↩
                xFFFFFFFF00000000;
56          // Add point to hash map
57              points.emplace(halfPoint, 1);
58      }
59      }
60      //cout << "Special Cases: " << specialCases.size() << ↩
            endl;
61      cout << "Points hit: " <<
62              ((points.size() * 2) + specialCases.size() ) ↩
                  << endl;
63      return 0;
64  }
```

B.2 epoint.cpp

A basic class to represent points on a given curve where the particular curve coefficients are static variables for the class. A simple Boolean flag indicates whether a given point is the Point at Infinity. The primary functionality for this class is the point addition formulas.

```
1  // Elliptic Curve Point Implementation
2  //  Coded by Thomas Simmons
3
4  #include "epoint.h"
5
6  using namespace NTL;
7
8  ZZ_p EPoint::a; ZZ_p EPoint::b;
9
10 // Point constructor. It is left to the user to handle x,y↩
       vals in the PaI case
11 EPoint::EPoint(ZZ_p iX, ZZ_p iY, bool ptAinf)
12 {
13     x = iX; y = iY; PaI = ptAinf;
14 }
15
16 // Copy constructor
17 EPoint::EPoint(const EPoint &pt)
18 {
19     x = pt.x; y = pt.y; PaI = pt.PaI;
20 }
21
22 // Sets the prime defining the field for the curve
23 void EPoint::setPrime(ZZ p)
24 {
25     ZZ_p::init(p);
26 }
27
28 // Sets the coordinates of the point or change to PaI
29 void EPoint::setCoord(ZZ_p iX, ZZ_p iY, bool ptAinf)
30 {
31     x = iX; y = iY; PaI = ptAinf;
32 }
33
34 // Sets the coefficients of the curve over which these ↩
      points exist
35 //  Returns true on success, false if discriminant is zero
36 bool EPoint::setCoeff(ZZ_p coA, ZZ_p coB)
37 {
38     if (!IsZero( ZZ_p(-16) * ((ZZ_p(4) * power(coA,3)) +
39                 (ZZ_p(27) * sqr(coB))) ))
```

```
40      {
41          a = coA; b = coB;
42          return true;
43      }
44      return false;
45  }
46
47  // Checks if the point is valid over the given curve
48  //  Returns true if y^2 = x^3 + ax + b
49  bool EPoint::isPoint()
50  {
51      ZZ_p ySide = sqr(y);
52      ZZ_p xSide = power(x,3) + (a * x) + b;
53
54      return ySide == xSide;
55  }
56
57  // Adds two points together as P += Q
58  void EPoint::add(EPoint Q)
59  {
60      ZZ_p yNeg;
61      negate(yNeg, Q.y); // We will need this later
62
63      // P + Point at Infinity = P
64      if (PaI || Q.PaI)
65      {
66          // If Q is the PaI, leave P unchanged. If P is PaI↩
                ,
67          //  P now equals Q
68          if (PaI) setCoord(Q.x, Q.y, Q.PaI);
69      }
70      // P + (-P) = Point at Infinity
71      else if (x == Q.x && y == yNeg)
72      {
73         PaI = true;
74      }
75      // P + P
76      else if (x == Q.x && y == Q.y)
77      {
78          ZZ_p xNew = sqr((3*sqr(x) + a) * inv(2 * y)) - (2 ↩
                * x);
79          y = yNeg + (((3*sqr(x) + a) * inv(2 * y)) * (x - ↩
                xNew));
80          x = xNew;
81      }
82      // P + Q
83      else
84      {
85          negate(yNeg, y);
```

```
86          ZZ_p xNew = sqr((Q.y - y) * inv(Q.x - x)) - x - Q.x;
87          y = yNeg + (((Q.y - y) * inv(Q.x - x)) * (x - xNew));
88          x = xNew;
89      }
90  }
91
92  // Adds two points together as R = P + Q
93  //  Returns the value R
94  EPoint EPoint::operator+(EPoint Q)
95  {
96      EPoint R(a,b,PaI); // Create a copy of this point, P
97      R.add(Q);
98
99      return R;
100 }
101
102 // Prints the point to standard output
103 void EPoint::print()
104 {
105     std::cout << "(" << x << "," << y << ")";
106 }
107
108 // Returns a 64 bit int where the higher order 32 bits are x and the others are y
109 uint64_t EPoint::toLongLong()
110 {
111     // Copy in x coord and shift left by 32
112     uint64_t output = 0;
113     output ^= conv<long>(x);
114     output<<=32;
115     // Copy in y coord
116     output ^= conv<long>(y);
117
118     return output;
119 }
```

B.3 icart.cpp

This class wraps up the Icart function to allow for simple usage. A given Icart object will store the constants 27^{-1} and 3^{-1} for a given prime in order to avoid needless recomputation.

```
1  // Icart's function implementation
2  //  Coded by Thomas Simmons
3
4  #include "icart.h"
```

```
5  #include <iostream>
6
7  //using namespace NTL;
8
9
10 Icart::Icart(ZZ* p, ZZ_p* coA, ZZ_p* coB)
11 {
12     setPrime(p);
13     a = *coA; b = *coB;
14 }
15
16 // Sets the prime defining the field for the curve and ↩
       stores certain values
17 void Icart::setPrime(ZZ* p)
18 {
19     //ZZ_p::init(*p);
20     // Icart hash function uses 1/3 root, equivalent to (2↩
           p-1)/3
21     exp = MulMod( SubMod( MulMod(ZZ(2), *p, *p), ZZ(1), *p↩
           ),
22                   InvMod(ZZ(3),*p), *p);
23     // Store inverse values to be used later
24     ts = inv(ZZ_p(27));
25     th = inv(ZZ_p(3));
26 }
27
28 // Icart's hash function
29 EPoint Icart::hash(ZZ_p u)
30 {
31     // 0 maps to the point at infinity
32     if (IsZero(u))
33     {
34         return EPoint(ZZ_p(0), ZZ_p(0), true);
35     }
36
37     // v = (3a - u^4) / 6u
38     ZZ_p v = ((ZZ_p(3) * a) - power(u, 4)) * inv(ZZ_p(6) *↩
           u);
39     // x = (v^2 - b - u^6/27)^(1/3) + u^2/3
40     ZZ_p x = power( sqr(v) - b - (power(u, ZZ(6)) * ts), ↩
           exp) +
41                     (sqr(u) * th);
42     // y = ux + v
43     ZZ_p y = (u * x) + v;
44
45     return EPoint(x, y, false);
46 }
```

B.4 test.py

This glue code performs the file I/O that tests a particular prime with a given number of conductors. The calculations to simplify the long-form Weierstrass coefficients to the familiar a, b values were also performed in the Python file.

```
# Python script to automate Icart function coverage
#  The magic inequality we seek is |N - (5/8)q| <= 55q^(1/2)

from math import sqrt
from subprocess import Popen,PIPE
from sys import argv
import linecache


# Open primes file
primes = open("data/IcartPrimes" + argv[1] + ".txt","r")
# Open Cremona data file
conductors = open("data/conductors/cremona.00000-09999","r")
# Counter for number of times inequality holds
losses = 0

# For all of Icart's primes, test the inequality
for num in primes:
    # Make prime into string and usable number
    primeS = num.rstrip('\r\n')
    try:
        primeN = int(primeS)
    except ValueError:
        print("Incorrect processing for prime: " + primeS)
    continue

    # Pull in repeated conductors
    for c in xrange(0,100):

        line = linecache.getline(conductors.name, c+1)

        # Process Cremona data, which starts in form "n [a1,a2,a3,a4,a6]"
        cond = line.split(" ")
        cond[1] = cond[1].lstrip('[').rstrip(']\n')
        a = cond[1].split(",")
        for i in range(5):
            a[i] = int(a[i])

```

```
39          # Find A,B from long form coefficients
40          b2 = pow(a[0],2,primeN) + 4*a[1] % primeN
41          b4 = a[0]*a[2] + 2*a[3] % primeN
42          b6 = pow(a[2],2,primeN) + 4*a[4] % primeN
43
44          c4 = pow(b2,2,primeN) - 24*b4 % primeN
45          c6 = (-1)*pow(b2,3,primeN) + 36*b2*b4 - 216*b6 % ↩
                primeN
46
47          inv48 = pow(48,primeN-2,primeN)
48          inv864 = pow(864,primeN-2,primeN)
49
50          A = (-1) * c4 * inv48
51          B = (-1) * c6 * inv864
52
53          # Prepare call to external program ['prog name','↩
                prime','coA','coB']
54          cmd = ["./icart",primeS,str(A),str(B)]
55
56          # Store output of program in array 'output'
57          output = []
58
59          #  Call Icart function
60          p = Popen(cmd, stdout=PIPE)
61          #  Grab stdout, decode to string, then slice to ↩
                grab point count
62          out = p.communicate()[0].lstrip('Pointsh: ').↩
                rstrip('\r\n')
63          try:
64              output.append(int(out))
65          except ValueError:
66              print("Invalid Icart output for prime ",prime)
67              continue
68
69          print(output[0],primeS,A,B)
70
71          lhs = abs(output[0] - (5.0/8.0 * float(primeN)))
72          rhs = 55 * sqrt(float(primeN))
73
74          #print("LHS: ",lhs,"RHS: ",rhs)
75
76          if (lhs <= rhs):
77              print("Point count for prime " + primeS + " is↩
                    " + output[0])
78          else:
79              losses += 1
80              print("Inequality does not hold for prime:" + ↩
                    primeS + " and conductor:" + cond[0] + "(↩
                    line " + c + ")")
```

```
81
82 primes.close()
83 conductors.close()
84 print("The inequality did not hold ",losses," times")
```