ABSTRACT

Title of thesis: TIMING ATTACKS
ON CRYPTOSYSTEMS:
18 YEARS LATER

Clarice Glowacki, Master of Arts, 2014

Thesis directed by: Professor Lawrence Washington
Department of Mathematics

This work applies methodology for cryptosystem timing attacks to elliptic curve encryption using parametric coordinates. Additionally, we attempt to replicate the results found by Paul Kocher regarding timing attacks on RSA cryptosystems. Multiple implementations including Sage, MuPAD, Mathematica, and Python are attempted. Viability of timing attacks with modern computing power is assessed.

TIMING ATTACKS ON CRYPTOSYSTEMS: 18 YEARS LATER

by

Clarice Megan Dziak Glowacki

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Arts
2014

Advisory Committee:
Dr. Lawrence Washington, Chair/Advisor
Dr. Jonathan Rosenberg
Dr. James Schafer

Table of Contents

Chapter 1:

1.1 Background

In 1996 Kocher [1] showed that given certain timing information, it was possible to determine the secret decryption exponent used in RSA encryptions. His method required knowledge of approximately 250 ciphertexts, the amount of CPU time used to decrypt the ciphertexts with the secret exponent, and the public modulus n. When the first k bits of the secret exponent were known, Kocher's data showed that with 85% accuracy, he could guess the next bit in in the secret exponent. Since then, much effort has been made to counteract timing attacks on RSA encryption, often at the expense of computation time for the user. Kocher's computations were all completed on a 120-MHz Pentium computer running MSDOS. CPU speeds have increased in the last 18 years and how a machine processes commands has changed. In addition, elliptic curve encryption has become a preferred cryptosystem. We want to understand how these changes affect the feasibility of a timing attack.

1.1.1 Elliptic Curve Encryption

Elliptic curve encryption is based on the group law for elliptic curve structures. Given two points on a curve, when added together, one gets another point on the curve. However, the addition of points on elliptic curves depends on the points given. Let E be an elliptic curve defined by $y^2 = x^3 + Ax + B$ and $P = (x_1, y_1), Q = (x_2, y_2)$ be points on the curve. Then $P + Q = R = (x_3, y_3)$ for projective coordinates is defined as:

1. If $x_1 \neq x_2$, then

   $x_3 = m^2 - x_1 - x_2$, $y_3 = m(x_1 - x_3) - y_1$, where $m = \frac{y_2 - y_1}{x_2 - x_1}$

2. If $x_1 = x_2$, but $y_1 \neq y_2$, then $P + Q = \infty$

3. If $P = Q$ and $y_1 \neq 0$, then

   $x_3 = m^2 - 2x_1$, $y_3 = m(x_1 - x_3) - y_1$, where $m = \frac{3x_1^2 + A}{2y_1}$

4. If $P = Q$ and $y_1 = 0$, then $P + Q = \infty$

Additionally, $P + \infty = P$ for all points P on E.

Elliptic curve encryption involves the use of a single point P on an elliptic curve and a secret "exponent" n. To encrypt, one computes nP which is done most efficiently through the use of a double and add system. Let n with length w be given by the binary representation $b_1 b_2 \dots b_w$. We may assume that $b_1 = 1$ and n is read left to right in the intuitive way. We follow this procedure:

1. Let $k = 1$, and $S_1 = 0$.

2. If $b_k = 1$, let $R_k = S_k + P$. If $b_k = 0$, let $R_k = S_k$.

3. Let $S_{k+1} = R_k + R_k$.

4. If $k = w$, stop, If $k < w$, increment k and return to step 2.

Notice that $S_k + P$ is only computed if $b_k = 1$, but in both cases, $2R_k$ is found.

1.1.2 RSA Encryption

In Kocher's paper, he outlines the use of RSA timing attacks. To decrypt a ciphertext y using RSA, one computes $y^d \mod n$ where d is a secret decryption key and n is the public product of two secret primes denoted p, q. The above steps are modified as follows:

1. Let $k = 1$, and $s_1 = 1$.

2. If $b_k = 1$, let $r_k = s_k y \mod n$. If $b_k = 0$, let $r_k = s_k$.

3. Let $s_{k+1} = r_k^2 \mod n$.

4. If $k = w$, stop, If $k < w$, increment k and return to step 2.

Here, similar to above, $s_k y$ is only computed if $b_k = 1$ and in both cases, r_k^2 is found.

1.1.3 Timing Attacks

For both elliptic curve and RSA encryption, a timing attack is based on the assumption that a bit's value changes the amount of time it takes for a computer to complete steps 2 and 3. The theory, described by Kocher in [1], is also the same

for both encryption methods. From statistics, we know that if two processes are independent, the sum of their variances is the same as the variance of their sum. Thus if total time $t = t' + t''$ then $Var(\{t\}) = Var(\{t'\}) + Var(\{t''\})$.

For exposition purposes, we will write the group law additively (as in the elliptic curve case). Assume that eavesdropper Eve knows the plaintexts Y_i and the total amount of time it takes to compute nY_i. Assume that she also knows bits $b_1 \dots b_{k-1}$ of the coefficient n. Since Eve knows each Y_i, and the hardware being used, she can time how long it takes to compute $R_1, \dots R_{k-1}$ in the above algorithm for each Y_i and then subtract to determine how long it takes to find $R_k, \dots R_w$. Call this time t_i for each Y_i.

Eve wants to know if $b_k = 1$ or $b_k = 0$. As mentioned above, if $b_k = 1$ then $S_k + P$ is computed. If $b_k = 0$ this addition is not performed. Let t_i' be the amount of time it takes to do this addition step and let $t_i'' = t_i - t_i'$. Effectively, t_i'' is the amount of time it takes to do everything after the supposed addition. Note that at this point, Eve does not know whether or not the addition occurs. However, she may assume that it does, time the computation and then find $Var(\{t_i\})$ and $Var(\{t_i''\})$. If the addition does occur, then we may assume that t_i' and t_i'' are independent times and therefore

$$Var(\{t_i\}) \approx Var(\{t_i'\}) + Var(\{t_i''\}) > Var(\{t_i''\}). \tag{1.1}$$

If the addition does not occur, then t_i' is not part of the actual computations done so t_i and t_i' are independent and

$$Var(\{t_i''\}) \approx Var(\{t_i\}) + Var(\{-t_i'\}) > Var(\{t_i\}). \tag{1.2}$$

Thus, Eve need only check which variance is larger, $Var(\{t_i\})$ or $Var(\{t_i''\})$

Chapter 2:

2.1 Attempted Methods

For all of the following implementations, except for Sage, computations were performed on a 2.93 GHz Intel Core 2 Duo CPU running a 64-bit Windows OS. Sage computations were performed on a Linux virtual machine.

2.1.1 Elliptic Curve Attack

2.1.1.1 Sage

We first used Sage because of its existing elliptic curve infrastructure. The code used in Sage can be found in Appendix A. The elliptic curve used is the NIST-521 curve with parameters (in Sage):

- $p = 2^{521} - 1$
- $A = p - 3$
- $B = 1093849038073734274511112390766805569936207598951683748994586394495953116150735016013708737573759623248592132296706313309438452531591012912142327488478985984$

Figure 1 shows a histogram of 500 elliptic curve multiplications. The x-axis gives the total time observed to compute nP and the y-axis shows the frequency of an observed time. The same random 160-bit integer was used for the coefficient and the 500 points were found using Sage's `E.random_point()` function. For timing, the built-in `cputime()` function was called. The data in Figure 1 have a mean of $0.01953702s$ and a standard deviation of $0.0051370s$. Similar to Kocher's observations, the data shows a rough bell curve. While we were unable to exactly determine the resolution of the `cputime()` function, based on the times observed, we assume that it is no better than the nearest microsecond, but possibly worse. This is reflected in the gaps between bars on the histogram as some times were not observed.

To determine the stability of a given computation time, we repeated the multiplication nP with the same 160-bit n 30 times for each P and observed the time. The standard deviation of the 30 measured times was computed. Figure 2 shows a histogram of this data. The x-axis gives the standard deviation and the y-axis shows the number of points for which a standard deviation was observed. Most points had a standard deviation of less than $4ms$, indicating reasonable stability.

Next, we measured the time to compute nP for a single P and 500 choices for n to understand how measurements would vary for different 160-bit integers. Figure 3 shows a histogram of the results. In this graph the x-axis gives the total time to computte nP and the y-axis shows the number of coefficients n for which the total time was observed. We see a rough variation in the data, but also the same problem as in Figure 1 with some intermediate times not observed.

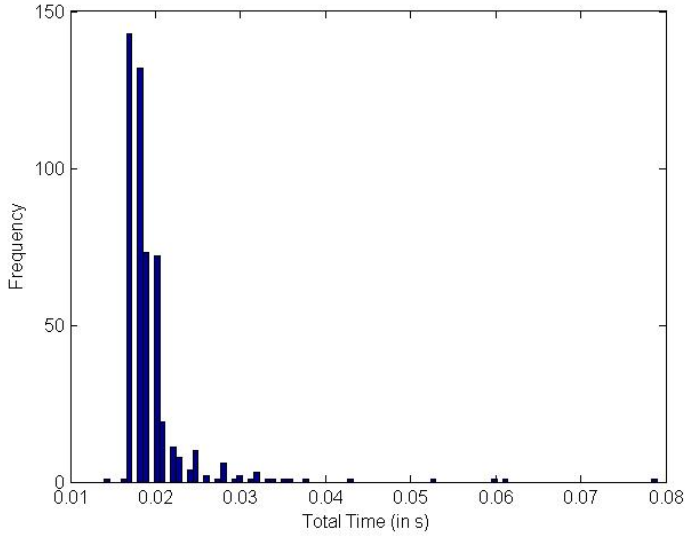

Figure 1: Elliptic Curve Multiplication Times, Sage

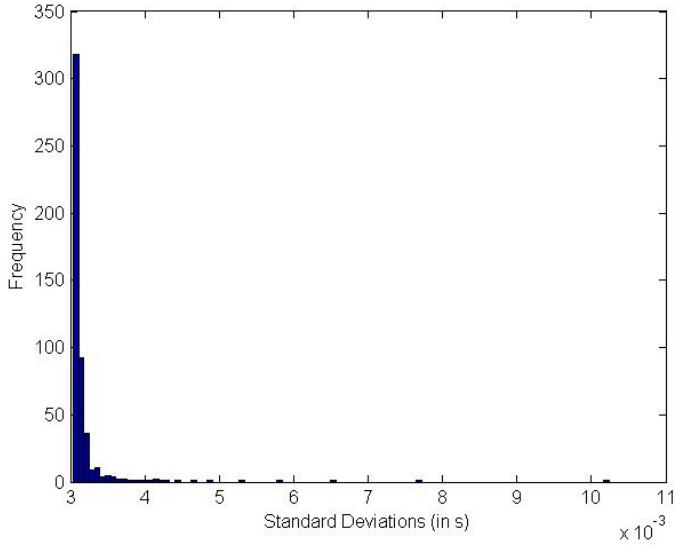

Figure 2: Standard Deviations for Elliptic Curve Multiplication Times, Sage

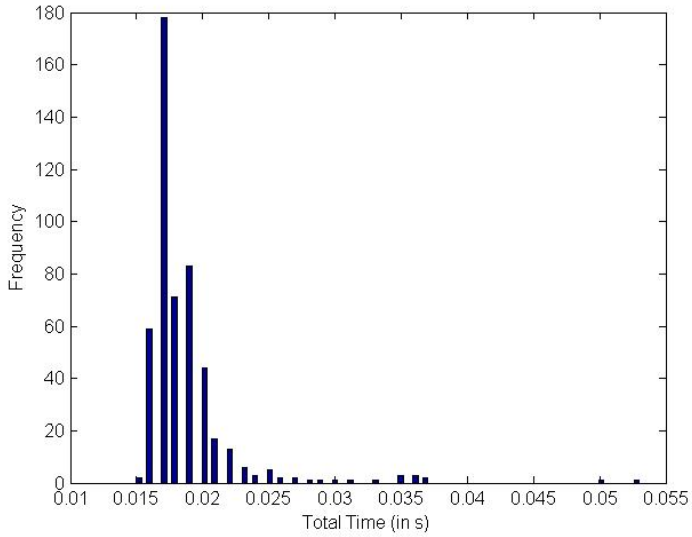

Figure 3: Elliptic Curve Multiplication Times for 500 160-bit Coefficients, Sage

When the timing attack was attempted, it was not successful. First, we used a small coefficient $n = 57 = 111001_2$ thinking that a small coefficient is similar to knowing only a couple of bits of large exponent. We assumed that 5 bits were known and set $n = 11001_2$. In 100 repetitions of the attack guessing only 1 bit, the correct bit was guessed 45 times and incorrect bit 55 times for 45% accuracy. Next, we used a large, 160-bit coefficient with 159 bits assumed known. In 100 repetitions of the attack guessing only 1 bit, the correct bit was guessed 51 times and the incorrect bit 49 times for 51% accuracy. In both cases, time measurements from 500 random elliptic curve points were used to compute the variances. Note that this is double the number of inputs that Kocher used.

The timing attack theory says that when an incorrect bit is guessed, we should see larger variances than prior to the incorrect guess. We tested the attack with 80 of 160 bits known and observed the results and variances. Table 2.1 shows the

results of 5 repetitions of the attack. Column 1 gives the bit at which an incorrect guess was made, column 2 gives the variances observed at the incorrectly guessed bit, and column 3 gives the variances observed at the end of the attack. The two variances which are compared in the theory are listed $Var(t'')$ first and $Var(t)$ underneath. The variances observed at the end of the attack are nearly double the variances observed at the incorrect bit. However, this increase is very gradual, not the expected spike. Variances that are less than or equal to the variances at the incorrect bit were seen at least 10 bits later in each test.

Bit	Variances at bit	Variances at bit 160
81	1.3814080989096613e-05	2.026191450093075e-05
	1.3873913853961377e-05	2.0191891613147683e-05
81	1.8148831897157328e-05	2.524744423186938e-05
	1.728345749229817e-05	2.4314313284479428e-05
82	1.712263188382494e-05	1.9198801907495903e-05
	1.434403782944255e-05	2.1434962315087e-05
82	1.3683165155011767e-05	1.7723214643276426e-05
	1.1890501679049207e-05	1.949544140933549e-05
81	1.4712336205836341e-05	2.373369625073616e-05
	1.4687941022601642e-05	2.24747658376279e-05

Table 2.1: Changes in Variance After an Incorrect Bit is Guessed During an Elliptic Curve Timing Attack

Given the above observations, the exact reason that an elliptic curve attack does not work in Sage is not clear. The data collected are normally distributed and the timing resolution was expected to be refined enough. Even using a larger number of points did not successfully yield reasonable accuracy. It is possible that Sage computes elliptic curve multiplications so quickly and that the code to do so is significantly optimized as to skew observed time data. In addition, the precision and resolution of the timer may not be good enough to accurately capture differences in

bits.

2.1.1.2 MuPAD

We next attempted to implement the elliptic curve timing attack in Matlab because of its perceived slowness. Due to the size of the parameters for the NIST-521 curve, Matlab's symbolic engine MuPAD was used to carry out the elliptic curve multiplication. The code for these operations was modified from code written by Dr. Rosenberg found at [2]. The modified code can be found in Appendix B.

First, 500 points on the NIST-521 curve were multiplied by a 160-bit integer. The computations were timed using MuPAD's `time()` function which measures CPU time in milliseconds. Figure 4 shows a histogram of the times observed. In this graph, the x-axis gives the total time observed to compute nP and the y-axis gives the frequency of a measured time. Interestingly, the fractional milliseconds were always the same for a given whole millisecond. For example, a computation that took between $78ms$ and $79ms$ would always be recorded as $78.0005ms$. This fact can be seen in the large gaps between bars in the histogram. The data shown in the histogram has a mean of $61.995ms$ and a standard deviation of $7.478ms$. Out of the 500 points used, only 4 had multiplication time less than $40ms$. This analysis was completed on two sets of timing data and the means differed by approximately $3ms$ and standard deviations by approximately $2ms$. While the observed times in Figure 4 do appear to have a rough bell shape, such little variation is a concern for timing attacks.

We next wanted to check the stability of timing data with the intent to use an average time for the attack if necessary. To determine how much timing data may change from one iteration to the next, we timed 30 repetitions of the same computation nP for the 500 points and 160-bit integer used earlier. The results can be seen in Figure 5. Here the x-axis represents the standard deviation of the 30 repetitions, the y-axis counts the number of points for which a given standard deviation was observed. The data shown in this graph have a mean of $5.2048ms$ and have a rough bell shape. In all tests, the first time was larger than all subsequent times. It was discovered that Matlab caches computations using a built-in `remember()` function. This function saves the results of large computations and is intended to speed up repeated operations. For that reason, it is not possible to completely eliminate caching making averaging an ineffective method to guarantee timing stability.

We also wanted to ensure that for a given point, there would be enough variation in timing data to distinguish between different 160-bit integers. Figure 6 shows the observed time for multiplication of a single point with 500 different 160-bit integers. The x-axis gives the time to compute nP for a single P and 500 different 160-bit n. The y-axis gives the frequency of an observed time. Once again, the graph shows very little variation. Additionally, all data points counted at one bar were recorded as exactly the same time.

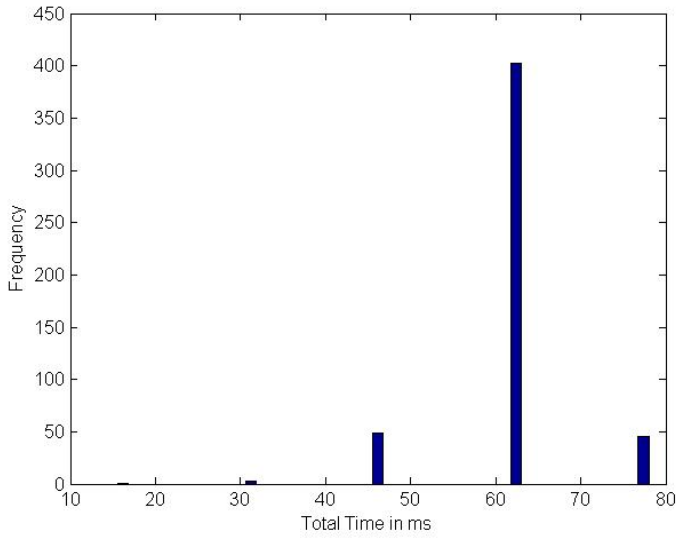

Figure 4: Elliptic Curve Multiplication Times, MuPAD

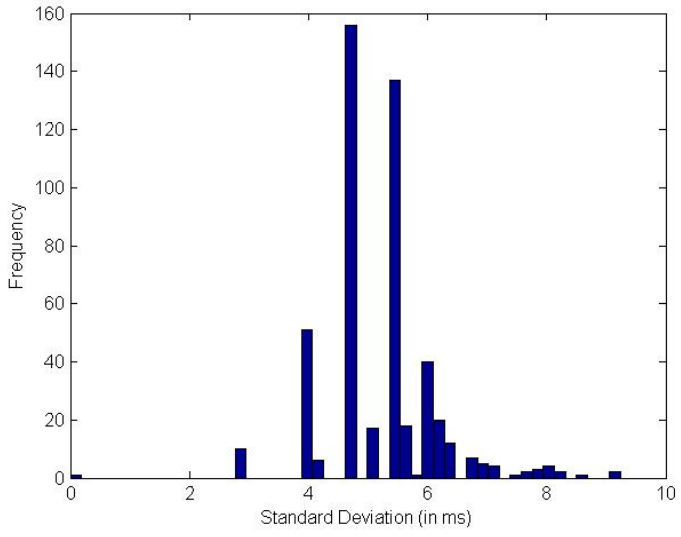

Figure 5: Standard Deviations for Elliptic Curve Multiplication Times, MuPAD

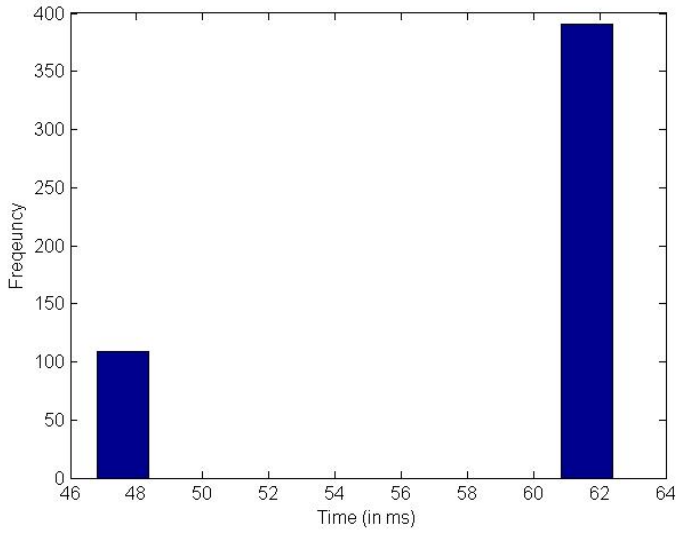

Figure 6: Elliptic Curve Multiplication Times for 500 160-bit Coefficients, MuPAD

When the timing attack was attempted, it did not work. We started with a small coefficient, $n = 57 = 111001_2$ and bits known. In 50 repetitions, the correct bit was guessed 13 times and incorrect bit 37 times for 26% accuracy. Then we attempted a large (160-bit) coefficient with 159 bits known. In 50 repetitions, the correct bit was guessed 23 times and incorrect bit 27 times for 46% accuracy.

We concluded that while MuPAD may be slow enough, there is not enough precision in the time measurement and consequently variation to distinguish between bits.

2.1.1.3 Mathematica

We next chose to try implementing in Mathematica hoping that increased precision could be obtained. Unfortunately, precision for Mathematica is dependent

on the machine used and was only to the nearest millisecond. Using the `timing[]` function we timed 500 elliptic curve multiplications with the same 160-bit integer coefficient.

Figure 7 shows the resulting distribution. In this graph the x-axis represents the time observed to compute nP and the y-axis shows the frequency of the time observed. The data in Figure 7 have a mean of $.0159s$ and standard deviation of $.0027s$. Similar to the data collected from MuPAD, the Mathematica data showed a lack of variation. All multiplications counted in the bar at $.015s$ were recorded as $.0156s$. Figure 8 shows a histogram of standard deviations for 30 repetitions of multiplication with the same 160-bit coefficient for each of the 500 points. Nearly 90 points have a standard deviation of $0.0s$ while most points have a standard deviation between $.002s$ and $.01s$.

Additionally, we timed the multiplication of one point with 500 different 160 bit integers. Figure 9 gives a histogram of the results where the x-axis gives total computation time in seconds and the y-axis shows the number of coefficients n which took a given time. Nearly all of the points had an observed time of $0.0156s$. These results were not better than those for MuPAD so a timing attack was not attempted in Mathematica.

The code that obtained these results can be found in Appendix C.

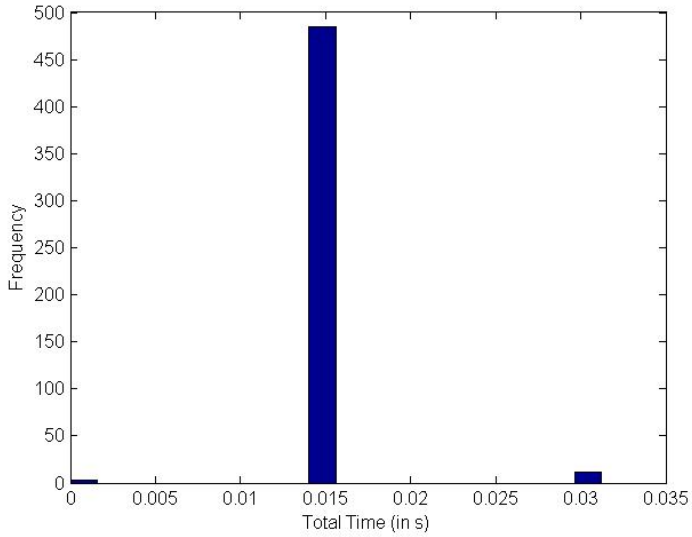

Figure 7: Elliptic Curve Multiplication Times, Mathematica

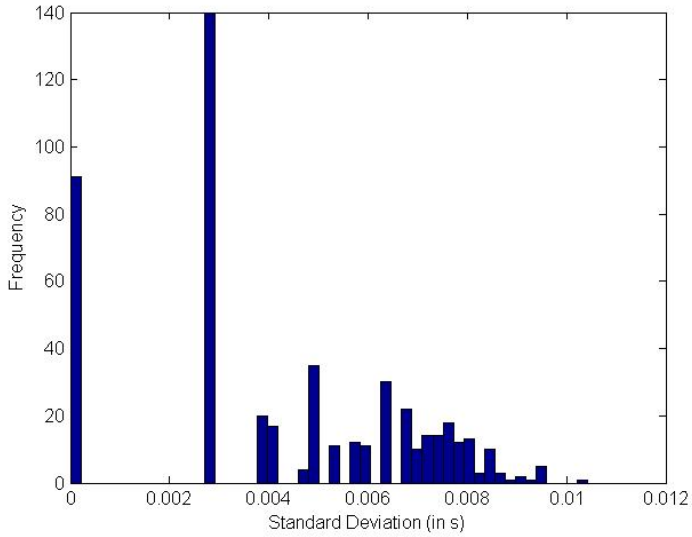

Figure 8: Standard Deviations for Elliptic Curve Multiplication Times, Mathematica

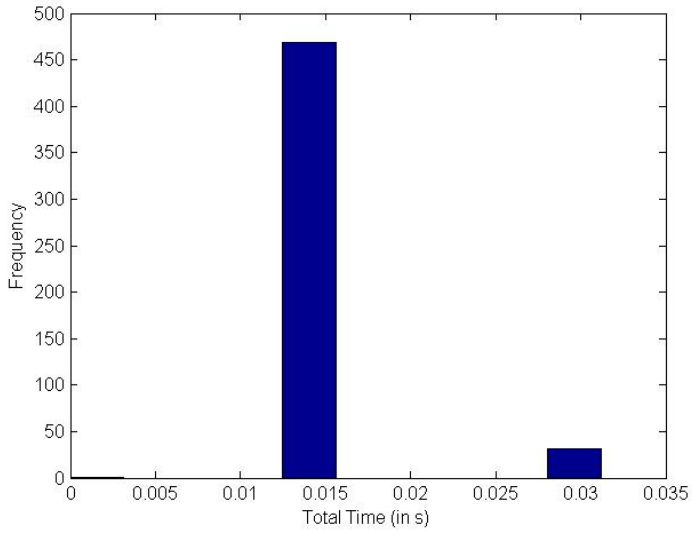

Figure 9: Elliptic Curve Multiplication Times for 500 160-bit Coefficients, Mathematica

2.1.2 RSA Attack

Based on the difficulty in obtaining data which is consistent and varied enough for the elliptic curve timing attack, we returned to Kocher's original attack on RSA. We used RSA-704/212 with parameters

- $p = 909121352959781887844065830260043748589260831032835872042851216896041152864093336782495078836795675680614 1$
- $q = 143859259110045265727809126284429335877899002167627883200914172429324360133004116702003240828777970252499.$

2.1.2.1 Python

We attempted to implement the RSA attack in Python. The `time.perf_counter()` function with $.3\mu s$ resolution was used to measure CPU time spent on computations.

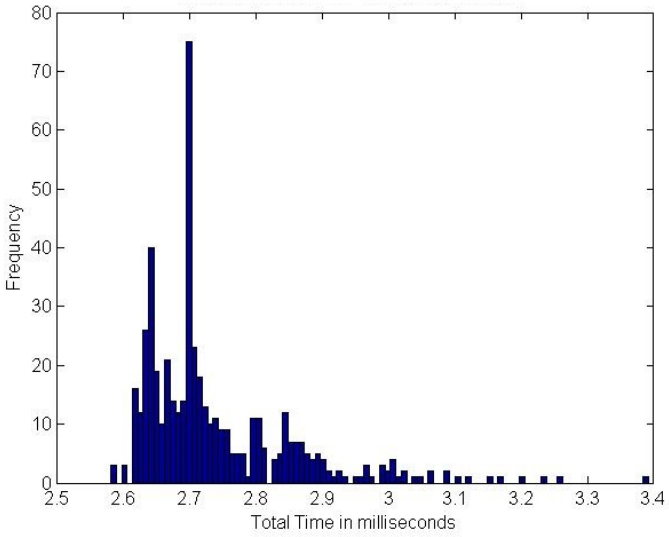

Figure 10: Modular Exponentiation Times, Python

Figure 10 shows the histogram of 500 exponentiations using a 254 bit exponent. The data shown has a mean of $273.1\mu s$ and standard deviation of $160\mu s$. Upon repetition, mean total time differed by less than $5\mu s$. and standard deviation by less than $45\mu s$. Compared to Kocher's original data, the total time was 3 orders of magnitude faster. We next checked the stability of timing data by computing the standard deviation of 30 repetitions of the same computation y^e for the 500 points used above. As can be seen in Figure 11, for most points the standard deviation was less than $100\mu s$.

We next attempted to complete the timing attack in Python. The code for this attack can be found in Appendix D. We began with a small exponent $d = 57 = 111001_2$ and 5 bits known, $d = 11001_2$, under the assumption that finding a small exponent would be equivalent to determining the first few bits of a large exponent. In 50 repetitions of this code, the correct final bit was guessed 25 times with 25

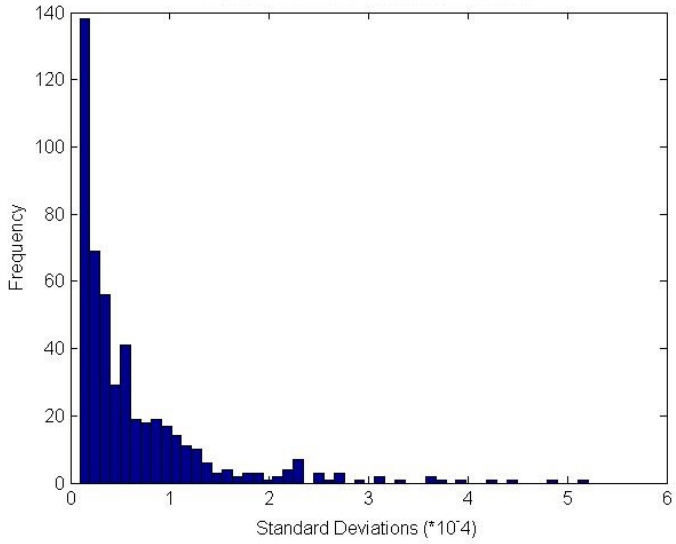

Figure 11: Standard Deviations for Modular Exponentiation Times, Python

incorrect guesses for 50% accuracy. Next, we used a large (254-bit) exponent with 253 bits known. In 50 repetitions of the attack, the correct final bit was guessed 20 times with 30 incorrect guesses for 40% accuracy.

In an attempt to determine the reason for such failure, we compared the exponent size and standard deviation of the total time distribution. We used 52 different exponents of ranging from 20-bit length to 1040-bit length. For each exponent, each dot in Figure 12 represents the standard deviation of the total time distribution for that exponent. The process was repeated 10 times for each exponent. As the exponent gets larger, we see greater spread in the standard deviations measured, indicating that the timing data does not remain consistent and may not be reliable.

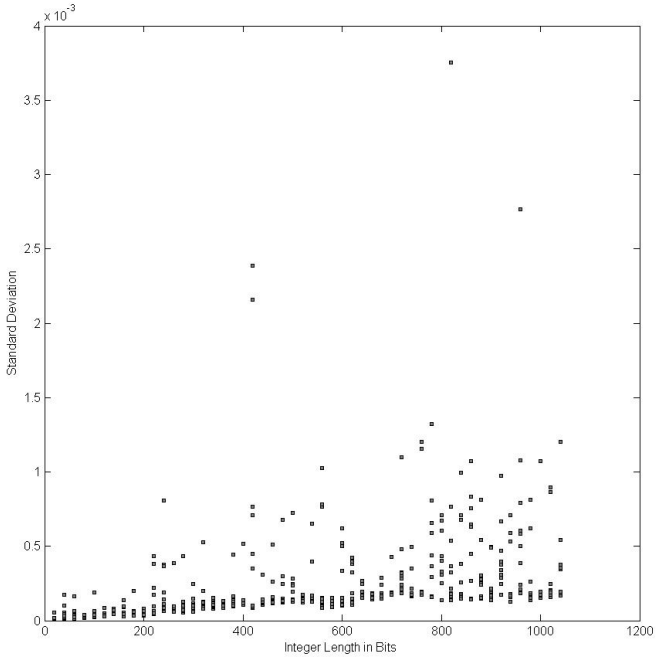

Figure 12: Total Time Standard Deviations for Modular Exponents of Different Lengths

Today's RSA encryptions typically use much larger decryption exponents. Figure 13 shows a histogram of the total time for computing $y^d \mod n$ when d is a 1024-bit integer. The data shown has a mean of $10.57ms$ and a standard deviation of $.3968ms$. These computations take significantly more time than those with the 256-bit exponent, but still 1 order or magnitude less than the times Kocher observed. Figure 14 shows a histogram of standard deviations. For each y-value the standard deviation of 30 timed repetitions of y^d for a 1024-bit d were calculated. The x-axis gives the standard deviation and the y-axis gives the frequency of a given standard deviation. This suggests that at the 1024-bit level, we may be able to use a timing attack to determine the next bit. However, given the earlier data, it is not likely

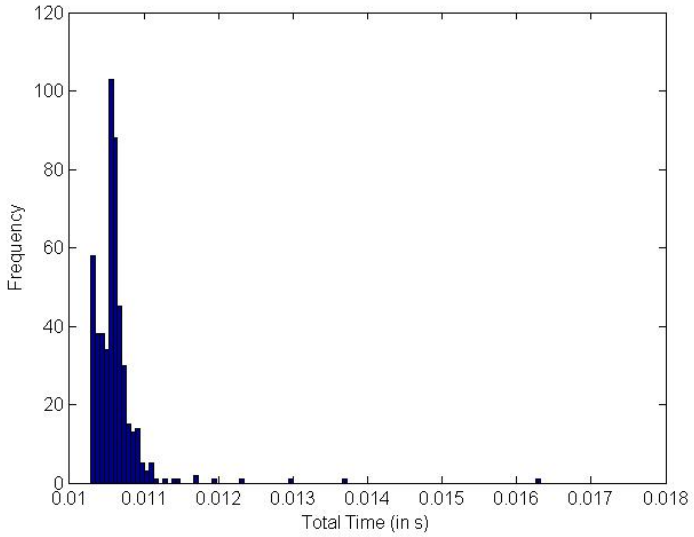

Figure 13: Modular Exponentiation Times for 1024-bit Exponent, Python

we would be able to start from 0 known bits and guess the entire exponent with reasonable success.

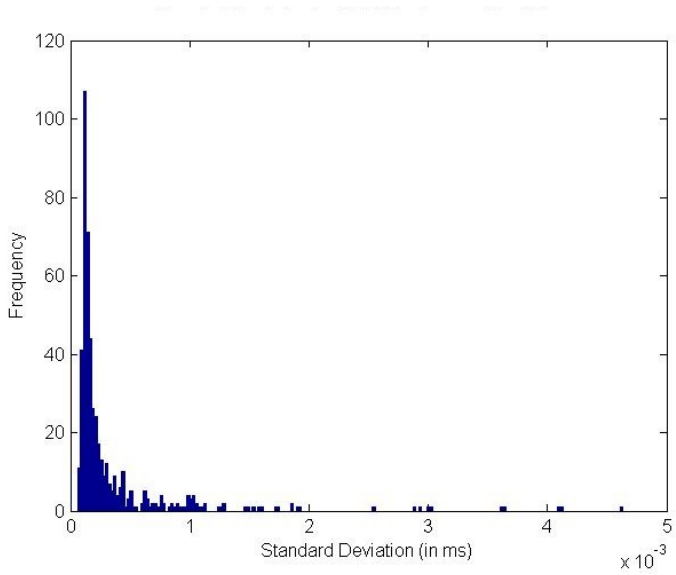

Figure 14: Standard Deviations of Modular Exponentiation Times for 1024-bit Exponent, Python

Chapter 3:

3.1 Computing Power

Nearly 20 years have passed since Kocher's results were published. In that time, the computer industry has seen the introduction of multi-core CPU's and parallel processing. In Kocher's paper, he observes modular multiplication times which take at least $1ms$ and modular exponentiation times which take at least $400ms$. The computer he used had a 120-MHz Pentium processor running MSDOS. The results we obtained were from a computer running a 2.93-GHz Intel Core 2 Duo processor. We saw modular exponentiation times around $2.7ms$. While the times we observed were only 2 orders of magnitude faster than those from 1996, there are potential complications introduced by the dual core processor. In particular, a scenario where the timing function is carried out by one processor and the actual computation by another would be a problematic for obtaining accurate timing.

In the future, it is not unreasonable to expect parallelization of the actual computation [3](especially if a GPU is utilized by the user's encryption software). This results in extremely fast computations compared to those not parallelized. An especially fast computation in itself is not a problem. However, computations so fast that bit differences cannot be distinguished eliminates the possibility of a tim-

ing attack. Thus for a timing attack to be viable in the future, we must have timers with large CPU time precision to counteract the increased speed of the computations. Assuming that a timing function is precise enough to measure a computation with a small exponent, that measurement must be distinguishable from one where the exponent differs by a single bit with variation in timing accounted for. Even with an accurate and precise timing device, lack of consistency will make single bit changes indistinguishable. Any person trying to implement a timing attack must be sure to use the same exact computer specifications and encryption implementation. However, this is not enough if the timing data obtained cannot be expected to be reliable.

Chapter 4:

4.1 Conclusions

Based on the data collected and analysis above, it is suggested that timing attacks may no longer be a viable way to obtain a secret key in both the elliptic curve and RSA methods. An eavesdropper Eve is assumed to have access to the ciphertexts and the total time it took to decrypt each ciphertext with the same secret key. However, the accuracy, precision and consistency of the timing data observed by an eavesdropper are not guaranteed especially if optimization techniques such as parallelization are used for fast computation. Additionally, our observations show it is unlikely that an eavesdropper would be able to accurately determine an entire secret exponent starting with the first bit due to such small, fast computations and timer precision. Given these obstacles, we do not believe that timing attacks will be a good use of resources to determine a secret key.

Chapter A: Sage Files

A.1 SageCode

```
# Randomly generates points and secret coeffcient c
p = 2^521 - 1
F = GF(p)
A = p - 3
B = 109384903807373427451111239076680556993620759895168374899458639449595311615073501601370873757375962324859213229670631330943845253159101291214232748847898598 4

E = EllipticCurve([F(A), F(B)])

from sage.misc.prandom import randrange
#c= getrandbits(160)
import numpy
Points10000=numpy.array([])
for i in range(10000):
    Pi=E.random_point()
    Points10000= numpy.append(Points10000,[Pi])
save(Points10000, 'Points10000')
#save(c, 'C')
#c
```

```
# Muliplies c*Points
import numpy
#import sage.misc.trace
Points = load('Points500')
c = load('C')
#c=57
p = 2^521 - 1
F = GF(p)
A = p - 3
B = 109384903807373427451111239076680556993620759895168374899458639449595311615073501601370873757375962324859213229670631330943845253159101291214232748847898598 4
```

```
E = EllipticCurve([F(A), F(B)])

Time = numpy.array([], dtype=object)
#Make sure that points are of type ECPoint
Pt=[]
for i in range(0, len(Points),3):
    Pi = E(Points[i], Points[i+1], Points[i+2])
    Pt=Pt+[Pi]

for j in range(len(Pt)):
    ti = cputime()
    _=c*Pt[j]
    ci=cputime(ti)
    Time = numpy.append(Time, [ci])
Time




#Elliptic Curve Timing Attack
import numpy

Points = load('Points500')
p = 2^521 - 1
F = GF(p)
A = p - 3
B = 10938490380737342745111123907668055699362075989516837489
        9
        4586394495953116150735016013708737573759623248592132296
        70
        6313309438452531591012912142327488478985984
E = EllipticCurve([F(A), F(B)])


c=load('C')
#c=57
cBin=bin(c)
len(cBin)
# Let w give the number of known bits

w=159   #when c is 160-bit
#w=5    #when c is 6 bits
#w=80   #test length


cBin = cBin[2:len(cBin)]
KnownBitsInit = cBin[1:w]

KnownBits=KnownBitsInit
KnownBitsIntInit=int(str(KnownBits), base=2)
KnownBitsInt=KnownBitsInit
print(cBin)
Pt =[]
for i in range(0, len(Points),3):
    Pi = E(Points[i], Points[i+1], Points[i+2])
    Pt = Pt+[Pi]
```

```
Time=numpy.array([], dtype=object)
for i in range(len(Pt)):
    ti=cputime()
    _=c*Pt[i]
    ci=cputime(ti)
    Time=numpy.append(Time, [ci])

for n in range(100):
    KnownBits=KnownBitsInit
    KnownBitsInt=KnownBitsIntInit
    KnownTime = numpy.array([], dtype=object)
    for k in range(len(Pt)):
        tk = cputime()
        _=KnownBitsInt*Pt[k]
        ck=cputime(tk)
        KnownTime = numpy.append(KnownTime, [ck])
    t=Time-KnownTime
    vart= variance(t)
    for j in range(len(cBin)-w):
        NextBit0 = KnownBits+'0'
        NextBit1 = KnownBits+'1'
        INextBit0 = int(str(NextBit0), base=2)
        INextBit1 = int(str(NextBit1), base=2)
        Time_NextBit1 = numpy.array([], dtype=object)
        Time_NextBit0 = numpy.array([], dtype=object)
        for j in range(len(Pt)):
            tj_NextBit1 = cputime()
            _=INextBit1*Pt[j]
            tj1=cputime(tj_NextBit1)
            Time_NextBit1 = numpy.append(Time_NextBit1, [tj1])
            tj_NextBit0 = cputime()
            _=INextBit0*Pt[j]
            tj0=cputime(tj_NextBit0)
            Time_NextBit0=numpy.append(Time_NextBit0, [tj0])
        tPrime = Time_NextBit1-KnownTime
        TimeDiff1 = t-tPrime
        v1=variance(TimeDiff1)
        #v1
        #vart
        if v1<vart:
            KnownBits = NextBit1
            KnownTime=Time_NextBit1
            t= Time-KnownTime
            vart=variance(t)
            print(INextBit1)
        elif v1>vart:
            KnownBits = NextBit0
            KnownTime=Time_NextBit0
            t= Time-KnownTime
            vart=variance(t)
            print(INextBit0)
        else:
            print('Same variance')
```

```
#Gather Standard Deviation Data
import numpy
Points = load('Points500')
c = load('C')
p = 2^521 - 1
F = GF(p)
A = p - 3
B = 109384903807373427451111239076680556993620759895168374899
        4
        586394495953116150735016013708737573759623248592132296706
        3
        133094384525315910129121423274884789859 84
E = EllipticCurve([F(A), F(B)])

StdDevs = numpy.array([], dtype=object)
Reps=numpy.array([], dtype=object)
Pt=[]
for i in range(0, len(Points),3):
    Pi = E(Points[i], Points[i+1], Points[i+2])
    Pt=Pt+[Pi]
len(Pt)
for j in range(500):
    for k in range(0,30):
            tk = cputime()
            _=c*Pt[j]
            ck=cputime(tk)
            Reps=numpy.append(Reps, ck)
    StdDevs = numpy.append(StdDevs, std(Reps))
StdDevs



#Multiply single point by 500 160-bit Integers

import numpy
Points=load('Points500')
Coeffs=load('Coeffs500')

Pt=[]
for i in range(0, len(Points),3):
    Pi = E(Points[i], Points[i+1], Points[i+2])
    Pt=Pt+[Pi]
len(Pt)

Times=numpy.array([], dtype=object)
for i in range(500):
   ti= cputime()
   _=Coeffs[i]*Pt[143]
   ci=cputime(ti)
   Times= numpy.append(Times, ci)
Times
```

```
std(Times)
numpy.mean(Times)
```

Chapter B: MuPAD Files

B.1 ellcurve.mu

```
plugin:=proc(x, a, b, n, bit)
local y2, y;
begin
  y2:=_mod(x^3+a*x+b, n):
  if numlib::legendre(y2,n)=1
  then  y:=numlib::sqrtmodp(y2, n);
    if bit=1 then
    return(y)
    else
    return(_mod(-y,n))
    end_if;
        else return(0)
        end_if;
  end_proc:

testEC := (x,y,a,b,n) -> if (x=infinity and y=infinity)
or _mod(x^3+a*x+b-y^2, n) = 0
then TRUE else FALSE end_if:

addEC := proc(x1,y1,x2,y2,a,b,n)
local z, m, num, den, x3;
begin
if (x1=infinity and y1=infinity)
   then return([x2,y2]) end_if;
if (x2=infinity and y2=infinity)
   then return([x1,y1]) end_if;
if (x1=x2 and y1=y2 and y1=0)
   then return([infinity,infinity]) end_if;
if (x1=x2 and y1<>y2)
   then return([infinity,infinity]) end_if;
if (x1=x2 and y1=y2) then den:=2*y1 else
   den:=x2-x1 end_if:
z := gcd(den,n):
if (z<>1 and z<>n)
        then print("found factor of n", z);
   return() end_if:
if (x1=x2 and y1=y2) then num:=3*x1^2+a else
```

```
num:=y2-y1 end_if:
m := _mod(1/den ,n)*num:
x3 := _mod(m^2 - x1 -x2, n):
return([x3, _mod(m*(x1-x3)-y1, n)])
end_proc:

multEC := proc(k,mult)
  local z, out, x1, y1;

begin
read("C:\\Users\\Clarice\\SkyDrive\\Documents\\School\\MatlabFiles\\521Points1000"):

  z:= mult:
out:=[infinity,infinity]:
x1:=Xvals[k]:
y1:=Yvals[k]:
n:=p:
A:=a:
B:=b:
while z<>0 do
  while (_mod(z,2))=0 do
   z:=z/2:
   [x1,y1] := addEC(x1,y1,x1,y1,A,B,n):
   forget(addEC);
 end_while:
   z:=z-1:
   out := addEC(x1,y1,out[1],out[2],A,B,n):
   forget(addEC);
end_while;

return(out)
end_proc:
```

B.2 TimeMuPAD.mn

```
P521 Parameters
a:=-3
b:=79691428531592760000:
p:=6864797660130609714981900799081393217269435300143305409394463459185543183397656052122559640661454554977296311391480858037121879997166438125740282911150571 51:

Load necessary files

reset();
delete HISTORY;
HISTORY;
read("C:\\Users\\Clarice\\SkyDrive\\Documents\\School\\MatlabFiles\\ellcurve.mu");
read("C:\\Users\\Clarice\\SkyDrive\\Documents\\School\\MatlabFiles\\521Points1000"):
a:=-3:
b:=109384903807373427451111239076680556993620759895168374899458 6
```

```
        3944959531161507350160137087375737596232485921322967063 1
        3309438452531591012912142327488478985984:
p:=6864797660130609714981900799081393217269435300143305409394463
        4591855431833976560521225596406614545549772963113914808 5
        8037121987999716643812574028291115057151:


Generate 500, 160-bit coefficients

C:=[]:
IntC:=[]:
for i from 1 to 500 do
C:=append(C,stringlib::random(160,["0","1"], Prefix= "1" ));
IntC:=append(IntC, text2int(C[i], 2));
end_for:
write("160Coeff", IntC);
IntC;
Multiply 500 160-bit Coefficients by 1 point
read("160Coeff"):
read("C:\\Users\\Clarice\\SkyDrive\\Documents\\School
        \\MatlabFiles\\ellcurve.mu");
TimeSinglePoint:=[];

for j from 1 to 500 do
ti:=time(multEC(2,IntC[j])):
TimeSinglePoint:=append(TimeSinglePoint, ti):
end_for:
TimeSinglePoint;



Find Standard Deviation for same Point and Coefficient
 Repeated 30 times

IntC:=124163239648439419369474490533141024713125101186 8:
StdDevs:=[]:
for i from 1 to 500 do
Reps:=[]:
for j from 1 to 30 do
ti:=time(multEC(i,IntC)):
Reps:=append(Reps, ti)
end_for:
StdDevs:=append(StdDevs, stats::stdev(Reps)):
end_for:
StdDevs;


The Elliptic Curve timing attack

HISTORY:=0:
reset();
read("C:\\Users\\Clarice\\SkyDrive\\Documents\\School
        \\MatlabFiles\\ellcurve.mu");
TotalTime:=[]:
```

```
a:=-3:
b:=109384903807373427451111239076680556993620759895168374899458639
        449595311615073501601370873757375962324859213229670631330 9
        438452531591012912142327488478985984:
p:=686479766013060971498190079908139321726943530014330540939446345
        918554318339765605212255964066145455497729631139148085803 7
        121987999716643812574028291115057151:
IntC:=12416323964843941936947449053314102471312510118 68;
BinC:=int2text(IntC,2):
KnownBits:=BinC[1..159]:
KnownBitsInt:=text2int(KnownBits,2):


Large coeffcient
TotalTime:=[];
for i from 1 to 500 do
ti:=time(multEC(i,IntC)):
TotalTime:=append(TotalTime,ti);
HISTORY:=0:
end_for:

for k from 1 to 10 do:
HISTORY:=0:
KnownBits:=BinC[1..159]:
KnownBitsInt:=text2int(KnownBits,2):
KnownTimes:=[]:
for i from 1 to 500 do
tik:=time(multEC(i, KnownBitsInt)):
KnownTimes:=append(KnownTimes,tik):
end_for:
t:=TotalTime-KnownTimes:

vart:=stats::variance(t):
for j from 1 to 1 do
  NextBit0:=KnownBits."0":
  NextBit1:=KnownBits."1":
  INextBit1:=text2int(NextBit1,2):
  INextBit0:=text2int(NextBit0,2):
  TimeNB1:=[]:
  TimeNB0:=[]:
  for l from 1 to 500 do
    TimeNB1:=append(TimeNB1, time(multEC(l, INextBit1))):
    TimeNB0:=append(TimeNB0, time(multEC(l, INextBit0))):
  end_for:
  tprime:=TimeNB1-KnownTimes:
  TimeDiff1:=t-tprime:
  v1:=stats::variance(TimeDiff1):
  print(vart);
  print(v1);
  if (v1<vart)
  then KnownBits:= NextBit1:
     KnownTimes:= TimeNB1:
     t:=TotalTime-KnownTimes:
     vart:= stats::variance(t):
```

```
      print(INextBit1);
   elif (vart<v1)
   then KnownBits:=NextBit0:
      KnownTimes:= TimeNB0:
      t:=TotalTime-KnownTimes:
      vart:= stats::variance(t):
      print(INextBit0);
   else print("Same Variance") end_if;
end_for:
end_for:
```

Elliptic Curve attack with small coeffcient

```
IntC:=57;
BinC:=int2text(IntC,2):
KnownBits:=BinC[1..5]:
KnownBitsInt:=text2int(KnownBits,2):

TotalTime:=[];
for i from 1 to 500 do
ti:=time(multEC(i,IntC)):
TotalTime:=append(TotalTime,ti);
HISTORY:=0:
end_for:

for k from 1 to 10 do:
KnownBits:=BinC[1..5]:
KnownBitsInt:=text2int(KnownBits,2):
KnownTimes:=[]:
for i from 1 to 500 do
tik:=time(multEC(i, KnownBitsInt)):
KnownTimes:=append(KnownTimes,tik):
end_for:
t:=TotalTime-KnownTimes:

vart:=stats::variance(t):
for j from 1 to 1 do
  NextBit0:=KnownBits."0":
  NextBit1:=KnownBits."1":
  INextBit1:=text2int(NextBit1,2):
  INextBit0:=text2int(NextBit0,2):
  TimeNB1:=[]:
  TimeNB0:=[]:
  for l from 1 to 500 do
    TimeNB1:=append(TimeNB1, time(multEC(l, INextBit1))):
    TimeNB0:=append(TimeNB0, time(multEC(l, INextBit0))):
  end_for:
  tprime:=TimeNB1-KnownTimes:
  TimeDiff1:=t-tprime:
  v1:=stats::variance(TimeDiff1):
  if (v1<vart)
```

```
  then KnownBits:= NextBit1:
     KnownTimes:= TimeNB1:
     t:=TotalTime-KnownTimes:
     vart:= stats::variance(t):
     print(INextBit1);
  elif (vart<v1)
  then KnownBits:=NextBit0:
     KnownTimes:= TimeNB0:
     t:=TotalTime-KnownTimes:
     vart:= stats::variance(t):
     print(INextBit0);
  else print("Same Variance") end_if;
end_for:
end_for:
```

Chapter C: Mathematica Files

C.1 EllipticCurve.m

```
(* This computes using Elliptic Curves over a field of definition
   that is a prime (finite) field of order p. The curve is in
   Weierstrass form y^2 = x^3 + ax + b and the points are also
   in Weierstrass (affine) form. *)

BeginPackage["EllipticCurve`"]

ECPt::usage=\
"ECPt[{x, y}, {a, b}, p] represents the point {x, y} on the elliptic \
curve y^2 = x^3 + ax + b over the integers mod p. The operations of \
addition and multiplication by an integer are implemented.";

RandomECPt::usage=\
"RandomECPt[{a, b}, p] returns a random point on the elliptic curve \
y^2 = x^3 + ax + b over the integers mod p. Before computing, this \
performs a sanity check to see if (not implemented) \
the curve is non-singular.";

IdentityECPt::usage=\
"IdentityECPt[{a,b}, p] returns the identity point on the specified
                curve.";

OrderECPt::usage=\
"OrderECPt[ECPt[...]] computes the order of the point on the elliptic \
curve.";

Begin["`Private`"]

Format[ECPt[{x_,y_},{a_,b_},p_]] := {x,y};
Format[ECPt[Infinity,{a_,b_},p_]] := {0};

ECPt/:ECPt[{x1_,y1_},{a_,b_},p_] + ECPt[{x2_,y2_},{a_,b_},p_] := \
        Module[{m,x3,y3,xgcd},
                If[x1 == x2 && Mod[y1+y2,p] == 0,
                        ECPt[Infinity,{a,b},p],
                (* Else *)
                        m = If[x1 == x2,
```

```
                                xgcd = ExtendedGCD[2*y1,p];
                                Mod[(3*x1^2+a)*xgcd[[2,1]],p],
                                (* Else *)
                                xgcd = ExtendedGCD[x1-x2,p];
                                Mod[(y1-y2)*xgcd[[2,1]],p]];
                        If[xgcd[[1]] != 1,
                                Print["Found factor of ",xgcd[[1]]];
                                Abort[]];
                        x3 = Mod[m^2 - x1 - x2, p];
                        y3 = Mod[-m*(x3-x1) - y1, p];
                        ECPt[{x3,y3},{a,b},p]]]

ECPt/: ECPt[Infinity,{a_,b_},p_] + ECPt[pt2_,{a_,b_},p_] := \
        ECPt[pt2,{a,b},p]

ECPt/: -ECPt[{x_,y_},{a_,b_},p_] := ECPt[{x,Mod[-y,p]},{a,b},p]

ECPt/: n_Integer * ECPt[{x_,y_},{a_,b_},p_] := \
        Module[{mult=n, accum=ECPt[Infinity,{a,b},p],
                powpt=ECPt[{x,y},{a,b},p]},
        If[mult < 0, powpt = -powpt; mult = -mult];
    Print "Here";
        While[mult != 0,
                If[OddQ[mult], accum = accum + powpt; mult = mult-1];
                powpt = powpt + powpt;
                mult = mult/2];
        accum]
IdentityECPt[{a_,b_},p_] := ECPt[Infinity,{a,b},p]

RandomECPt[{a_,b_},p_] := \
        Module[{x,y},
        (* Sanity check first: Is {a,b} a non-singular curve *)
(* Don't check for primality so we can use for EC factoring
        If[!PrimeQ[p],
                Message[ECPt::notprime,p];
                Return[ECPt[Infinity,{a,b},p]]]; *)
        (* Find the first value of x for which x^3 + ax + b
                has a square root *)
        x = Random[Integer,{0, p-1}];
        While[JacobiSymbol[x^3+a*x+b,p] != 1, x=Mod[x+1,p]];
        y = PowerMod[x^3+a*x+b,1/2 ,p];
        ECPt[{x,y},{a,b},p]];

ECPt::notprime = "ECPt only defined for prime fields,
        `1` is not prime";
OrderECPt[pt:ECPt[{x_,y_},{a_,b_},p_]] := \
        Module[{accum = pt, i=1, id = IdentityECPt[{a,b},p]},
        While[(accum =!= id) && (i <= p+1+2*Sqrt[p]),
                accum += pt; i++];
        i]

End[];  (* EllipticCurve`Private` *)
(* Protect[the names] *)
Print[Names["EllipticCurve`*"], " have been defined."];
```

```
EndPackage[]   (* EllipticCurve` *)
```

C.2 ErrorAnalysis.nb

```
CInt = FromDigits[RandomChoice[{0,1}, 160], 2]
Points=Table;
Points << C:\Users\Clarice\SkyDrive\Documents\School\
        MathematicaFiles\521Points500;
a=-3;
b=24551555460089438177402939151974517847691080581611912380 65;
p=62771017353866807638357894232076664160839087003903249612 79;
TotalTime=Table[First[Timing[CInt*Points[[i]]]], {i,500}];
37656009605505155544833212089000097468959647691 6


Points=Table;
Points << C:\Users\Clarice\SkyDrive\Documents\School\
        MathematicaFiles\521Points500;
<<    C:\Users\Clarice\SkyDrive\Documents\School\
        MathematicaFiles\EllipticCurve.m;
a=-3;
b=24551555460089438177402939151974517847691080581611912380 65;
p=62771017353866807638357894232076664160839087003903249612 79;
CInt=76560096055051555448332120890000974689596476916;
StdDevs=Table[StandardDeviation[
Table[First[Timing[CInt*Points[[i]]] ], {30}]], {i, 500}]
```

Chapter D: Python Files

D.1 DataAnalysis.py

```
import pickle
import statistics
import io
import time
import numpy

yvalsFile = open("yvals.pickled", "rb")
YVals = pickle.load(yvalsFile)
yvalsFile.close()

#256 Bit Expon
#exponFile = open("expon.pickled", "rb")
#Expon = pickle.load(exponFile)
#exponFile.close()


#1024 Bit Expon
p = 9091213529597818878440658302600437485892608310328358720428512168960411528640933367824950788367956756806141
q = 1438592591100452657278091262844293358778990021676278832009141724293243601330041167020032408287779702524 99
n = 7403756347956171282804679609742957314259318888923128084936232638972765034028266276891996419625117843995894330502127585370118968098286733173273108930900552505116877063299072396380786710086096962537934563796359
Expon=1554692822276335430311223174401846775739366240132801343485197860001375635641618469391445338490206408278722508666139077082054997520145660372241893073209000564478473914777473699720760705363972291621713537820140391249325698078792509941722551375799229090002415075827823925458170682411135696389934474204448361086 50
BinExpon=bin(Expon)

TotalTime = numpy.array([], dtype=object)
TotalTime2= numpy.array([], dtype=object)
StdDevs = numpy.array([], dtype=object)
#Averages = numpy.array([], dtype=object)
```

```
#Compute Time Data or Standard Deviations.
#Must change indents and remove '#'
for y in range(500):
    #Reps = numpy.array([], dtype=object)
    #for x in range(30):
    tic = time.perf_counter()
        #YVals[1]**5%n
    pow(YVals[y], Expon, n)
    toc = time.perf_counter()
        #Reps = numpy.append(Reps, toc-tic)
    TotalTime=numpy.append(TotalTime, toc-tic)

    #StdDevs= numpy.append(StdDevs, numpy.std(Reps))
#Averages = numpy.append(Averages,[statistics.mean(Reps[i:i+20])
    #for i in range(1, 200, 20)])

print(TotalTime)
print(statistics.mean(TotalTime))
print(numpy.std(TotalTime))
#print(StdDevs)
#print(Averages)
```

D.2 ExponLengthVStdDev.py

```
import io
import pickle
import numpy
import time

yvalsFile = open("yvals.pickled", "rb")
YVals = pickle.load(yvalsFile)
yvalsFile.close()

ExpListFile = open("expon500.pickled", "rb")
Expons=pickle.load(ExpListFile)
ExpListFile.close()

p = 909121352959781887844065830260043748589260831032835872042851216896041152864093336782495078836795675680614
q = 143859259110045265727809126284429335877899002167627883200914172429324360133004116702003240828777970252499
n = 740375634795617128280467960974295731425931888892312808493623263897276503402826627689199641962511784399589433050212758537011896809828673317327310893090055250511687706329907239638078671008609696
2537934563796359

StdDevs=numpy.ndarray(shape=(52,10), dtype=object)
for i in range (52):
    print(Expons[i])
    for x in range(10):
        Times=numpy.array([], dtype=object)
```

```
        for y in YVals:
            tic=time.perf_counter()
            pow(y,Expons[i],n)
            toc=time.perf_counter()
            Times=numpy.append(Times, toc-tic)
        tSD=numpy.std(Times)
        StdDevs[i,x]=tSD

print(StdDevs)
```

D.3 RSATimeAttack.py

```
import pickle
import io
import time
import numpy
import statistics

yvalsFile = open("yvals.pickled", "rb")
YVals = pickle.load(yvalsFile)
yvalsFile.close()
#256-bit Exponent
#exponFile = open("expon.pickled", "rb")
#Expon = pickle.load(exponFile)
#exponFile.close()

p = 9091213529597818878440658302600437485892608310328358720428512168960411528640933367824950788367956756806141
q = 1438592591100452657278091262844293358778990021676278832009141724293243601330041167020032408287779702524 99
n = 7403756347956171282804679609742957314259318888923128084936232638972765034028266276891996419625117843995894330502127585370118968098286733173273108930900552505116877063299072396380786710086096962537934563796359

#1024-bit Exponent
Expon=1554692822276335430311223174401846775739366240132801343485197860001375635641618469391445338490206408278722508666139077082054997520145660372241893073209000564478473914777473699720760705363972291621713537820140391249325698078792509941722551375799229090002415075827823925458170682411135696389934474204448361086 50
print(Expon)
BinExpon=bin(Expon)
print(len(BinExpon))
#160-bit Exponent
#Expon = 737594632511118677735731068268133317738981665915
#BinExpon=bin(Expon)
#print(len(BinExpon))


# Repeat the attack 10 times
```

```
for j in range(10):
        TotalTime = numpy.array([], dtype=object)

        #Find Total Time (assumed given)
        for x in range(500):
                RepsTot= numpy.array([], dtype=object)
                for z in range(1):
                        tic = time.perf_counter()
                        #x**Expon%n alternate for exponentiation
                        pow(YVals[x], Expon, n)
                        toc = time.perf_counter()
                        RepsTot = numpy.append(RepsTot, toc-tic)
                TotalTime = numpy.append(
                        TotalTime,statistics.mean(RepsTot))

        k=len(BinExpon)-1
        KnownBits = BinExpon[2:k]
        KnownTime = numpy.array([], dtype=object)
        KnownBitsInt = int(KnownBits, 2)

        #Find time known given some bits are known
        for x in range(500):
                Reps = numpy.array([], dtype=object)
                for z in range(1):
                        tic = time.perf_counter()
                        #x**KnownBitsInt%n
                        #alternate for exponentiation
                        pow(YVals[x], KnownBitsInt, n)
                        toc = time.perf_counter()
                        Reps = numpy.append(Reps, toc-tic)

                KnownTime = numpy.append(
                        KnownTime,statistics.mean(Reps))
        #print(KnownTime)
        t= TotalTime-KnownTime
        vart = numpy.var(t)

        #Compute times for both guesses for the next bit
        for j in range(len(BinExpon)-len(KnownBits)-2):
                NextBit0 = KnownBits+'0'
                NextBit1 = KnownBits+'1'
                INextBit0 = int(str(NextBit0), base=2)
                INextBit1 = int(str(NextBit1), base=2)
                #print(INextBit1)
                #print(INextBit0)
                TimeNextBit1 = numpy.array([], dtype=object)
                TimeNextBit0 = numpy.array([], dtype=object)
                for y in range(500):
                        Reps = numpy.array([], dtype = object)
                        Reps2 = numpy.array([], dtype = object)
                        for z in range(5):

                            tic2 = time.perf_counter()
                            #y**INextBit0%n
```

```
                    pow(YVals[y], INextBit0, n)
                    toc2 = time.perf_counter()
                    Reps2 = numpy.append(Reps2, toc2-tic2)

                    tic = time.perf_counter()
                    #y**INextBit1%n
                    pow(YVals[y], INextBit1, n)
                    toc = time.perf_counter()
                    Reps = numpy.append(Reps, toc-tic)

                TimeNextBit1 = numpy.append(
                        TimeNextBit1,statistics.mean(Reps))
                TimeNextBit0 = numpy.append(
                        TimeNextBit0,statistics.mean(Reps2))

            tPrime = TimeNextBit1-KnownTime
            TimeDiff1 = t-tPrime
            v1=numpy.var(TimeDiff1)
            print(vart)
            print(v1)


            #Comparison of variances for the next bit
            if v1<vart:
                KnownBits = NextBit1
                #print(NextBit1)
                KnownTime=TimeNextBit1
                t=TotalTime-KnownTime
                vart = numpy.var(t)
            elif v1>vart:
                KnownBits = NextBit0
                #print(NextBit0)
                KnownTime=TimeNextBit0
                t=TotalTime-KnownTime
                vart = numpy.var(t)
            else:
                print('Same variance')

print(int(KnownBits,2))
```

Bibliography

[1] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in CryptologyCRYPTO96*, pages 104–113. Springer, 1996.

[2] Jonathan Rosenberg. ellcurve.mn. `http://www.math.umd.edu/~jmr/456/ellcurve.pdf`.

[3] Masumeh Damrudi and Norafida Ithnin. Numerical analysis of parallel modular exponentiation for RSA using interconnection networks. *SCIENCEASIA*, 39:103–106, 2013.