Uncovering Vulnerabilities in Smartphone Cryptography: A Timing Analysis of the Bouncy Castle RSA Implementation

Sarani Bhattacharya[1], Dilip Kumar Shanmugasundaram Veeraraghavan[2], Shivam Bhasin[3], and Debdeep Mukhopadhyay[4]

[1] imec, Belgium
[2] sarani.bhattacharya@imec.be
[3] KU Leuven, Belgium
[4] NTU Singapore
[5] Indian Institute of Technology, Kharagpur

Abstract. Modern day smart phones are used for performing several sensitive operations, including online payments. Hence, the underlying cryptographic libraries are expected to adhere to proper security measures to ensure that there are no exploitable leakages. In particular, the implementations should be constant time to prevent subsequent timing based side channel analysis which can leak secret keys. Unfortunately, we unearth in this paper a glaring timing variation present in the BouncyCastle implementation of RSA like ciphers which is based on the BigInteger Java library to support large number theoretic computations. We follow up the investigation with a step-by-step procedure to exploit the timing variations to retrieve the complete secret of windowed RSA-2048 implementation. The entire analysis is possible with a single set of timing observation, implying that the timing observation can be done at the onset, followed by some post processing which does not need access to the phone. We have validated our analysis on Android Marshmallow 6.0, Nougat 7.0 and Oreo 8.0 versions. Interestingly, we note that for newer phones the timing measurement is more accurate leading to faster key retrievals.

1 Introduction

Smart phones have become indispensable in day-to-day activities of people across the globe. Numerous applications thus communicate with several central service providing servers and hence needs to support standardized encryption schemes to protect against eavesdroppers. It also implies that the applications and libraries supporting these cryptographic operations should satisfy state-of-the-art secure implementation guidelines, and provide protection against side-channel attacks. These attacks can compromise the cryptographic secret by observing execution footprint of the theoretically secure algorithms. Deployment and usage of insecure libraries can prove to be detrimental leading to huge financial damage. The most recent and interesting works on security violation on an Android includes location tracking by the battery measurements which are being affected by the effect of signal strength at that particular location as in [9], using signal processing and machine learning to reconstruct acoustic signal from

gyroscope sensor measurements as in [8], construction and retrieving password using machine learning on the sensor data measurements while the passwords are entered [3], while the authors in [12] provide a survey of existing threats and vulnerabilities of the mobile computing platforms.

The timing side-channel attack was revisited for mobile devices in [13], where the authors target the T-table based implementation of AES. The authors in [2] showed that the doubling and addition operations in an Elliptic curve scalar multiplication can be distinguished by using Electromagnetic side-channel traces on the smart phone. A similar full key extraction has been demonstrated on ECDSA in [5] using the EM probes and USB charging cable. Similar discussions on RSA and ECC can be found in [10] where the EM signals leak a large part of the secret. In [7], the authors performed extensive set of experiments to make cross-core cache attacks such as Prime+Probe, Flush+Reload, Evict+Reload, and Flush+Flush feasible on a non-rooted Android application. The defense to these forms of cache based attacks have been proposed in [11] where the effect of the keystrokes timing attacks are being mitigated by adding a large number of fake keystrokes in between. While in [1], the authors illustrate EM based secret exponent retrieval from a single decryption on any arbitrary ciphertext in the OpenSSL fixed-window constant-time implementation of RSA and does not require any previous knowledge of the cache organization.

The existing works in the literature either requires separate EM based monitoring setup or access to the internal sensors or knowing architectural details. Since Android 7.0 Operating System optimizations have restricted any user app's stealthy access to the sensors such as battery, location sensors and many more. In this paper, we explore the non-constant time vulnerable implementation of BouncyCastle cryptographic library in presence of timing measurements as side channel. Our timing side channel vulnerability analysis neither requires the knowledge of the underlying architecture nor access to the restricted sensor data and has been validated on Android Marshmallow 6.0, Nougat 7.0 and Oreo 8.0 versions.

1.1 Overview of the paper

There are a few standardized cryptographic library implementation which have been developed using the recommended algorithmic specification. In this work, we target one such widely popular crypto library `BouncyCastle` which follows NIST standard and includes FIPS140-2 Level1 certified streams to perform crypto application on the widely popular Android phones. We show that the cipher implementation supported in the library are vulnerable against timing attacks. In common practice data used in cryptographic algorithms could be arbitrarily large in size. When such operations involving large data size are executed on Android, instances of a specialized class named `BigInteger` in `java.math` package is used for data representation. `BigInteger` class provides a favorable feature wherein the size of the data stored isn't constrained and additionally, this class also constitutes of many important member functions which are often useful to perform faster computations.

In this paper, we analyze the vulnerability of one such member function `modPow`, the modular exponentiation function which is employed in popular encryption algorithms such as RSA, ElGamal etc. We target the implementation of RSA encryption algorithm using `modPow` as a subroutine call and uniquely retrieve the private key used for decryption. In this analysis we require timing measurements on entire RSA decryption for BouncyCastle crypto call. This measurement of timing being non-constant and dependent on the data as well as the algorithm, this measurement is subjected to statistical analysis on three different phases which eventually leads to full key leakage. The first phase is to identify the sparse multiplication locations for the target windowed exponentiation. Once these consecutive locations are identified, we define theoretical constraints on the surrounding bit locations going by a worst-case analysis. This eventually leaks partial bits for each of these windows. In the final phase, we follow a Difference of Mean analysis on the remaining bits which leaks the remaining unknown key bits in the window. The Difference of Mean approach is utilized on the fact that the modular reduction on Montgomery Multiplication is non-constant time based on the input ciphertexts and thus bins can be separated based on the number of reduction statements executed by these inputs.

We support each of these phases with detailed validations in the result section, where we target the RSA-2048 bit secret and retrieve the bits subsequently. This analysis neither requires any architectural specification nor any micro-architectural event dependency. We have ported this vulnerability on multiple Android phones to demonstrate that this attack is platform independent.

1.2 Threat Model

In this paper, we focus on unearthing a timing side channel vulnerability on BouncyCastle cryptographic implementation. The threat model requires a spy app with malicious intention which can send ciphertexts to the decryption engine and receives the output plaintext and in the meanwhile, it measures the execution time for cryptographic primitives. The timing measurements for our monitor app are observed using `java.lang.System.nanoTime()` method in Java which returns the value of the most precise system timer, in nanoseconds. We invoke the decryption calls and the start and end times of the decryption are observed.

Responsible Disclosure *While we found this timing vulnerability of BouncyCastle and were working on key recovery, we have intimated the bug bounty team of BouncyCastle regarding the said vulnerability. They have acknowledged the timing vulnerability.*

2 Target Algorithm

In this work, we target the cryptographic implementations supported by the BouncyCastle provider. `BouncyCastle Crypto API` consists of lightweight crypto API which implements a set of all underlying cryptographic algorithm. Android

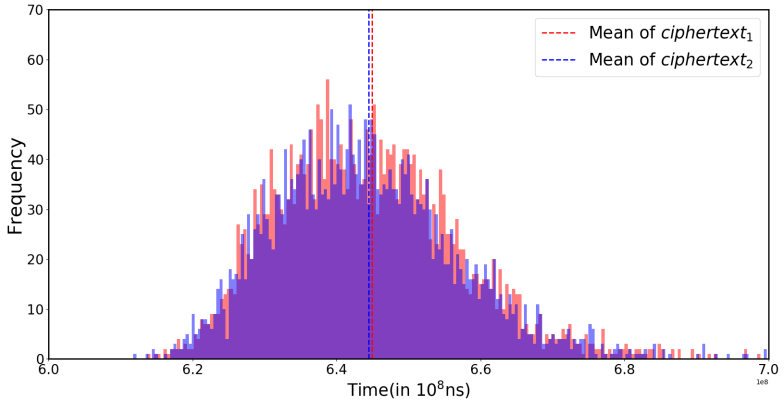


Fig. 1: Non-constant timing behaviour of two ciphertexts exponentiation over same secret

OS in some version of the smart phones incorporate a customized version of this library named as `SpongyCastle`. The public key exponentiation of the BouncyCastle implementation has been efficiently coded and documented throughout the BouncyCastle APIs, and the underlying mathematical operations are being performed using the `BigInteger` library from Java.

Before going into actual details regarding the `BigInteger` algorithmic implementations we start with a case study. We performed a simple experiment where we kept the secret key to be constant and we varied the input ciphertext. Figure 1 shows the distribution of timing samples received over two ciphertexts. The timing samples for ciphertext plotted in red are consistently higher than the timing values for the ciphertext plotted in blue. This is repeatable and thus confirms that there are some component codes in the BouncyCastle implementation which contributes to the timing difference. This observation motivated us to investigate the real reason behind this non-constant time implementation which is getting affected by the change in input ciphertext though the secret exponent has been kept constant in both cases. The following section gives a brief algorithmic overview of the underlying BigInteger window algorithm used by BouncyCastle.

2.1 The BigInteger window algorithm

The `modPow` function is the Java implementation of the modular exponentiation operation in BigInteger library. This function is invoked while performing the sensitive modular exponentiation in RSA like ciphers which eventually rely on the OpenJDK implementation. [6]. The `BigInteger` library of Java comes inbuilt with Android on smart phone platforms.

[6] The exact implementation can be found in `src/share/classes/java/math/BigInteger.java`

The function `modPow` implements windowed modular exponentiation through an algorithm adapted from Colin Plumb's C library. The `modPow` algorithm sequentially moves with the exponent bits and squares the result buffer for each bit of the exponent. Though the squaring is executed for each bit irrespective of its value, multiplication is selectively performed only once in a window and the position of multiplication is decided by the last encountered 1 in the window. The algorithm scans the exponent from the bit immediately after the Most Significant Bit (MSB), and gradually progresses by performing squaring for each bit until it encounters a leading 1, which marks the start of the window. This initiates a scan for the trailing 1 (within the length of the window) to decide upon the position upto which the multiplication going to happen. The following is a general example on 3-bit window (W = 3) for input m.

- When next bits are `100`: the sequence of operations are {`square, multiply by input` m^1`, square, square`}
- When next bits are `101`: the sequence of operations are {`square, square, square, multiply by` m^5}
- When next bits are `110`: the sequence of operations are {`square, square, multiply by` m^3`, square`}
- When next bits are `111`: the sequence of operations are {`square, square, square, multiply by` m^7}

Consider an example sequence 10101, leaving out the MSB, the sequence of operations for 0101 with the windowed method would be {`square, square, square, square, multiply by` m^5}= $((((m^2)^2)^2)^2)m^5 = m^{21}$, in contrast to naive square and multiply operation would have been {`square, square, multiply, square, square, multiply`}= $(((((m^2)^2)m)^2)^2)m = m^{21}$. It is to be noted, `modpow` involves only one multiplication per window.

2.2 Vulnerability of modPow algorithm

The windowed exponentiation algorithm as discussed above performs at most one multiplication per window depending on the bits in that window. This particular step incurs an extra computation in terms of that multiplication being performed or not. If the multiplication positions can be distinguished from the squarings, then it eventually reveals information about the position of these windows. The entire timing analysis can be performed with only one set of timing observations from the executions of BouncyCastle. The algorithm can be primarily divided into the following three parts. Figure 2 illustrates the overall idea of how bits get leaked in three phases.

Phase 1: Determining the positions of trailing Multiplication *The first task is to identify the positions of all such multiplication for the entire secret exponent.*

Phase 2: Partial key bits retrieval constraining the window size *Knowing the location of multiplications and the respective bit distances between two successive multiplication position could lead to multiple bit recovery. We define the constraints considering various cases that could arrive, since the start and end locations*

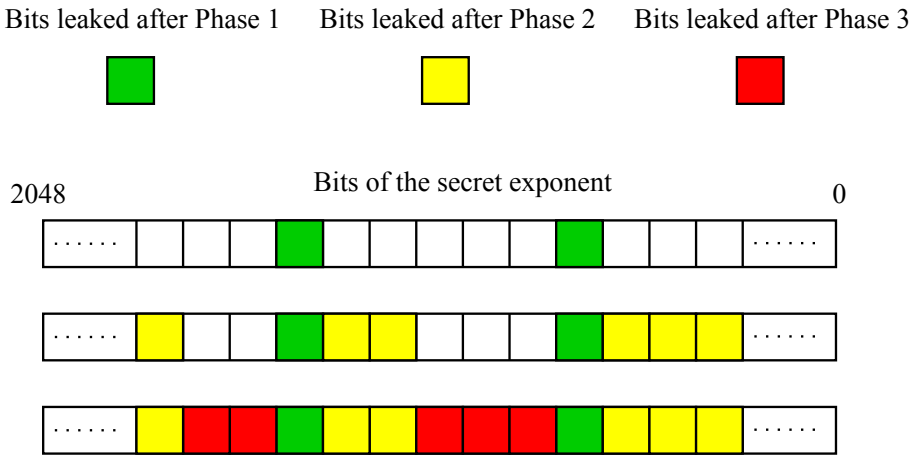


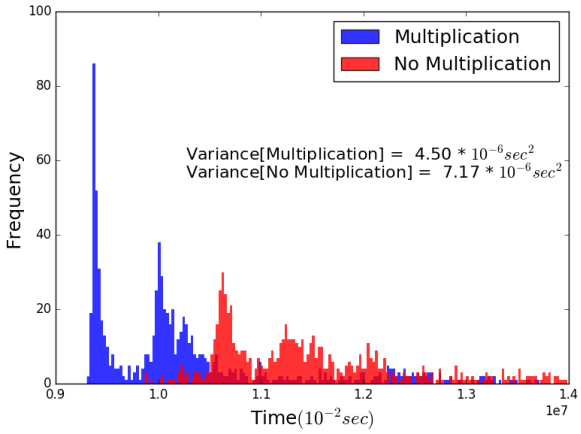
Fig. 2: Bits retrieved after each phase analysis

of each of the windows cannot be predicted at this stage. We follow by a worst-case analysis to derive the surrounding bits. A similar analysis with respect to CRT-RSA in [4] works on a different version of windowed mod-exp algorithm than the one used in `modPow` function of `BigInteger` class. In this paper we target the BouncyCastle and have successfully devised a `more efficient partial key recovery phase` than reported earlier in the mentioned paper.

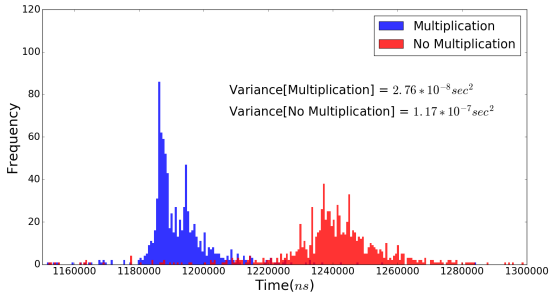
Phase 3: Full key recovery by Difference of Mean Analysis In this phase, we guess the partial unknown bit sequence and perform a Difference of Mean analysis based on the timing observations and split the observed timings into two bins based on a guess on the target key bit. The number of Montgomery reductions executed after each of the squaring and multiplication are non-constant and contribute to timing difference between two bins.

3 PHASE 1: Determining the position of Multiplication in windowed exponentiation

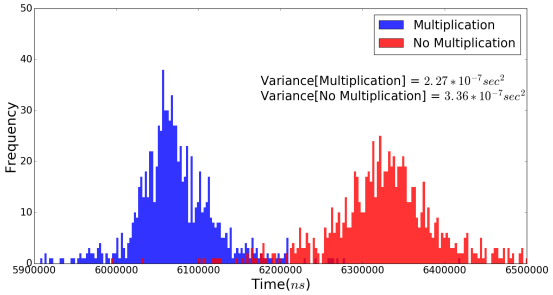
The previous section gives a description of the windowed exponentiation algorithm of RSA of the BigInteger library. According to the window algorithm, for a W-bit window of the exponent, there exists only one multiplication and W squarings. The position of this trailing multiplication happens in between the squarings. Also the number of squarings after which multiplication has to take place, ie, the position of the trailing multiplication depends on the final occurrence of bit 1 in the window. *This also means that information of location of these trailing multiplication gives away further information about the exponentiation window, thus the primary motive is to differentiate the location of such multiplications from the adjacent squarings.*



(a) Motorola G2



(b) Huawei Honor 8 pro



(c) Motorola G5

Fig. 3: Identifying multiplication positions of the secret exponent on three different platforms

3.1 Timing attack on Windowed Exponentiation Algorithm

The aim of this phase is to identify the location of multiplications happening over the sequence of bits from timing observation. Similar to Kocher's [6] timing attack on simple modular exponentiation, we follow by a difference in variance

analysis for multiplication positions in the windowed algorithm. This is an iterative procedure where the start and end bits of the consecutive windows Win are not yet known to the adversary, thus she begins with guesses made for each bit position. The event considered as the guess is that there `has` or `has not` been a multiplication performed at that position.

In this process, the total execution timing over a set of inputs performing the entire exponentiation on the secret exponent is observed beforehand. Since this is an iterative procedure, the analysis progresses bit by bit with the basic assumption that squaring and multiplication locations before the target bit position is already revealed. This partially known sequences of operation followed by guesses for the target location are simulated and the hypothetical timings are noted. The intuition is that the differences of previously observed total timings and the simulated ones for two different event guesses forms a distribution of difference timings, and the distribution with lesser variance validates the correct guess (elaborated in Appendix A).

Therefore as established in [6], correct guess causes reduction in variance, while a wrong guess affects the variance to increase. In windowed algorithm there are much fewer multiplications, since the nature of delayed multiplication only once per window. Also on an Android, the multiplications take considerably more time than squarings while running the windowed algorithm. The timings as observed over the Android application shows low variance if the multiplication has been correctly guessed at that position otherwise reveals that there has been no multiplication performed at the particular bit position. In simple words, the variance of the difference in timing reveals the position where multiplications take place.

3.2 Detection of Multiplications in Windowed Algorithm

We observed timing samples over our monitor app in Android and calculated the variance over the sample values over both guesses. Figure 3a illustrates the example location where the multiplication is getting guessed correctly on Motorola MotoG2[7]. As the multiplication was already taking place at that position of the secret, the variance of difference of timings when multiplication has been guessed is less than the case where the time till the squaring is considered. The values of variance for the respective distributions observed for each guess is annotated in the Figure 3a.

Figures 3b and 3c illustrate the cases where multiplications are identified on Huawei Honor 8 Pro[8] and MotoG5[9] respectively. The distributions in Figure 3 denote the distribution of two guesses and the variance of each guess is annotated in the figure. The `x-axis` denotes the difference in timing observed from

[7] Motorola MotoG2 2nd Gen with Qualcomm MSM8226 Snapdragon 400 Quad-core 1.2 GHz Cortex-A7 with Android Marshmallow 6.0

[8] Huawei Honor 8 Pro with Octa-core (4x2.4 GHz Cortex-A73 4x1.8 GHz Cortex-A53) processor with Android Oreo 8.0

[9] Motorola MotoG5 with Qualcomm MSM8937 Snapdragon 430 Octa-core 1.4 GHz Cortex-A53 processor with Android Nougat 7.0

the Android application and the simulated timing with the guess, whereas the `y-axis` denotes the frequency of those difference of timing observations. The guess with the lower variance is the correct guess.

The information of the locations where the multiplications have taken place are extremely important for the partial as well as the full recovery of the RSA secret. Since the location of trailing multiplication also reveals the last occurrence of a set(1) bit in the window, these bits themselves have to be 1. Also there are some more bits in the window that can be retrieved just by knowing the position of the multiplication which we discuss in the next section.

4 PHASE 2: Partial Key Recovery Analysis

Unlike simple modular exponentiation algorithm, a multiplication in windowed exponentiation imposes constraint on several bits surrounding it. If the i^{th} multiplication say l_i for window Win_i (with length W) takes place at bit position say b_j, then the start and end location of Win_i can be intuitively predicted to be less than $\mathrm{W}-1$ bits away from l_i in both directions ie, some bits around the bit location b_j. This follows from the simple fact that l_i (bit location b_j) belongs to Win_i and each window is W bits wide. Since multiplication for a window is executed at the last occurrence of bit 1 in the window, and also say if the window Win_i includes some more bits after l_i, then it means that all the bits following b_j in Win_i must be 0s. Thus it is intuitive, that the location of multiplication carries information about several bits surrounding it. In the rest of this section, we discuss the importance of these bit locations and a procedure to recover more bits of the secret key.

4.1 Importance of multiplication locations

Let l_i and l_{i+1} be the index locations of two adjacent multiplications belonging to windows Win_i and Win_{i+1} separated by k bits from bit location b_j to b_{j+k} (refer to Figures 4 and 5 for illustration). As both l_i and l_{i+1} indicate position of bits representing the multiplications of their respective windows, both of them should be 1s and all the bits in between them are yet unknown. Based on the relative bit distance k between the known bits l_i and l_{i+1} one or more unknown bits of the key can be predicted. We categorize the relative distance into two cases: $l_{i+1} - l_i \geq \mathrm{W}$ and $l_{i+1} - l_i < \mathrm{W}$.

Case 1: $\boldsymbol{l_{i+1} - l_i = k \geq \mathrm{W}}$ The following analysis reveals more bit in the bit locations $b_j, \cdots, b_{j+k}$ as illustrated in Figure 4. For the next part we will use the index locations such as l_i and l_{i+1} as an alias to bit locations b_j and b_{j+k} respectively to avoid confusion.

- In order to predict the boundaries of Win_{i+1}, we find that as the bit l_{i+1} is part of Win_{i+1}, the beginning of window Win_{i+1} could be any one of the bits in the range $\{b_{(j+k)-\mathrm{W}+1}, \cdots, b_{j+k}\}$.

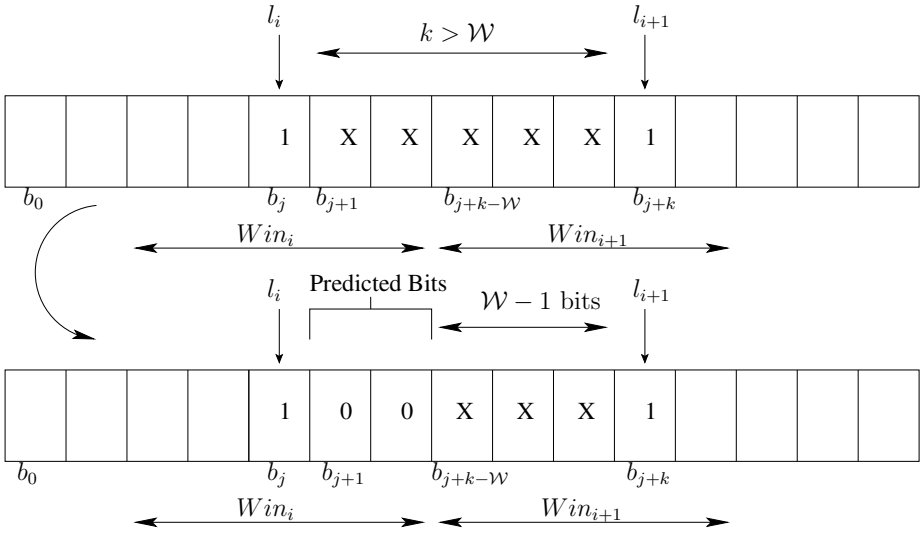


Fig. 4: Case 1

- If $(l_{i+1} - l_i) > \mathrm{W}$, neither of the bits between b_j and $b_{(j+k)-\mathrm{W}}$ can belong to Win_{i+1}. These bits from $b_j, b_{j+1} \cdots, b_{(j+k)-\mathrm{W}}$ should either belong to Win_i or neither of the two windows.
- Now, considering that the multiplication for Win_i occurs at l_i, the bit b_j has the last set bit for the window Win_i. The bits following b_j which belong to window Win_i can be surely identified to be 0. Therefore if any of the bits between $b_{j+1}, \cdots, b_{(j+k)-\mathrm{W}}$ belongs to Win_i, then it must be 0.

Case 2: $l_{i+1} - l_i = k < \mathrm{W}$ The second case deals with the possibility where the next multiplication, l_{i+1}, occurs in some bit position within the next W bits from l_i ie, $k < \mathrm{W}$. In this case, the start bit of Win_{i+1} could be any bit between b_{j+1} and b_{j+k} and as Win_{i+1} is W bits wide, Win_{i+1} spans to some bits (at least $\mathrm{W} - k$ bits) after l_{i+1} as illustrated in Figure 5. In other words, l_{i+1} isn't the last bit of the Win_{i+1}. As we already know that multiplication occurs at the trailing 1 of the window, the last set bit of Win_{i+1} is at l_{i+1}, ie, b^{th}_{j+k} location. Therefore one or more bits of Win_{i+1} following b_{j+k} should all be 0s. With this information, though we cannot predict the start bit of Win_{i+1}, we could retrieve some trailing bits. Win_{i+1} window could start at any bit from b_{j+1} to b_{j+k} and it could end at any bit from $b_{j+\mathrm{W}}$ to $b_{(j+k)+\mathrm{W}-1}$. But irrespective of the start bit of Win_{i+1}, we can safely say that the bits from b_{j+k+1} to $b_{j+\mathrm{W}}$ would always be part of Win_{i+1} and would all be 0s.

While the first case, $(l_{i+1} - l_i) > \mathrm{W}$, leaks some bits of the key in-between l_i and l_{i+1}, the second case, $(l_{i+1} - l_i) < \mathrm{W}$, leaks some bits after l_{i+1}. And when $(l_i - l_{i+1})$ is equal to W, no extra bits are leaked. We call the bits that can be predicted with the information of multiplications' locations as `predictable` bits.

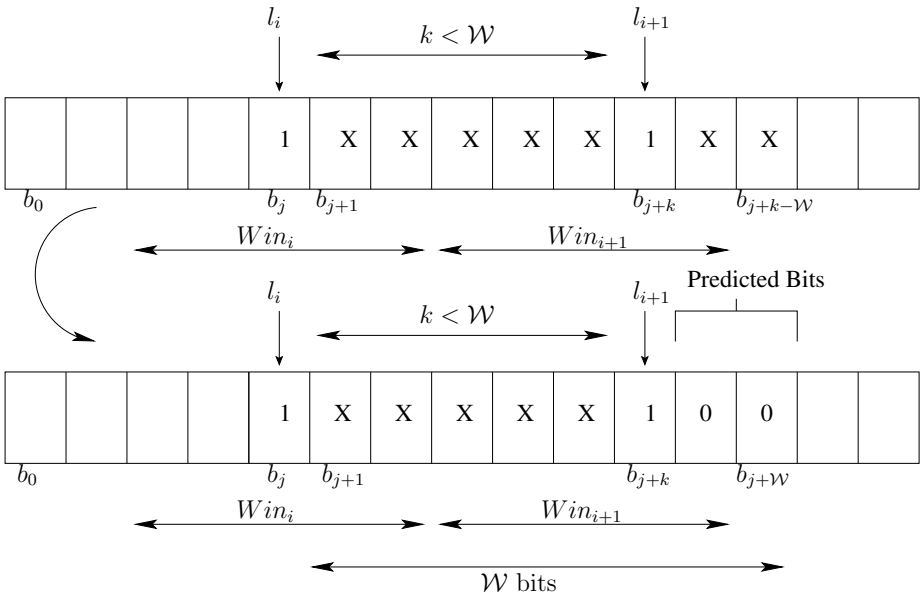


Fig. 5: Case 2

In the next subsection, we extend our analysis from two adjacent multiplications to all the multiplications of the key thereby retrieving all the `predictable` bits of the key.

4.2 Retrieving dependencies between two consecutive windows

In order to retrieve all the `predictable` bits of the secret key we consider not just two consecutive multiplications but all the multiplications of the key iteratively using the knowledge of bits from previous windows. Here, instead of categorizing the relative distance between l_{i+1} and l_{i+2}, we categorize the relative distance between the last known bit of Win_{i+1} and l_{i+2}. Let P_{i+1} be the Placeholder bit in this case which points the last known bit of Win_{i+1}. Just like the previous analysis, there can be two cases now $(l_{i+2} - P_{i+1}) > \mathrm{W}$ and $(l_{i+2} - P_{i+1}) < \mathrm{W}$. Figure 6 illustrates a case when locations of multiplications l_i, l_{i+1}, l_{i+2} satisfy the conditions $(l_{i+1} - l_i) < \mathrm{W}$ and $(l_{i+2} - P_{i+1}) < \mathrm{W}$ (Considering an example for RSA-2048 where W gets set as 7). By considering the condition $(l_{i+1} - l_i) < \mathrm{W}$ first, we find 3 bits of Win_{i+1} to be 0s. In the next step, instead of categorizing the relative distance between l_{i+2} and l_{i+1}, if we take into consideration the knowledge of the known bits then the distance between P_{i+1} and l_{i+2}, we would be able to recover more `predictable` bits of the key. Hence this modification increases the efficiency of our analysis when we consider multiple windows in succession. To maximize the efficiency of our analysis, we start from `MSB` of the key and successively consider multiplications by calculating the relative distance of l_{i+1} from the last known bit of W_i. By continuing the process till the *LSB* of the key, we can retrieve all *predictable* bits of the key.

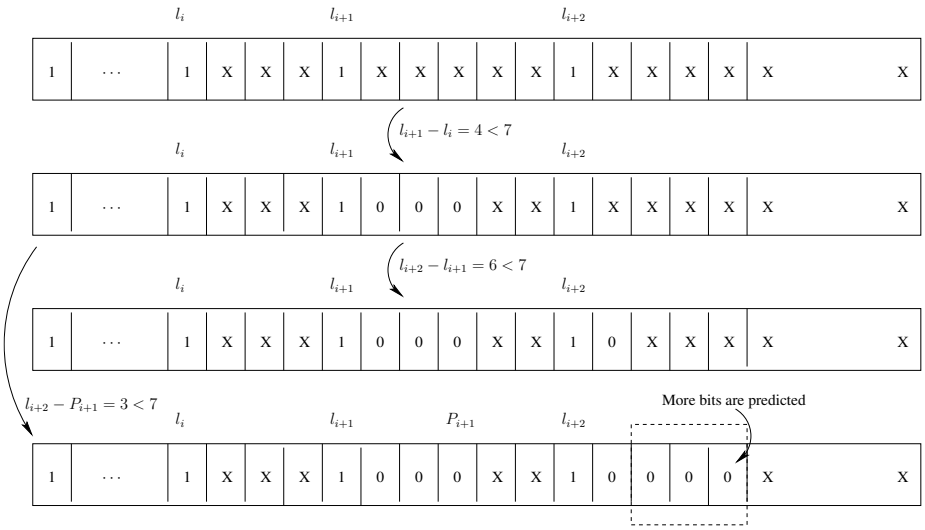


Fig. 6: Updating the placeholder bit P_{i+1}

4.3 Efficiency of our analysis

Continuing our analysis from the previous section, here we mathematically approximate expected number of bits we can predict by our analysis for a randomly chosen exponent. Assume that we have partially revealed the key till l_i using the relative distance method. P_i being the last known bit of Win_i, we denote $\mathrm{E(P}_i+)$ as the expected number of bits we could recover in the unknown part of the key starting from P_i down to the `LSB`. Based on the location of the next multiplication l_{i+1}, one or more bits of the key would be predicted by considering the relative distance between P_i and l_{i+1} and we can define $\mathrm{E}(n)$ as a recurrence relation in terms of $\mathrm{E}(n-1)$. In the first case we consider the situation where $k \geq \mathrm{W}$ and it ends up in a sum of two equation, where one part reveals some bits in between the l_i and l_{i+1}^{th} multiplication and the other part is a recursive call to the bits following l_{i+1}^{th} multiplication.

$$E(n) = \sum_{k=\mathrm{W}}^{n} [(k - \mathrm{W} + 1) + E(l_{i+1}+)] Pr[(l_{i+1} - \mathrm{P}_i) == k]$$

Similarly for the case where $k < \mathrm{W}$,

$$E(n) = \sum_{k=1}^{\mathrm{W}-1} [(\mathrm{W} - k + 1) + E(n - \mathrm{W})] Pr[(l_{i+1} - l_i) == k]$$

Solving the probability term and expectation term separately (illustrated in details in Appendix C), and adding up the expressions we obtain the general equation for the expectation of the number of bits predicted as $E(\mathrm{n})$:

$$E(n) = \left(\frac{7}{3} - \frac{1}{2^{\mathrm{W}}}\right) + \sum_{i=1}^{n-\mathrm{W}} \frac{1}{2^i} E(n - i - (\mathrm{W} - 1))$$
$$= \left(\frac{7}{3} - \frac{1}{2^{\mathrm{W}}}\right) \frac{n}{\mathrm{W} + 1}$$

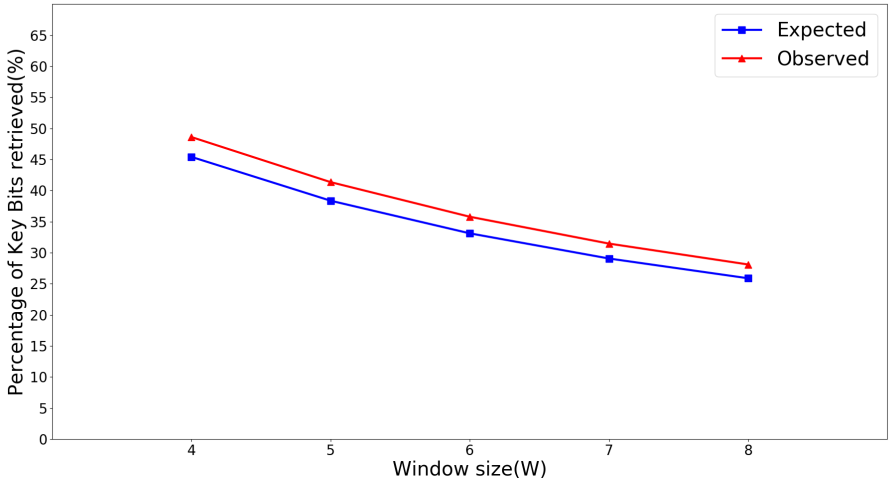


Fig. 7: Expected Efficiency versus Observed Efficiency

The entire expectation calculation has been validated over a number of random secret exponents and Figure 7 illustrates the correctness of the equation since it exactly matches with our simulation results from an automation as shown in the figure.

For RSA-2048 with a 7-bit window, we are expected to predict 29% of the exponent bits which is nearly twice the lower bound of our analysis. Note that the proportion of bits that can be retrieved is inversely proportional to window size. Decreasing the size of our exponent would also reduce the window size thereby increasing the expected proportion of bits our analysis could predict. For instance, RSA-512 uses a 5-bit window and our expected efficiency increases to 39%.

5 PHASE 3: Recovering the remaining secret key

In this section, we put together all the discussions from the earlier sections to frame a full recovery attack for the windowed algorithm of the BouncyCastle implementation of RSA. The *BigInteger* implementation uses Montgomery Algorithm to perform fast modular multiplication and squaring, and it consists of

Phone	Squaring	Mult.()	subN()
Motorola MotoG2	~1000000ns	~1440000ns	~10000ns
Motorola MotoG5	~200000ns	~260000ns	~2600ns
Huawei Honor 8Pro	~40000ns	~50000ns	~500ns

Table 1: Platform specific function execution timings

two steps: a multiplication and a reduction. We observed that ciphertexts decrypted with the same secret key had similar range of decryption timings due to the similar pattern of squarings and multiplication, but there is a small difference in their timing which arise from the varying execution time of Reduction function, `montReduce()`.

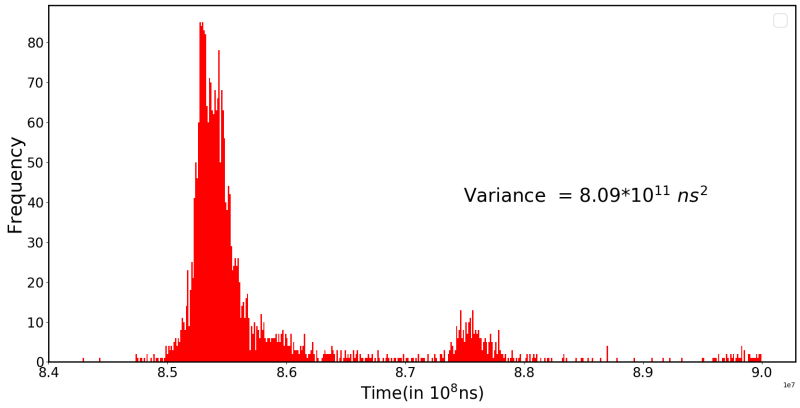
The subroutine `subN()` is executed whenever the conditional statement in `montReduce()` Line 16 of Listing 1.2 is satisfied which contributes to the timing difference. `subN()` is the conditional subtraction module of the `montReduce()` function which performs the reduction. In the code snippet, `n` is the value to be reduced and `mod` is the modulus, both represented as arrays of integers in BigEndian form.

Listing 1.1: Reduction Function

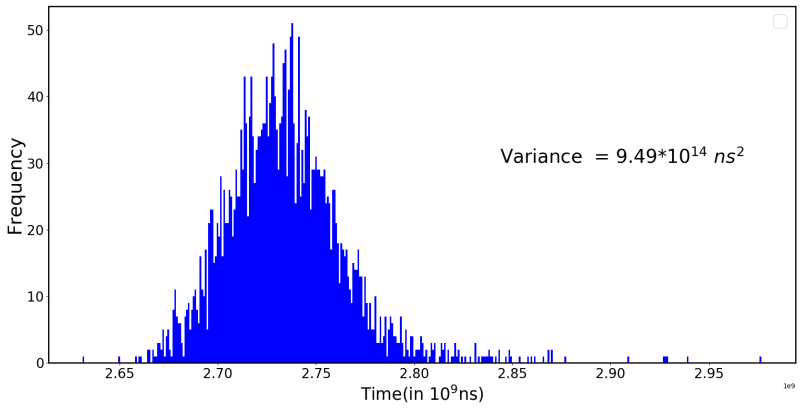
```
private static int[] montReduce(int[] n,
int[] mod, int mlen, int inv) {
  int c=0;
  int len = mlen;
  int offset=0;
  do {
    int nEnd = n[n.length-1-offset];
    int carry = mulAdd(n, mod, offset,
    mlen, inv * nEnd);
    c += addOne(n, offset, mlen, carry);
    offset++;
  } while(--len > 0);
  int j = 0;
  while(c>0){
        c += subN(n, mod, mlen);
  }
  while (intArrayCmpToLen(n, mod, mlen) >= 0)
  {
        subN(n, mod, mlen);
  }
  return n;
}
```

Multiplication takes almost fixed time given any two inputs vectors whereas the execution time for *Reduction* varies depending on the modulus and the input vector, practical measurements over separate plarforms are tabulated in Table 1. Table 1 refers to the range of the timing observations over squaring, multiplication and `subN()` function for each of these phones. Since the number of times reduction happens is dependent on both the modulus and the intermediate result from multiplication, we show next that this could be exploited.

Figure 8 shows that the newer phone as in Figure 8a are computationally faster and have lesser variance in timing observations than the older phone in



(a) Distribution of decryption time on Honor 8 Pro



(b) Distribution of decryption time on Moto G2

Fig. 8: Increase in accuracy on using a newer phone for timing measurements

Figure 8b, which inherently implies that the effect of noise is lower in newer phones than the older one.

In this paper, we focus on building up an attack algorithm using this non-constant times of reduction. The analysis proceeds in two steps: an `Offline` phase which requires sub-simulation over partially known key bits and then followed by `Online` timing measurement over the pre-computed samples separated by the `Offline` phase.

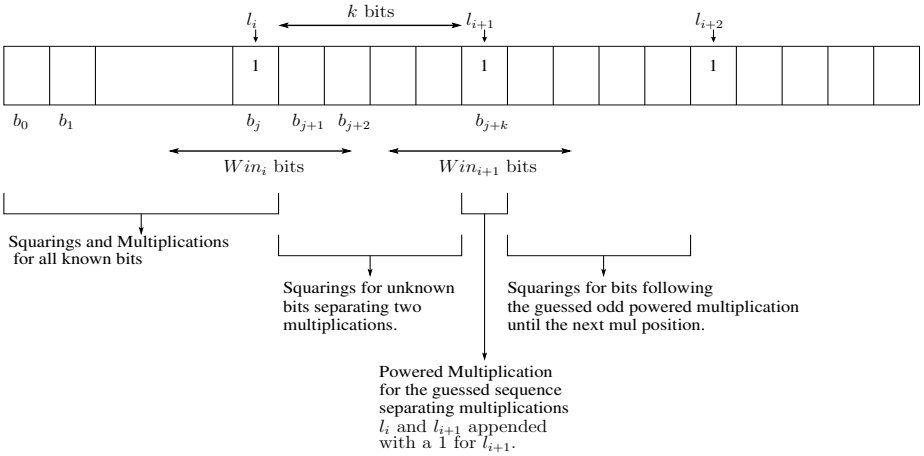


Fig. 9: Simulating reduction on guessed sequence of bits

5.1 Enforcing Timing Difference in `Reduction` Function

Thus starting from the fact that the positions of multiplication operation has been identified, we target to identify the number of unknown k bits between two identified positions of multiplications l_i and l_{i+1}. These k bits are denoted as $b_j, b_{j+1}, b_{j+2}, \cdots, b_{j+k-1}, b_{j+k}$ and the number of such bits ie, k may vary depending on how far the consecutive multiplications are separated. Continuing our discussion from previous section, depending on the number of bits k, there can be two cases when being compared with the window size W.

- If $l_{i+1} - l_i \geq \mathrm{W}$, the number of bits in between two consecutive multiplications l_i and l_{i+1} can be at most $\mathrm{W} - 1$.
- On the other hand, if $l_{i+1} - l_i < \mathrm{W}$, then the number of unknown bits in this case is $k - 1$ which is lesser than $\mathrm{W} - 1$.

Thus, the number of unknown bits in between two consecutive multiplications can either be $k-1$ or at most the size of the window $\mathrm{W}-1$, which we denote as R henceforth. The sequence of unknown bits can assume any of binary equivalent values (of R bits) from the set $\{0, 1, \cdots, 2^R - 1\}$. We use each of these 2^{R} sequence of binary bits and denote them as $\{Seq_0, Seq_1, \cdots, Seq_{2^R-1}\}$, where each of these are considered as guesses for the `Offline` subsimulations.

5.2 Offline Phase

The Offline phase of our analysis assumes each of the 2^{R} sequence of binary bits as one of its guess and separates a set of input ciphertexts based on the number of extra reductions. The separation of the input ciphertexts are conditioned over the guessed sequence of binary bits. So $\forall Guess \in \{Seq_0, Seq_1, \cdots, Seq_{2^R-1}\}$ as illustrated in Figure 9, the adversary simulates the following:

1. the squarings and multiplications for all the known bits from the MSB to the l_i^{th} multiplication (which is assumed to be already known in an iterative model),
2. the squarings for the sequence of all bits in between the l_i^{th} and l_{i+1}^{th} multiplication (is constant as there should be k squarings),
3. the multiplication for the guessed odd power for l_{i+1} (this is where the guess is incorporated), and
4. the squaring for the bits following l_{i+1} till the next multiplication at l_{i+2} (the reductions in this part which are actually getting affected from the guessed multiplication power).

The Montgomery reductions involved in the 4^{th} component ie, the squarings for number of bits separating l_{i+1} and l_{i+2} are the most crucial events which decides in the separating procedure. This is because the guessed powered multiplication in the 3^{rd} component takes place with the help of the precomputed table of odd powers. Thus the squaring operation immediately following the multiplication directly operates on the intermediate result produced by the multiplication with the guessed power. The guessed odd power of multiplication varies with the guessed sequence and so does the intermediate input to the squarings following l_{i+1}^{th} multiplication.

Separating inputs in bins The adversary takes a large set of ciphertexts as input, denoted as say M. For each input $\forall m_i \in$ M, the adversary simulates the windowed exponentiation algorithm for all the partial known bits and the assumed bits till the l_{i+2}^{th} multiplication and in turn calculates the number of Montgomery reduction operations have been encountered. Based on the number of reductions, the adversary selects the inputs m_i corresponding to maximum number of reductions d_i in the `Max_Bin` and alternatively puts input with minimum number of reductions in `Min_Bin`. As illustrated in Figure 10, this effectively generates two separate disjoint bins as `Max_Bin` and `Min_Bin` constituting of inputs having higher and lower number of reduction operations respectively. Note that, for each guess there would be always some inputs which would not qualify in any of the two bins. Those timing measurements are considered not to be useful in deciding the correct guess for that particular window, but may prove to be highly separating for some other window along the exponent. This process is repeated for all assumed guesses.

5.3 Online Phase

In this validation phase with actual timing measurements from Android device, we expect the correct guess to win from the 2^R possible guessed binary sequences. Offline phase generates two bins Max_Bin^j and Min_Bin^j constituting of input m_i's which are having maximum and minimum number of simulated reductions.

The odd powered multiplication indexed at l_{i+1}^{th} location can be identified using the following steps:

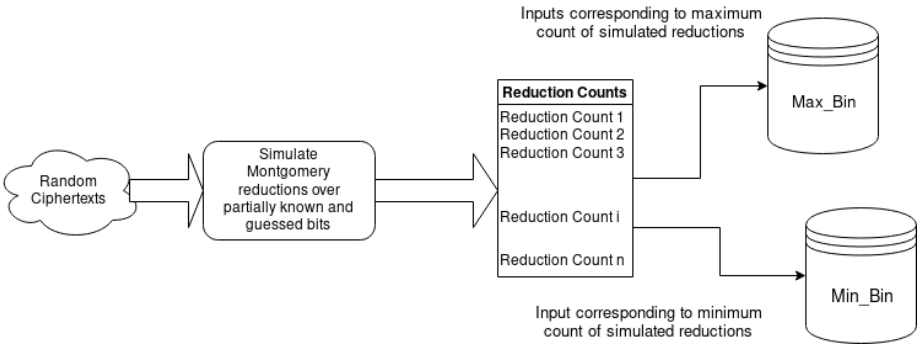


Fig. 10: Separating inputs based on guess and number of Simulated Reduction Counts

- $\forall j \in 2^R$, online phase prepares two more sets $Time\_MaxBin^j$ and $Time\_MinBin^j$ which include timing values for the respective input ciphertexts from two bins Max_Bin^j and Min_Bin^j.
  $Time\_High^j = \{t_i : \text{total decryption time for } \forall m_i \in High\_Bin^j\}$
  $Time\_Low^j = \{t_i : \text{total decryption time for } \forall m_i \in Low\_Bin^j\}$.
- We now construct $Separation^j$ based on difference of mean of the two distributions $Time\_MaxBin^j$ and $Time\_MinBin^j$.

Thus, at the end of the Online Phase, we select the correct guess as $Correct\_guess = j | Separation^j = max(Separation^0, \cdots, Separation^{2^R})$ using the Difference of Mean (DoM) Analysis.

6 Experimental details from the measurement setup

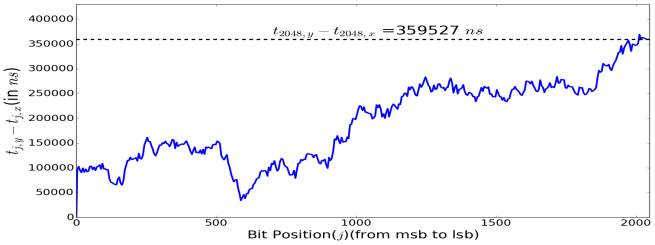
6.1 Timing the conditional reductions

In this subsection, we start with the actual reason of the differences in timing. According to the window algorithm, the major contributing factors for decryption time are the multiplications and squarings, and the number of times these squarings and multiplications that are executed is entirely dependent on the secret key. The implementation of Montgomery reduction following every squaring and multiplication is not a constant time function as discussed earlier. We observed that ciphertexts decrypted with the same secret key had similar decryption timings due to the similar pattern of squarings and multiplication, but there is a small difference in their timing which arise from the varying execution time of Reduction function, `montReduce()`.

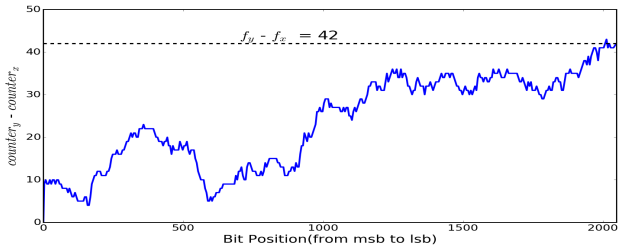
The subroutine `subN()` is executed whenever the conditional statement in `montReduce()` Line 16 of Listing 1.2 in Appendix D is satisfied which contributes to the timing difference. Based on our experiments on `RSA-2048` following the `FIPS 186-4` standard, we illustrate the differences in decryption time for a ciphertext compared to another to be based on the total number of reductions encountered, which we denote as f_i for ciphertext c_i. The symbols that we use in the subsequent discussion are detailed in Table 2.

Symbol	Representation
f_x, f_y	Number of *subN()* subroutine executions while decrypting ciphertext c_x, c_y
$w_{k,x}, w_{k,y}$	Number of *subN()* subroutine executions of the k bits between l_i^{th} and l_{i+1}^{th} multiplications for ciphertext c_x, c_y,
$t_{j,x}, t_{j,y}$	Time observed for execution till the j^{th} bit from the MSB for the secret exponent, for ciphertexts c_x, c_y.

Table 2: Tabular representation of symbols



(a) Difference in timing observation for two different ciphertexts



(b) Difference in counts of the number of Montgomery reductions for two different ciphertexts

Fig. 11: Proportionality of observed timing and the number of reductions

We chose to decrypt two example ciphertexts, say c_x and c_y on the same 2048-bit secret key with *subN()* frequencies $f_x = 425$ and $f_y = 467$(wlog), and also monitor their intermediate timings. We denote $t_{j,i}$ as the time observed for execution till the j^{th} bit from the MSB for the secret exponent, for a particular ciphertext c_i. In addition, we maintain two counters $counter_x$ and $counter_y$ for the two ciphertexts. These counters are incremented whenever *subN()* is executed while decrypting the ciphertexts along the secret exponent bits. In order to show that the timing observation for a ciphertext is directly proportional to the total number of reductions encountered, we show that the difference in intermediate timings of the ciphertexts c_x and c_y is related to the difference of the counters $counter_x$ and $counter_y$. Figure 11 shows that the variation of decryption time and counter values along the bit positions. The variation of difference of intermediate timings as illustrated in Figure 11a, has an exact same nature as

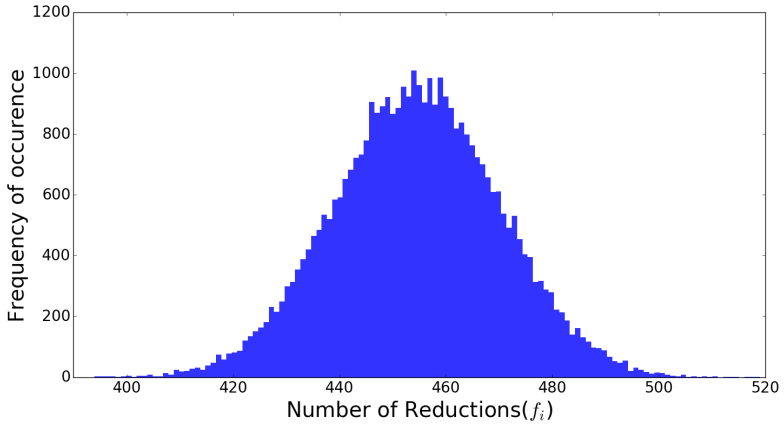


Fig. 12: Frequency Distribution for f_i

the difference in the number of reduction operations plotted in Figure 11b. The difference in average decryption time between c_x and c_y is 359527 ns, dividing this by $\|f_y$ - $f_x\| = 42$ gives 8560 ns for every execution of *subN()* which lies in the range of the expected value for MotoG2 as mentioned in Table 1.

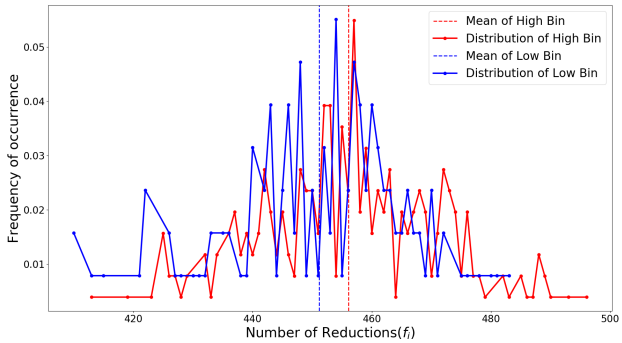
6.2 Distribution of f_i is gaussian in nature for a 2048 bit RSA

In this subsection we choose the standard RSA-2048 and select a modulus and key pair following `FIPS 186-4` standard. A random set of ciphertexts were decrypted using this selected secret key and observed the number of Montgomery reductions f_i for each of the ciphertexts c_i. It has been observed that the distribution of the total number of Montgomery reductions encountered for a particular secret key lies in the range from 400 to 520. Figure 12 shows the frequency distribution for a collection of 36000 ciphertexts. We repeated the decryption process with different random collections of ciphertexts, the frequencies of all the collections were normally distributed around similar mean values. This result is immensely important while we understand the efficiency of partitioning the bin.

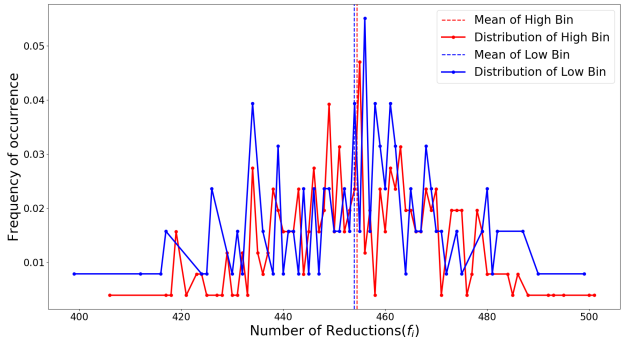
6.3 Efficiency of bin partitioning approach

Let $Guess^j$ denote the unknown bits of Win_i^{th} window and $Guess^j == Correct\_Guess$, then $\frac{\sum_{i \in High\_Bin_{Guess^j}} f_i}{|High\_Bin_{Guess^j}|} > \frac{\sum_{i \in Low\_Bin_{Guess^j}} f_i}{|Low\_Bin_{Guess^j}|}$.

Let $w_{k,i}$ denotes the number of *subN()* subroutine executions of the k bits between l_i^{th} and l_{i+1}^{th} multiplications for ciphertext c_i, then $\frac{\sum_{i \in High\_Bin_{Guess^j}} w_{k,i}}{|High\_Bin_{Guess^j}|} >$



(a) Correct Guess



(b) Wrong Guess

Fig. 13: Difference of Mean(DoM)

$\frac{\sum_{i \in Low\_Bin_{Guess^j}} w_{k,i}}{|Low\_Bin_{Guess^j}|}$.

Now, if we consider the distribution of f_i - $w_{k,i}$, it is independent of the bits in between l_i^{th} and l_{i+1}^{th} multiplications. This follows from the argument in the previous subsection and thus it would be Gaussian as well and there wouldn't be any correlation between the bins i.e, $\frac{\sum_{i \in High\_Bin_{Guess^j}} (f_i - w_{k,i})}{|High\_Bin_{Guess^j}|} \approx \frac{\sum_{i \in Low\_Bin_{Guess^j}} (f_i - w_{k,i})}{|Low\_Bin_{Guess^j}|}$.

Hence the average of the overall frequency distribution, f_i, would be Gaussian but with the mean $High\_Bin_{Guess^j}$ greater than $Low\_Bin_{Guess^j}$ only when $Guess^j == Correct\_Guess$.

Alternatively, for all other $Guess^j \neq Correct\_Guess$, the overall frequency distribution, f_i would be a Gaussian with the mean $High\_Bin_{Guess^j}$ same as the $Low\_Bin_{Guess^j}$, this is because $\frac{\sum_{i \in High\_Bin_{Guess^j}} w_{k,i}}{|High\_Bin_{Guess^j}|} \approx \frac{\sum_{i \in Low\_Bin_{Guess^j}} w_{k,i}}{|Low\_Bin_{Guess^j}|}$ since $Guess^j \neq Correct\_Guess$.

j	$\frac{\sum_{i \in Max\_Bin_{Guess^j}} f_i}{\lvert Max\_Bin_{Guess^j} \rvert}$	$\frac{\sum_{i \in Min\_Bin_{Guess^j}} f_i}{\lvert Min\_Bin_{Guess^j} \rvert}$	Difference
115	456.32	452.87	3.45
113	454.82	452.08	2.74
109	454.92	453.15	1.78
55	455.90	454.17	1.73
105	455.33	453.78	1.55
85	455.30	453.94	1.36
43	453.63	452.56	1.07
35	454.16	453.19	0.97
65	455.40	454.60	0.80
13	455.53	454.74	0.79

Table 3: Difference Table

Figure 13 illustrates the two cases, where 13a shows the separation of mean for the distributions from two different bins for the correct guess, while in 13b the separation is not visible as the means are overlapping for the wrong guess.

So in general for a pair of Correct and Wrong guess, $(\frac{\sum_{i \in High\_Bin_{Cor.\_Gues}} f_i}{\lvert High\_Bin_{Cor.\_Gues} \rvert} - \frac{\sum_{i \in Low\_Bin_{Cor.\_Guess}} f_i}{\lvert Low\_Bin_{Cor.\_Guess} \rvert}) >$ $(\frac{\sum_{i \in High\_Bin_{Wrong\_Guess}} f_i}{\lvert High\_Bin_{Wrong\_Guess} \rvert} - \frac{\sum_{i \in Low\_Bin_{Wrong\_Guess}} f_i}{\lvert Low\_Bin_{Wrong\_Guess} \rvert})$ The wrong guess does not partition the ciphertexts into meaningful subsets, instead it could be more or less a random partitioning with respect to $w_{k,i}$. Assuming that collection of ciphertexts to be large, the relationship would rather be : $\frac{\sum_{i \in High\_Bin_{Wrong\_Guess}} w_{k,i}}{\lvert High\_Bin_{Wrong\_Guess} \rvert} \approx \frac{\sum_{i \in Low\_Bin_{Wrong\_Guess}} w_{k,i}}{\lvert Low\_Bin_{Wrong\_Guess} \rvert}$.

We experimentally verified this observation by using a 2048-bit FIPS specified secret key performing modular exponentiation with a window size of 7. As the last bit of the sequence is a 1, we could have at most $64(2^6)$ possible combinations for the unknown bits. For our experiment, the secret 7 bits were of form 1110011(115) and the difference of the mean separated as $(\frac{\sum_{i \in High\_Bin_{Guess^j}} f_i}{\lvert High\_Bin_{Guess^j} \rvert} - \frac{\sum_{i \in Low\_Bin_{Guess^j}} f_i}{\lvert Low\_Bin_{Guess^j} \rvert})$ were calculated $\forall Guess^j \in \{1, 3, \cdots, 127\}$. Table. 3 displays the highest differences in descending order and we can verify that $Guess^j$ with value 115 has the highest difference as expected. This validates the Difference of Mean strategy in presence of multiple bins. The overall full-key recovery analysis can thus be iteratively performed over each sequence of unknown bits between two known multiplication positions.

7 Conclusion

The paper demonstrates a timing side-channel analysis on a number of available mobile platforms starting from the newest version of Android commercially available to the older ones. This timing exploit neither requires the architectural details of the underlying processor nor privileges to restricted sensor data or extra equipment for measurement, which also makes this analysis platform independent. The secret gets retrieved in 3 phases in this iterative recovery algorithm, but interestingly the adversary requires only a single set of timing observations in order to perform all the phases and thus faster the processor, more vulnerable is it to this genre of timing vulnerability.

References

1. Alam, M., Khan, H.A., Dey, M., Sinha, N., Callan, R.L., Zajic, A.G., Prvulovic, M.: One&done: A single-decryption em-based attack on openssl's constant-time blinded RSA. In: USENIX Security Symposium. pp. 585–602. USENIX Association (2018)
2. Belgarric, P., Fouque, P., Macario-Rat, G., Tibouchi, M.: Side-channel analysis of weierstrass and koblitz curve ECDSA on android smartphones. In: CT-RSA. Lecture Notes in Computer Science, vol. 9610, pp. 236–252. Springer (2016)
3. Berend, D., Jungk, B., Bhasin, S.: Guessing your PIN right: Unlocking smartphones with publicly available sensor data. In: ISVLSI. pp. 363–368. IEEE Computer Society (2018)
4. Bernstein, D.J., Breitner, J., Genkin, D., Bruinderink, L.G., Heninger, N., Lange, T., van Vredendaal, C., Yarom, Y.: Sliding right into disaster: Left-to-right sliding windows leak. In: CHES. Lecture Notes in Computer Science, vol. 10529, pp. 555–576. Springer (2017)
5. Genkin, D., Pachmanov, L., Pipman, I., Tromer, E., Yarom, Y.: ECDSA key extraction from mobile devices via nonintrusive physical side channels. In: ACM Conference on Computer and Communications Security. pp. 1626–1638. ACM (2016)
6. Kocher, P.C.: Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: CRYPTO. Lecture Notes in Computer Science, vol. 1109, pp. 104–113. Springer (1996)
7. Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., Mangard, S.: Armageddon: Cache attacks on mobile devices. In: USENIX Security Symposium. pp. 549–564. USENIX Association (2016)
8. Michalevsky, Y., Boneh, D., Nakibly, G.: Gyrophone: Recognizing speech from gyroscope signals. In: USENIX Security Symposium. pp. 1053–1067. USENIX Association (2014)
9. Michalevsky, Y., Schulman, A., Veerapandian, G.A., Boneh, D., Nakibly, G.: Powerspy: Location tracking using mobile device power analysis. In: USENIX Security Symposium. USENIX Association (2015)
10. Nakano, Y., Souissi, Y., Nguyen, R., Sauvage, L., Danger, J., Guilley, S., Kiyomoto, S., Miyake, Y.: A pre-processing composition for secret key recovery on android smartphone. In: WISTP. Lecture Notes in Computer Science, vol. 8501, pp. 76–91. Springer (2014)
11. Schwarz, M., Lipp, M., Gruss, D., Weiser, S., Maurice, C., Spreitzer, R., Mangard, S.: Keydrown: Eliminating software-based keystroke timing side-channel attacks. In: NDSS. The Internet Society (2018)

12. Spreitzer, R., Moonsamy, V., Korak, T., Mangard, S.: Systematic classification of side-channel attacks: A case study for mobile devices. IEEE Communications Surveys and Tutorials **20**(1) (2018)
13. Spreitzer, R., Plos, T.: On the applicability of time-driven cache attacks on mobile devices. In: NSS. Lecture Notes in Computer Science, vol. 7873, pp. 656–662. Springer (2013)

A Identifying Multiplication positions

The execution time for the algorithm varies depending on the number of bits of secret x (*i.e.* w) and its value. Let us denote the total execution time as $T = e + \sum_{i=0}^{w-1} t_i$, where t_i is the time required for the squaring and multiplication for i-th iteration of the loop *i.e.* which corresponds to the bit $x[i]$ and e includes the measurement error, loop overhead, and many other sources of noise. For a fixed x, time distributions are obtained by observing the execution time for different values of input y. Thus the variance of the distribution is: $Var(T) = Var(e) + w \cdot Var(t)$.

This iterative measurement leaks x being a squaring or multiplication starting from its most significant bit. $x[b]$ (where $0 \leq b < w$) can only be retrieved if all operation for bits from $x[w-1]$ to $x[b+1]$ are known. The attacker makes both guesses of $x^*[b]$ (just a squaring or a squaring and a multiplication) and obtains the execution time (T^*) for each of the two guesses. The detection the correct bit is decided by the difference between the time for the two executions is

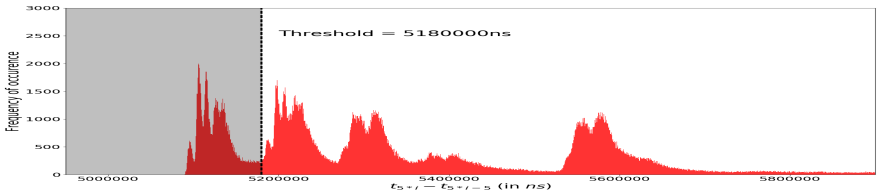
$$T - T^* = [e + \sum_{i=0}^{w-1} t_i] - [t_b^* + \sum_{i=b+1}^{w-1} t_i] = [e + \sum_{i=0}^{b-1} t_i] + t_b - t_b^*$$

Thus, if the guess is correct, the difference in the variance obtained is given by $Var[T - T^*] = Var(e) + b \cdot Var(t)$ On the other hand, on a wrong guess, $Var[T - T^*] = Var(e) + (b+2) \cdot Var(t)$.

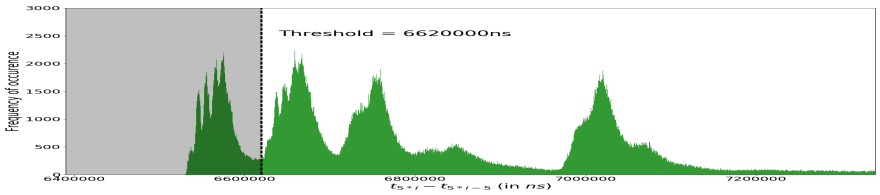
Therefore a correct guess causes reduction in variance, while a wrong guess affects the variance to increase. This distinguisher can be used to validate the guess for $x[b]$.

B Understanding the System Noise

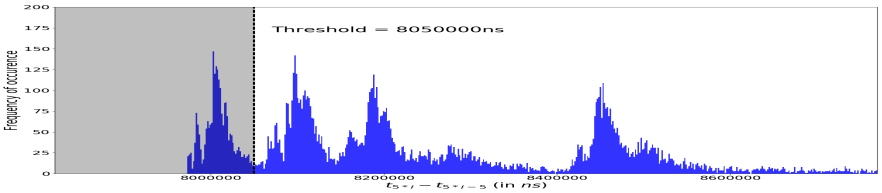
The effect of system noise in this set of experiments is immense. Filtering the system noise from the useful timing samples which carry information, plays a very important part in determining the success of the attack. Thus, to get a feel on how varied the timing samples can be, we record intermediate timings periodically after every few loop iterations. For this part of the experiment, without loss of generality we monitor the intermediate time after every 5 iterations of the loop. We record the timings at an interval of every 5 bits, starting from t_0, t_5 all the way to t_{2045} for a 2048-bit secret key. Here the subscripts denote the bit positions and modular exponentiation for decryption starts from the most significant bit. Therefore $t_0 < t_5 < \cdots < t_{2045}$. The difference $(t_{5*i} - t_{5*i-5})$



(a) Category 1



(b) Category 2



(c) Category 3

Fig. 14: Timing Distribution for three categories

Cat.	Operations	Mean(ns)	Threshold(ns)
1	No Mult. + Five Squarings	5374837	5180000
2	1 Mult. + Five Squarings	6828097	6620000
3	2 Mult. + Five Squarings	8280845	8050000

Table 4: Categories for 5 consecutive iterations

could belong to one of three categories, as described in Table. 4, based on the number of multiplications that take place in the five loop iterations. There can be atmost 2 multiplications happening from two adjacent windows and also there can be no multiplications. The adaptive window size being selected for a 2048 bit secret exponent of RSA is 7. The window though being of size 7 bits, the lookahead buffer decides when to start the window based on the bits encountered, thus this timing analysis with timing observation after every 5 bits is no way aligned to the actual windows being formed while the algorithm iteratively performs modular exponentiation.

The timing for squarings are same for all the three cases, but the multiplication operation do contribute for significant timing difference for the later two categories. Figure 14 shows the distribution of timing measurements observed for each of the categories, where each of them are multi-modal distributions and

the Table 4 shows the respective mean and threshold selection if we only filter the first gaussian curve under the shaded region to be useful, and subsequently discard the timing samples which belong to the later parts of the distribution. Discarding timing samples which belong to the other Gaussians in such a multimodal distribution are not recommended as compulsory for replicating our experiments. Though it has been observed that the most noisefree signal belongs only to the Gaussian under the shaded region. Consequently, the success of experiments can be converged with involvement of lesser samples as input if we consider the timing observations under the shaded region and thus the selection of the threshold plays an important role in filtering out useful signals.

C Efficiency of partial key recovery analysis

The expectation of number of bits that can be revealed from the position following the Placeholder bit P_i is represented as $E(n)$ ie, n bits from P_i to LSB. The quantity is actually dependent on the sum of Probabilities of the next multiplication (at l_{i+1}) occurring in next k bits from the last multiplication (at l_i). Now based on the window size W the expectation can be expressed in two different forms. In the first case we consider the situation where $k \geq \mathrm{W}$ and it ends up in a sum of two equation, where one part reveals some bits in between the l_i and l_{i+1}^{th} multiplication and the other part is a recursive call to the bits following l_{i+1}^{th} multiplication.

$$E(n) = \sum_{k=\mathrm{W}}^{n} [(k - \mathrm{W} + 1) + E(l_{i+1}+)] * \\ Pr[(l_{i+1} - \mathrm{P}_i) == k]$$

When $k == \mathrm{W}$, the probability term can be written as,

$$\begin{aligned} k = \mathrm{W} : Pr[(l_{i+1} - \mathrm{P}_i) == \mathrm{W}] \\ = \frac{1}{2^2} + \frac{1}{2^4} + \frac{1}{2^6} + \frac{1}{2^8} + \frac{1}{2^{10}} + \frac{1}{2^{12}} + \frac{1}{2^{13}} \\ = \frac{1}{3} \end{aligned}$$

And, when $k > \mathrm{W}$, the probability term can be written as,

$$\begin{aligned} k = \mathrm{W} + t : Pr[(l_{i+1} - \mathrm{P}_i) == \mathrm{W} + t] \\ = \frac{1}{2^{t+2}} + \frac{1}{2^{t+4}} + \frac{1}{2^{t+6}} + \frac{1}{2^{t+8}} + \frac{1}{2^{t+10}} \\ + \frac{1}{2^{t+12}} + \frac{1}{2^{t+13}} \\ = \frac{1}{2^t * 3} \end{aligned}$$

Now, the expectation is calculated as follows:

$$\sum_{k=n}^{n} (k - \mathrm{W} + 1) * Pr[(l_{i+1} - l_i) == k] = \\ \frac{1}{3} + \frac{2}{2 * 3} + \frac{3}{4 * 3} + \cdots + \frac{n-1}{2^n * 3} \\ = \frac{4}{3} \qquad \cdots (1)$$

Similarly for the case where $k < \mathrm{W}$,

$$E(n) = \sum_{k=1}^{\mathrm{W}-1} [(\mathrm{W} - k + 1) + E(n - \mathrm{W})] * Pr[(l_{i+1} - l_i) == k]$$

$$k = 1 : Pr[(l_{i+1} - \mathrm{P}_i) == 1] = \frac{2^{k-1}}{2^{\mathrm{W}}}$$

$$k = 2 : Pr[(l_{i+1} - \mathrm{P}_i) == 2] = \frac{2^{k-2}}{2^{\mathrm{W}}} + \frac{2^{k-2}}{2^{\mathrm{W}+1}}$$

$$k = 3 : Pr[(l_{i+1} - \mathrm{P}_i) == 3] = \frac{2^{k-2}}{2^{\mathrm{W}}} + \frac{2^{k-3}}{2^{\mathrm{W}+1}} + \frac{2^{k-3}}{2^{\mathrm{W}+2}}$$

$$k = 4 : Pr[(l_{i+1} - \mathrm{P}_i) == 4] = \frac{2^{k-2}}{2^{\mathrm{W}}} + \frac{2^{k-3}}{2^{\mathrm{W}+1}} + \frac{2^{k-4}}{2^{\mathrm{W}+2}} + \frac{2^{k-4}}{2^{\mathrm{W}+3}}$$

$$k = \mathrm{W} - 1 : Pr[(l_{i+1} - \mathrm{P}_i) == \mathrm{W} - 1] = \frac{2^{k-2}}{2^{\mathrm{W}}} + \frac{2^{k-3}}{2^{\mathrm{W}+1}} + \cdots + \frac{2^{k-(\mathrm{W}-1)}}{2^{2\mathrm{W}-2}} + \frac{2^{k-(\mathrm{W}-1)}}{2^{2\mathrm{W}-2}}$$

$$\sum_{k=1}^{\mathrm{W}-1} (\mathrm{W} - k + 1) Pr[(l_{i+1} - l_i) == k] = 1 - \frac{1}{2^{\mathrm{W}}} \qquad \cdots (2)$$

Solving the recursive part of the equation from $k \geq \mathrm{W}$ and $k < \mathrm{W}$, we get

$$\sum_{k=1}^{\mathrm{W}-1} [E(n - \mathrm{W})] * Pr[(l_{i+1} - l_i) == k] + \sum_{k=\mathrm{W}}^{n} E(n - k) * Pr[(l_{i+1} - l_i) == k]$$

$$= \frac{1}{2} E(n - \mathrm{W}) + \frac{1}{4} E(n - \mathrm{W} - 1) + \cdots + \frac{1}{2^{n-\mathrm{W}}} E(1)$$

$$= \sum_{i=1}^{n-\mathrm{W}} \frac{1}{2^i} E(n - i - (\mathrm{W} - 1)) \qquad \cdots (3)$$

Adding up Equations (1), (2) and (3) results in the general equation of expectation of the number of bits retrieved as:

$$E(n) = \left(\frac{7}{3} - \frac{1}{2^{\mathrm{W}}}\right) + \sum_{i=1}^{n-\mathrm{W}} \frac{1}{2^i} E(n - i - (\mathrm{W} - 1))$$

D Snippets of Montgomery Reduction

Listing 1.2 shows the `Reduction` function's definition.

Listing 1.2: Reduction Function

```
private static int[] montReduce(int[] n,
int[] mod, int mlen, int inv) {
  int c=0;
  int len = mlen;
  int offset=0;
  do {
    int nEnd = n[n.length-1-offset];
    int carry = mulAdd(n, mod, offset,
    mlen, inv * nEnd);
    c += addOne(n, offset, mlen, carry);
    offset++;
  } while(--len > 0);
  int j = 0;
  while(c>0){
        c += subN(n, mod, mlen);
  }
  while (intArrayCmpToLen(n, mod, mlen) >= 0)
  {
        subN(n, mod, mlen);
  }
  return n;
}
```

In the above code snippet `n` is the value to be reduced and `mod` is the modulus, both represented as arrays of integers in BigEndian form.

Listing 1.3: Array Comparison

```
final static long LONG_MASK = 0xffffffffL;

private static int intArrayCmpToLen(int[]
arg1, int[] arg2, int len) {
  for (int i=0; i<len; i++) {
    long b1 = arg1[i] & LONG_MASK;
    long b2 = arg2[i] & LONG_MASK;
    if (b1 < b2)
        return -1;
    if (b1 > b2)
        return 1;
  }
  return 0;
}
```