Using Timing Attacks Against Cryptographic Algorithms

Harry Budd

Supervisor: Martin Berger

August 31, 2017

Contents

Abstract

Computer algorithms that are written with the intent to keep data private are used in every day cryptography. These algorithms may exhibit execution time behaviour which is dependant on secret information that is not known to an outsider. When carefully analysed, this dependency may leak information that can be used to gain unintended access to private data, effectively nullifying the use of such algorithms. This threat poses a vital risk to the field of computer cryptography, and analysis should be done in attempt to eradicate this potential threat from any algorithms in modern day use.

In this paper, attacks are orchestrated against several algorithms that have previously been used in cryptography, resulting in the successful retrieval of secret data within a manageable time-scale.

1 Introduction

It is not uncommon for a computer algorithm to have a runtime which varies upon changing its input. For instance, conditional statements may give the processor various alternative paths to traverse, with some requiring more computation and subsequently more time to execute. Whilst in some cases this behaviour may be an intentional product of performance optimisation, timing variations to any degree are often unavoidable.

In the general scope of computer science there is no downside to having an algorithm run in non-fixed time, however, this behaviour poses a vital weakness in algorithms that are used in modern cryptography. Cryptographic algorithms are designed to encrypt and decrypt data, with the intent of keeping the data private and secure. If a cryptographic algorithm has a runtime which is dependant on its input, appropriate analysis of the resulting timing variations may give information pertaining to a secret, that is, a packet of information that is not supposed to be accessible. Secret information leaked in this way is referred to as a timing leak.

A program that has the sole intention of exploiting a timing leak in order to compromise a cryptographic system is called a timing attack, which is a specific type of side-channel attack. Side-channel attacks envelop a range of data-driven methods designed to attack cryptosystems by analysing a specific type of data that is obtainable via non-intrusive physical implementation of the cryptosystem. In regards to timing attacks, this specific data type is runtime data, which can be collected manually with the use of a timing library.

2 Developing an Understanding

The general methodology for executing a timing attack is best understood by first considering a straightforward attack. Algorithm 1 ensures that a user entered password matches a secret key. A potential real world use for this algorithm is for login authentication, where the user entered password is checked with the privately stored secret to ensure they match before allowing login access. In actuality this algorithm is too simplistic to be implemented in any modern day cryptosystem, but the substantial timing leaks it exhibits make it a good practical example for building an initial understanding of the general methodology, and framework, behind conducting a timing attack.

Algorithm 1 Algorithm for a password checker

```
function CHECK(password)
    if length(password) ≠ length(secret) then return False
    let i = 0
    while (i ≤ length(password)) and (password[i] ≡ secret[i]) do
        let i = i + 1
    return (i ≡ length(password))
```

Algorithm 1 has two substantial timing leaks: the **if** statement, and the **while** loop. If checking a password which has a different length to the secret, the function will return false at the **if** statement. Whereas, if the lengths were the same, the processor will continue on to the **while** loop. By the nature of serial computation, the processor will spend a longer period of time inside the function when the password and secret have equivalent lengths. A timing attack will take advantage of this by repeatedly measuring the execution time of this function for passwords of differing lengths. By analysing these times, an attacker will notice a significant spike in execution times for passwords that have the same length as the secret. Hence, by implementation of this attack, an attacker will obtain the length of the secret key without having any prior knowledge of this key. Using this information, a second attack is then tailored towards the **while** loop. The attacker collects a new set of runtime data by inputting passwords of the correct length, but this time varying the first character. A similar analysis as before will uncover the true first character. This new information is then used to find the second character via the same process, and the attack is continued in this way until the full secret is revealed.

3 Background

3.1 Cryptographic Techniques

The general methodology behind computer cryptography begins with an encryption stage, that is, an initial transformation of data from its primary state, called plaintext, into an alternative format, called ciphertext. Ciphertext is completely unreadable to an outsider, provided that the encryption method used is sufficiently secure. The only way of uncovering the plaintext is to then decrypt the ciphertext with the use of a unique secret key, generated by the encryption scheme being used.

There are two types of encryption schemes: symmetric key encryption and asymmetric key encryption. Symmetric key schemes use a single, shared key, which is used for both the encryption and decryption stages. Typically, the mathematics behind generating and using symmetric keys apply no physical constraint to the quantity of data that can be encrypted. This means that there is effectively no hard limit to the amount of data that can be encrypted this way, given that we provide the system with enough memory. Symmetric key encryption works well for a single user aiming to protect their data from other people, however, it has a significant weakness when it comes to secure data transfer due to the usage of a single key.

Secure data transfer is one of the primary uses for encryption. If Alice wishes to transfer a message to Bob using a symmetric key scheme such that Bob can read it, then Alice must find a way to exchange the secret key with Bob. The vulnerability then lies with the key exchange, because an outsider only needs to intercept the secret key during transmission, which they can then use to decrypt the ciphertext. This element of key transfer is thus the subject of research, as symmetric key encryption is only as safe as its key exchange method.

The alternative encryption scheme to symmetric, named asymmetric key encryption, combats the issue of requiring secure key transfer. The fundamental principle of this scheme lies in its asymmetric property, a pair of public and private keys as opposed to a single secret key. As the names suggest, the public key is publicly available, whereas the private key is accessible only by its owner. These keys have an underlying connection which allows them to work in pairs to encrypt and decrypt information, for example, if Alice wishes to send a secure message to Bob, then Alice must encrypt the message using Bob's readily available public key. The only way to decrypt the ciphertext is by using Bob's private key, which only Bob has access too. This allows secure transfer of data between parties. The underlying mathematical structure behind the public and private keys produce a drawback, however, and that is a physical limit to the amount of data that can be encrypted.

One widely used asymmetric encryption technique is RSA[1], first developed in 1977. The uniqueness to RSA is the method in which the public and private keys are generated. The key concept to asymmetric key generation is to find a one-way function capable of creating two keys which interact in the way that is required, in such a way that there is no computationally feasible procedure that can be used to reverse the process in order to find the private key from the public key. The creators of RSA used a one-way function that is based upon using very large prime numbers. They noted that by calculating the product of two primes, given that the primes are sufficiently large, it is not computationally feasible to find these prime factors from the product alone. The public and private keys are then generated under consideration of this in the following way:

1. Choose two large primes, p and q
2. Compute the product $n = p \cdot q$
3. Compute n's totient $\phi(n) = (p-1)(q-1)$, where ϕ is Euler's totient function
4. Choose integer e such that:
   - $1 < e < \phi(n)$
   - e and $\phi(n)$ are coprime
5. Find d which satisfies $d \cdot e \equiv 1 \mod \phi(n)$, typically done using the Extended Euclidean Algorithm

The public key exponent here is e, and d is the private key exponent. Encryption of a plaintext m to ciphertext c is done using the public exponent: $c \equiv m^e \mod n$, and decryption of this ciphertext is then done using the private exponent: $m \equiv c^d \mod n$. Due to the use of modular arithmetic in RSA, the size of the message is constrained to being less than the modulus, n. The key size, which refers to the length of n, is typically between 1024 and 4096 bits in modern day encryption. Usage of larger keys is possible, however, computation time becomes expensive and thus inefficient as the key size increases. This leaves us with, typically, being able to encrypt a maximum of 4096 bits of data, which is impractically small.

The data size constrictions of asymmetric encryption, and the weakness of insecure key transfer for symmetric encryption, can be negated by using a hybrid cryptosystem, that is, a cryptosystem which combines both asymmetric and symmetric key encryption methods. This is done by first using a symmetric key to encrypt some plaintext data. An asymmetric key technique is then used to transfer the symmetric key securely to a receiving party, and then consequently only the receiver can decrypt and read the ciphertext. Hybrid encryption clearly bypasses the disadvantages of using symmetric and asymmetric techniques separately, and thus has become common practise.

3.2 Kocher's Attack on RSA

Paul C. Kocher[2] made the first significant breakthrough in the topic of timing attacks in 1996, where he developed attacks capable of exposing secret keys used in RSA decryption, fixed Diffie-Hellman exponents used in Diffie-Hellman key-exchange(D-H), and further intrusive information used in other cryptosystems. In addition to posing a threat to these cryptosystems, his paper also provided plausible techniques for preventing timing attacks against RSA and D-H.

Operations used in both RSA and D-H are based upon the computation of $m = c^d \mod n$, where d is a private key of length w, n is public, c is the ciphertext accessible by an eavesdropper, and m is a plaintext message. Kocher discovered a timing leak within the square and multiply algorithm, which is used to compute $m = c^d \mod n$:

Algorithm 2 Square and Multiply algorithm for computing $m = c^d \mod n$

```
let R = c
for i = 1 upto w − 1 do
    let R = R^2 mod n
    if (bit i of d) is 1 then
        let R = R · c mod n
return R
```

The attacker needs to record the algorithms computation time for carefully selected varying c, in order to reveal d bit by bit until the entire exponent is obtained and the secret key is revealed. Under the assumption of perfectly accurate timing measurements, this attack would be straightforward and quick to implement; however, all timing measurements will contain various components of noise from different sources. The primary source of noise is due to the transferral of data over a network, where factors such as network latency and packet loss will form a non-constant noise component, which is often much more significant than the timing variation in the algorithm.

Kocher[2] proposed the idea of treating this as a signal detection problem, where the "signal" refers to the timing variation due to the exponent bit; that is, the particular time component of interest to the attacker. The "noise" component of this problem is a combination of measurement inaccuracies and the timing variations due to unknown bits. This attack is a statistical based approach which is outlined as follows:

First define $T = e + \sum_{i=1}^{w-1} t_i$ as the total time taken to decrypt a given c. Here, e is the component of noise due to measurement error, loop overhead, and the computations done outside of the loop. The time required for multiplication and squaring steps is then t_i for loop i, which is of focus to the attacker. Assuming the first $b-1$ bits of the secret key are uncovered, and a guess is being made at bit b, label this guess as d_b, composed of the first $b-1$ bits and an additional bit guess. The attacker can measure the time taken to decrypt the same message c with respect to the secret key guess d_b to be $\sum_{i=1}^{b-1} t_i$. If bit guess d_b is correct, these timing measurements should match the first $b-1$ loops of T, thus by subtracting from T we receive $e + \sum_{i=1}^{w-1} t_i - \sum_{i=1}^{b-1} t_i = e + \sum_{i=b}^{w-1} t_i$. Under the assumption that modular multiplication times are independently distributed, the variance of this can be calculated as $\mathrm{Var}(e) + (w-b)\mathrm{Var}(t)$.

In the case of an incorrect bit guess k_b, the last timing measurement in $\sum_{i=1}^{b-1} t_i$ will not match the corresponding measurement in T, thus the variance of $T - \sum_{i=1}^{b-1} t_i$ equates to $\mathrm{Var}(e) + (w + b - 2)\mathrm{Var}(t)$. This finding by Kocher allows the attacker to make two guesses at k_b, and calculate the variance of $T - \sum_{i=1}^{b-1} t_i$ over a number of samples of c for each guess, and should find that the lowest variance amongst these samples accords to the correct exponent bit guess. The attacker can then append this bit guess to their list of exponent bit guesses, and use this to attack the next bit in an analogous way.

Following on from this, Kocher derived a way to represent the number of samples required for a successful attack as a relationship between specific properties of the signal and noise. This results in a proportionality between the required number of samples and the length of the secret key:

$$\Pr(\text{Correct guess}) = \Phi\left(\sqrt{\frac{j}{2(w-b)}}\right)$$

where $\Phi(x)$ is the area of the standard normal curve from $-\infty$ to x, and j is the number of samples.

The experiment conducted in this paper was an application of the proposed timing attack to a slightly different modular exponentiation algorithm which was used in the RSAREF toolkit[3]. The algorithm in RSAREF calculates two bits at a time, in a way similar enough for the attack to be directly translatable with minor adjustments. This attack was applied in a closed system, thus negating network noise. This paper also leads on the assumption that the attacker knows the entire infrastructure of the cryptosystem itself, and so whilst the experimental results proved successful, the exact same procedure is not directly applicable to RSA cryptosystems in practical use. Due to the combination of network noise and lack of information about a cryptosystems infrastructure, applying time attacks to practical RSA implementations will prove to be a further complicated problem. The basis of this paper was to bring light to the fact that timing attacks are a viable threat to most secure brands of cryptosystems, and relevant time attack prevention techniques should be considered when building future cryptosystems.

One prevention technique touched on by Kocher is used widely today. This technique is an adaptation from signature blinding[4], which was introduced to the cryptographic field 15 years prior. The initial interpretation of signature blinding was to adapt a secure message transfer protocol in such a way that it allows one to get a message signed by another party, whilst simultaneously concealing the message from said party. Kocher's adaptation on this prevents attackers from knowing the input to the modular exponentiation function, however, whilst this prevents discovery of the entire secret key via timing attacks, information such as the secret key's hamming weight is still accessible from an attack.

3.3 Dhem et al.'s Attack on RSA

In 1998, J.-F.Dhem et al.[5] explored further developments to the ideas proposed by Kocher, in an attempt to provide a more practical implementation of a timing attack. As previously mentioned, Kocher's ideas were based on the assumption that an attacker has extensive knowledge of the infrastructure of the cryptosystem they are attacking, which for real world scenarios is often not the case. Under consideration of this, Dhem et al. developed an attack on smart cards, for which extensive knowledge is not necessarily needed. A smart card is a form of hardware security token, which serves as a method of two-factor authentication. The fact that this is a hardware device means that the attack does not need to be carried out over a network, negating the component of noise that is due to network latency/packet loss.

In this paper, timing attacks were carried out on a CASCADE smart card, which is encrypted with a form of RSA encryption. The RSA implementation on this card involves the computation of $m = c^d \mod n$, and it is known that, for this specific hardware, the computation is done using the previously mentioned square and multiply algorithm in conjunction with the Montgomery algorithm[6] which is used for all modular multiplication and squaring steps. The Montgomery algorithm is used for computing $a \cdot b \mod n$, and for some inputs this algorithm takes one extra reduction step, the timing variation due to which is measurable.

By taking this conditional extra step in the Montgomery algorithm into consideration, Dhem et al. managed to carry out a statistical based attack similar to Kocher's, whilst selecting specific inputs that would lead to a higher variance due to the extra step. This increase in variance gave more distinct thresholds under which to base the bit guess decision criteria under. However,

after experimental evaluation, it was found that the modular multiplications by constant c were correlated and thus not independent, leading to a high statistical bias amongst the decision criteria. In consideration of this bias, Dhem et al. proposed an alternate attack which focuses on the modular squaring step, as opposed to the modular multiplication step, which is outlined as follows:

Similarly to Kocher's attack, suppose that the first $b-1$ bits of the secret key are known, and we are to attack bit b. As the first $b-1$ bits are known, we can execute the first $b-1$ iterations of the square and multiply algorithm whilst keeping track of the value R. Denote R_{temp} as the value R holds before the conditional multiplication step due to bit b.

Let us first suppose that bit b is 0. In this case the next calculation will be

$$R = R_{temp}^2 \mod n$$

As we know this is calculated using the Montgomery algorithm, we separate a set of samples of c into two subsets, C_1 and C_2. One of these subsets will contain c's which require an extra reduction step at this stage, with the other subset containing those that do not require an extra step.

Next consider the other case: bit b is 1. The following calculations in this case are:

$$X = R_{temp} \cdot c \mod n$$

$$R = X^2 \mod n$$

Similarly to the first case, we can filter through our set of c, creating two more subsets C_3 and C_4. One of these subsets will contain all c which require an extra reduction at the $X^2 \mod n$ calculation, after the intermediate multiplication calculation, and the other subset will contain c's for which the extra reduction step does not take place.

We are now able to build decision criteria after calculating T_i, the average execution time for subset C_i. If bit b is 0, $\mid T_1 - T_2 \mid$ should give a difference approximately equal to the time taken for an extra step in the Montgomery algorithm. $\mid T_3 - T_4 \mid$ should negligible, however, as subsets C_3 and C_4 are separated according to the wrong criteria, that is, under the assumption that bit b is 1. The converse of this applies if bit b is 1, which gives us the following decision criteria:

$$\text{bit } b = \begin{cases} 0 & \text{if } \mid T_1 - T_2 \mid > \mid T_3 - T_4 \mid \\ 1 & \text{if } \mid T_1 - T_2 \mid < \mid T_3 - T_4 \mid \end{cases}$$

Using this criteria we are able to find bit b from bits $1...b-1$, and then repeat this same strategy analogously to recover the entire secret key. Note here that as this attack requires executing some calculation in the loop following the bit under evaluation, it is not possible to reveal the last bit, thus it must be guessed.

This attack also has an error-detection property. If an incorrect bit guess is made, the subsequent subset separations will be meaningless as R_{temp} will not correspond to its true value. Due to this, $\mid T_1 - T_2 \mid$ and $\mid T_3 - T_4 \mid$ will both be negligible, as will the difference D between them. This difference D is plotted in Figure 1 which illustrates this feature well; it is clear that an error has occurred near bit 10.

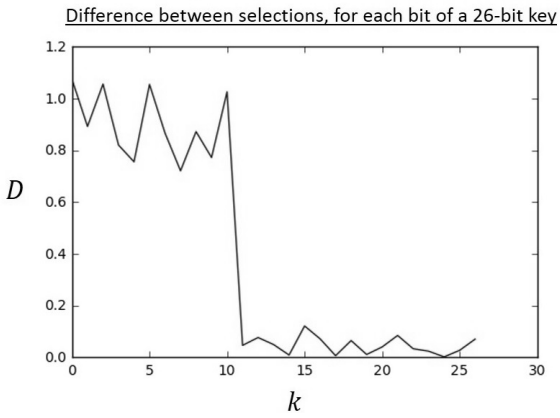


Figure 1: Illustration of error detection for a 26-bit key

Once an error is detected, correcting it is straightforward. We simply need to take one step back, flip our original guess for this bit, and continue evaluating the difference for the following bit guesses to see if the error has been corrected. If the error remains, take an additional step back and repeat this process until the error has been corrected.

Dhem et al. carried out an experiment on this attack by attacking an emulator for a CASCADE smart card, which resulted in the development of a timing attack capable of retrieving a 512-bit secret key at a rate of 1 bit per 37 seconds. This was one of the first two successful practical implementations of a timing attack, second to an attack presented by Lenoir during the rump session of CRYPTO'97.

In addition to the usage of blinding mentioned by Kocher, Dhem et al. proposed two additional countermeasures. The less efficient of the two was to add a randomised timing delay to an algorithm, in order to mask variations in time with an element of randomness. This is not an ideal countermeasure, however, due to its inefficiency. The alternative idea was to apply a specific modification to the Montgomery algorithm in such a way to remove all timing variation.

The blinding technique proposed in this paper is by application of a randomised function to the ciphertext before the modular exponentiation is calculated, then an inverse function is applied to the result before being returned as the plaintext. As this function is randomised, the attacker cannot mimic the transformation and thus it is not possible to simulate the internal calculations, leaving this attack irrelevant. Kocher[2] proposed a way to create such a function in his paper.

3.4 Remote Timing Attacks

Up until this point, it was general belief that timing attacks could not be carried out over web servers, as result of network noise significantly masking timing variations in the cryptographic algorithm. In addition to this, Kocher's and Dhem at al.'s attacks did not work against RSA cryptosystems which use Chinese Remainder Theorem (CRT), leading to the belief that CRT was not vulnerable to timing attacks. However, in 2000, Brumley and Boneh[7] published a timing attack capable of extracting private keys from an openSSL-based web server hosted on a local network, which uses CRT for its RSA decryption.

In the case of RSA decryption that uses CRT, $m = c^d \mod n$ is calculated in three parts. Firstly, $m_1 = c^{d_1} \mod p$ and $m_2 = c^{d_2} \mod q$ are computed, where $n = p \cdot q$, and d_1, d_2 are computed from d. Then, m_1 and m_2 are combined using CRT to get m. By performing a timing attack on this, an attacker can obtain p and q. It is then possible to find the secret key by computing $d = e^{-1} \mod (p-1)(q-1)$. This was the discovery that Brumley and Boneh made, that those previously failed to uncover.

Using this attack, Brumley and Boneh managed to extract secret keys over an openSSL server ran on their campus network, provided that the network has small latency variance, such as a LAN or campus network. Due to these findings, various network based cryptographic libraries now implement blinding techniques as means of protection against timing attacks, where the need for relevant prevention was barely considered beforehand. The addition of blinding comes at a cost, however, with a 2-10% performance cost according to an RSA press release in 1995.

4 Data Collection

It is apparent from previous literature that reliable timing attacks require the ability to take precise timing measurements. Therefore, it is vital to take this into consideration when choosing a programming language in which to develop my experiments.

4.1 C++

Both Kocher and Dhem et al. developed their timing attacks using C++. C++ is a lower level language, resulting in a smaller contribution of noise from the background processes than when compared to higher level languages, such as Python. In addition to this, there are a number of timing libraries available for C++ that allow the collection of low resolution timing measurements, which is vital for detecting small timing variations such as the reported $2\mu s$[1] timing variation in Dhem et al.'s attack.

[1] Variation of 422 CPU cycles measured with a 200MHz processor.

I intend to implement attacks on a 3.8GHz Intel i5 quad-core processor running Windows 7. After experimentation with various timing libraries in C++ using GCC compiler, many difficulties arose, such as clashes with my operating system, and timing resolutions not being as low as expected. This left only one timing method that I was able to work with, Microsoft's Read Time-Stamp Counter[8] (RDTSC).

The RDTSC instruction assembly provides a low-overhead method of getting CPU timing information. Upon CPU start up, a Time Stamp Counter (TSC) is initialised, which keeps a count of how many CPU cycles have occurred since initialisation. When making a call to RDTSC, the count is returned, providing a method of measuring precisely how many CPU cycles occur over given time period with the highest possible resolution.

Following further experimentation with RDTSC, a number of complications were encountered. There were cases where timing variations were effectively null (10 nanoseconds) for operations that were expected to take approximately 10000× longer. This was due to compiler optimisation, where the GCC compiler was ignoring the operation inside the timing measurement. Compilers optimise to preserve memory or minimize execution time, one way it does this is by ignoring operations which the compiler deems unnecessary. When timing a single operation a multiple number of times, the structure is typically as follows:

Algorithm 3 Structure for timing x^2

Let $a = \text{RDTSC}()$
Compute x^2
Let $b = \text{RDTSC}()$
Let Elapsed $= b - a$

As the resulting computation of the operation being timed, in this case x^2, is not being assigned to a variable or affecting memory in any way, the compiler assumes this calculation is unnecessary and thus ignores it. Ways to avoid this occurrence is to set compiler flags to prevent optimisation completely, or to force the operation to update some value in memory so that the compiler deems it as a necessary operation.

After fixing issues with compiler optimisation, timing measurements were still exhibiting unreliable and erroneous behaviour. Further research uncovered that this was likely due to a combination of issues with the RDTSC instruction assembly raised by Microsoft[9]. Each CPU core has its own TSC, but are not typically synchronised, as each CPU core does not start up at the exact same time. RDTSC naively assumes that the thread is always running on the same processor, meaning that thread jumps between processors can result in timing values that are out of sync when calls to RDTSC are made. Fixing this issue is not easy, as this process is "exacerbated when combined with modern power management technologies that idle and restore various cores at different times, which results in the cores typically being out of synchronization"[9]. These difficulties have arisen with the advancement of CPU technologies, and have lead to RDTSC not being a reliable method of obtaining accurate timing information.

4.2 Python

The abandonment of RDTSC left me with no access to any reliable timing API's in C++, forcing the consideration of alternate solutions. The first option was to use a different programming language. Python was the only option here, as it is the only other programming language I have sufficient experience with.

I carried out an experiment on Python's highest resolution timer, time.clock(), by repeated measurement of the runtime of computing 10^2. As can be seen in figure 2, after timing this operation 100000 times, there were a portion of erroneous measurements (0.1%). These outliers are likely due to garbage collection, an automatic procedure which manages memory. When objects no longer in use are occupying memory, the *garbage collector* reclaims this memory. This procedure causes large timing delays which can skew timing measurements if activated at the wrong time. As only approximately 0.1% of measurements are affected by this procedure, it is simple enough to sort a list of measurements by magnitude and discard a portion of the largest values. This portion is best worked out experimentally, as it may change depending on application.

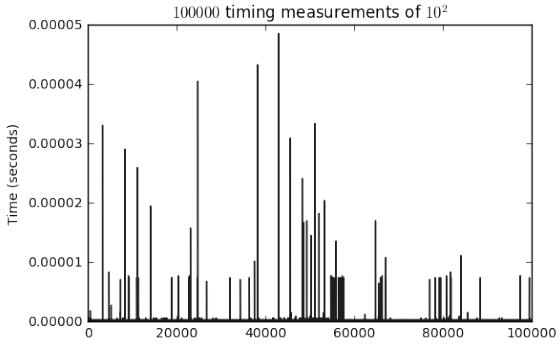
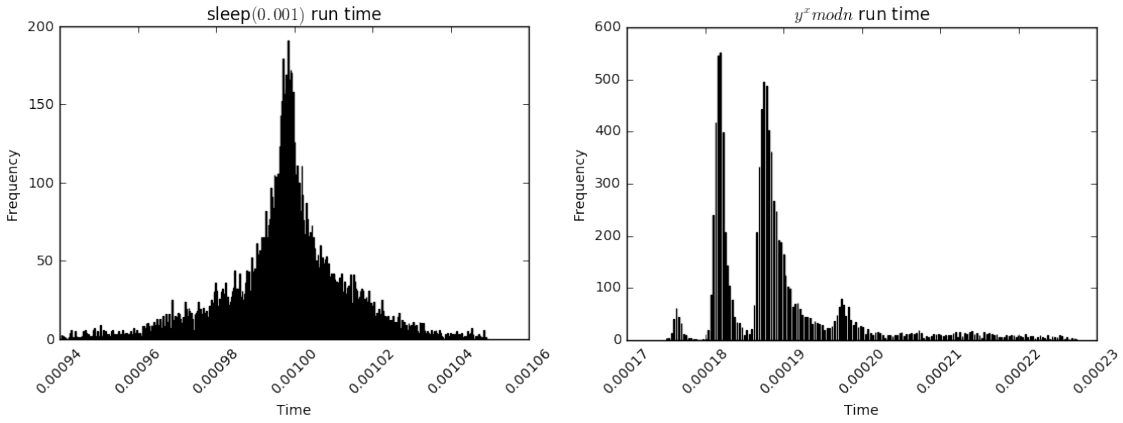


Figure 2: Illustration of outliers due to Garbage Collection

Measurements over 100000 trials

After removing outliers, I conducted further experiments to test the resolution of this timer. Figure 3a shows 10000 measurements of pythons sleep(0.001), which tells the processor to sleep for 0.001s. Although there is significant variance due to noise, there is a distinct peak at 0.001s, as expected. Figure 3b shows 10000 measurements of the the square and multiply algorithm 2 used in RSA, with constant parameters. This distribution has no clear mean, with several different peaks. This binning into multiple peaks demonstrates discrete behaviour, implying that resolution is not good enough at this time scale (10μs). From this, it is clear that Pythons timing API is not capable of providing reliable timing measurements at the scale of 10μs, which would be necessary for detecting the 2μs timing variations that occur when developing a timing attack on RSA.



(a) Sleep for 0.001 seconds

(b) RSA decryption

Figure 3: Comparison between large and small timing measurements

4.3 Synthetic Time

Here I propose the use of a synthetic time construct. This would be in the form of a global variable which is manually incremented at the instance of each calculation. For example, consider algorithm 4, the previously mentioned algorithm for a check function with the addition of synthetic time:

Algorithm 4 Password checker with synthetic time

```
let global time = 0
function CHECK(password)
    let time = time + 1
    if length(password) ≠ length(secret) then return False
    let i = 0
    let time = time + 1
    while (i ≤ length(password)) and (password[i] ≡ secret[i]) do
        let i = i + 1
        let time = time + 1
    return (i ≡ length(password))
```

In each occurrence of a new computation, the global time variable is incremented. In this example, the increment was simply 1 each time for demonstration, but experimentation can be done to find increments that are more representative of the true computation time.

In order to make the synthetic time construct more realistic, noise can be injected from a specified distribution. As figure 3a seems to exhibit a normal distribution of noise, I will inject Gaussian noise whilst using synthetic timing measurements. Although the use of Gaussian noise is not a definite representation of the true noise involved in timing channels, taking a greater amount of measurements will still make attacks translatable regardless of the underlying distribution.

Using synthetic time will allow strict management of noise contribution, thus providing a good environment to run experiments in. For instance, noise could be made negligible in order to test an attacks true capabilities of finding timing variations and using them accordingly. The noise can then be gradually increased, whilst being able to tweak the attack to cope with it.

5 Experiments

5.1 Attacking the Check Function

The first attack I developed was on algorithm 1, the password check function. This was written in C++11 with a synthetic timer, and is attached as '*checkattack.cpp*'. It can crack passwords with length of up to 250 alphanumeric characters, including capitals. This 250 length limit is adjustable.

Whilst attacking the length, instead of attempting a number of different lengths and choosing the one with the highest runtime, it increments successively until a condition is met to detect an outlier. This condition is when a timing measurement is greater than one standard deviation from the mean of the sample of other measurements. Different conditions were experimented with, but this one proved to work most optimally. Although rarely, noise can mask the detection of this timing variation. Because of this, a limit of length 250 was set to avoid a potential infinite loop. In the case that a variation is missed and the program reaches length 250, it resets the attack and repeats from length 1. This limit is easily adjustable if the user wishes to attack passwords of higher length.

Without access to any C++ timers, even ones with poor resolution, benchmarking this attack could only be done with rough experimentation via use of a stopwatch. There is a trade-off between rate of incorrect character guesses and run time. A reasonable balance was found with the use of 1000 trials done for each timing measurement, as this seemed to eradicate most errors whilst keeping the overall time of attack to a minimum. Attacks on 200 character passwords take roughly 10 seconds, with attack time increasing linearly with password length. Brute forcing a 200 character password would take $(26+26+10)^{200}$ operations, taking roughly in the region of 10^{300} years under the assumption of 10^{30} computations per second. This is an astronomical improvement.

The attack was tested with Gaussian noise with varying parameters. Varying the mean of the distribution has no impact, as the same mean will be applied to every measurement, thus the mean is set to 1. Variance is the variable which behaves like noise, and as such, the attack success is only affected by the variance. Figure 4 shows how the percentage of incorrect character guesses increases with variance. It is clear that the attack is stable up until the variance surpasses 10, in which case the error rate becomes significant and the attack may take several attempts. This can easily be combated by increasing the number of timing measurements taken, with a handicap of slightly longer attack times. Other noise distributions such as log-normal and exponential were also tested, giving similar results with different variance thresholds.

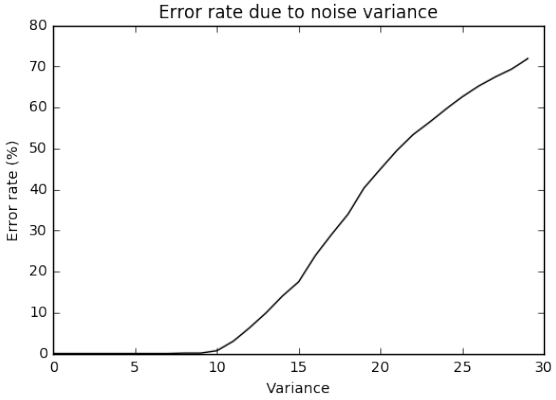


Figure 4: Error rate due to variance increase

5.2 Attacking RSA

5.2.1 The Attack

My attack on RSA was done using Dhem et al's method. The attack was written in Python 2.7 in a jupyter notebook, attached as *'RSAattack.ipynb'*. I coded my own RSA algorithm with the use of a open-source class built for Montgomery Reduction, written by Nayuki[2]. In order to check for extra reduction steps, I wrote a separate algorithm in this class which returns a binary value according to whether or not an extra step has occurred during a Montgomery multiplication.

Random primes $p = 857$ and $q = 641$ were chosen to get an $n = p \cdot q$ value sufficiently large to build an attack. These numbers are too small to be used in cryptography as they are too easy to brute force, however, the size of n past a certain point does not influence the capabilities of a timing attack as it not dependant on n. The mention of n being sufficiently large is due to the message size being constrained to $[1, n-1]$. If n is too small, the number of plausible messages may not be large enough to sample from in order to build a reliable attack.

As the Montgomery operations of importance are square operations, I investigated the likelihood of an extra step being taken for some $y^2 \mod n$ calculation. Figure 5 displays the number of extra steps taken during Montgomery multiplication over 1000 inputs. It was found that over a given input space, approximately 1% of these inputs will require an extra reduction step. Thus, for a sample size S corresponding to the number of samples in each subset of messages C_1, C_2, C_3, C_4, we need $n > 100S$. The chosen p, q values give an n value capable of accommodating up to $S \approx 5500$, which will be sufficient for my experiments.

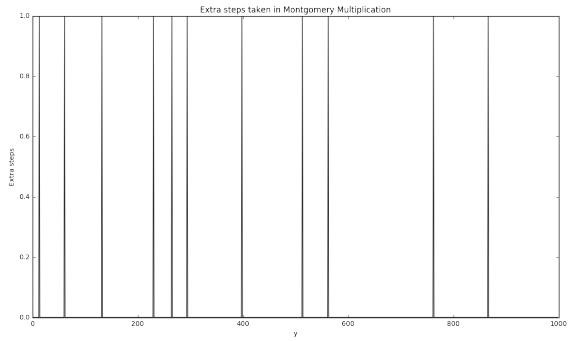


Figure 5: Extra steps taken in Montgomery multiplication over 1000 inputs (y)

Filtration of messages into their corresponding subsets was done by first inputting the revealed part of the private exponent into the square and multiply algorithm to find R_{temp}. This was then used to form the two conditions:

1. $R_{temp}^2 \mod n$

[2]https://www.nayuki.io/page/montgomery-reduction-algorithm

2. $(R_{temp} \cdot c \mod n)^2 \mod n$

The conditions were used in conjunction with the function used to check for extra Montgomery steps in order to separate a list of messages M into its relevant subsets C_1, C_2, C_3, C_4. As figure 5 shows, occurrence of extra steps taken are randomly distributed, so the subset sorting could be made no faster than a recursive loop through all M. By making M the set of all integers in $[1, n-1]$, this list can be sifted through until all subsets contain S samples, then sorting can be terminated. This results in 4 subsets of messages following the desired criteria, and must repeated for each subsequent bit guess.

Through experimentation, a value of $S = 1000$ was found to be sufficient for attacking 64-bit keys. S is relative to the size of the secret key, as the number of unknown exponent bits contributes towards noise amongst timing variations.

When taking a timing measurement, consistent and reliable results come from repeating the same measurement a number of times and taking an average. I will refer to the number of timings taken for one measurement as the number of *trials*. Figure 6 displays how the variance of a measurement decreases by increasing the number of trials. The number of trials has a trade-off with runtime, so finding a balance is necessary. A trial number of 100 has been chosen as it discards the majority of variance whilst keeping runtime minimal; the decrease in variance between 100 and 200 trials is not enough to warrant doubling the execution time.

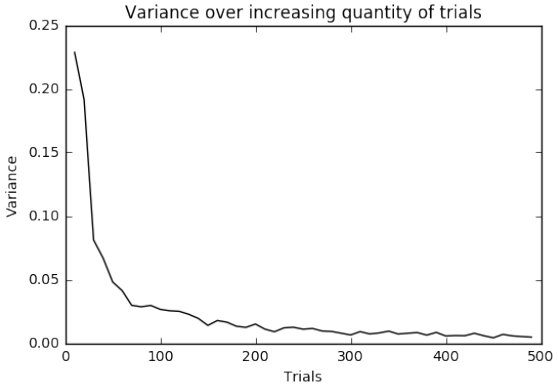


Figure 6: Decrease in variance due to increase in number of trials for a timing measurement

Synthetic time with Gaussian noise was used, $\mu = 1, \sigma^2 = 3$

5.2.2 Error Detection

Error detection was simple to implement. After each bit guess, a difference $D = \mid d_1 - d_2 \mid$ is stored, where $d_1 = \mid T_1 - T_2 \mid$, $d_2 = \mid T_3 - T_4 \mid$, the decision criteria mentioned in Section 3.3. Error detection is done by comparing $\alpha = D_{b-5}, ..., D_b$ with $\beta = D_1, ..., D_{b-5}$, i.e. the most recently stored 5 D values with those previous. 5 is chosen as opposed to 1, as there are cases where D is low although no error has been made, therefore it is best to wait until several low values have occurred before declaring an error. It is also due to this, that errors within the first 5 bit guesses cannot be detected. The comparison was done with a simple means test, concluding an error when $\overline{\alpha}$ is greater than 1 standard deviation from $\overline{\beta}$.

Errors are corrected by taking steps back, flipping a bit value, and continuing for 5 more bit guesses to see if D improves. If it doesn't improve, and additional step back is taken, and this process is repeated until the error no longer resides.

5.2.3 Results

Using a synthetic timer with normally distributed noise, with $\mu = 1, \sigma^2 = 3$, I managed to uncover a 64-bit key in 4.3 hours, at a rate of 4.1 minutes per bit. Dhem et al. reported that their attack was capable of cracking 64-bit keys at a rate of 20 bits per second, a much faster rate than my attack. Further optimisation of sample size and trial number will speed my attack up, but not to the scale of 20 bits per second. The primary reason for such a slow attack is likely due to to being written in Python, which is a significantly slower language than C++.

Although usage of Python and lack of optimisation may have a minimal effect on individual sections of code, due to the nature of timing attacks containing various nested loops, these seemingly insignificant timing differences quickly add up to create huge delays.

5.2.4 Remark

In this attack, length of secret key is assumed to be known. In cases where this length cannot be inferred from the public key and modulus, the attack can still work with appropriate adjustments. The principle that error detection works on is that if a bit guess is wrong, then there is no difference in error criteria. Following this, if one can detect that both bit guesses of 1 and 0 are wrong, then the only explanation is that this bit doesn't exist, and hence, the end of the secret key has been reached.

6 Further Improvements

One issue has been the reliance on a synthetic timer, which may not necessarily exhibit noise representative of the noise which occurs from real timing measurements. Further investigation into the true underlying noise distribution should be done, for instance, by the use of Kernel Density Estimations[10], which provide a method of estimating a probability density function (pdf) from a sample of data. By using a Kernel Density Estimation on a sample of noisy data, it'd be possible to get an estimation of a relevant distribution, which could then be used to inject synthetic noise which is more representative of a practical scenario.

As proposed in Section 5.2.3, the attack on RSA could be re-written in C++ in attempt to improve runtime. In addition to this, some parts of the attack are possible to parallelise. These parts include the filtration of messages into subsets, and simulation of modular exponentiation for large samples of messages.

The number of message samples used in the RSA attack can also be further optimised and experimented with. However, due to the current long runtime of this attack, it would take a very long time to gather enough data in order to draw meaningful conclusions that may be of use. This would become more plausible after improvements are made to the attacks runtime.

7 Conclusion

Timing attacks pose a serious risk to cryptographic algorithms, and as such should be carefully investigated with the intent to expose any timing leaks. Algorithms which were previously assumed to be the most secure of their kind have been shown to exhibit timing leaks leaving them vulnerable to attacks in 0 day timing windows. Techniques can be used to hide any potentially dangerous timing variations, such as signature blinding, which comes with a 2-10% cost to performance.

References

[1] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[2] Paul C. Kocher. *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*, pages 104–113. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[3] RSA Labratories. Rsaref: A cryptographic toolkit, 1994.

[4] David Chaum. *Blind Signatures for Untraceable Payments*, pages 199–203. Springer US, Boston, MA, 1983.

[5] Jean-François Dhem, François Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. *A Practical Implementation of the Timing Attack*, pages 167–182. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[6] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985.

[7] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701 – 716, 2005. Web Security.

[8] Microsoft. Read time-stamp counter. `https://msdn.microsoft.com/en-us/library/twchhe95.aspx`. Accessed: 08/08/2017.

[9] Microsoft. Game timing and multicore processors. `https://msdn.microsoft.com/en-us/library/ee417693(VS.85).aspx`. Accessed: 08/08/2017.

[10] Emanuel Parzen. On estimation of a probability density function and mode. *The annals of mathematical statistics*, 33(3):1065–1076, 1962.