# Remote Timing Attacks are Practical

David Brumley
dbrumley@stanford.edu

Dan Boneh
dabo@cs.stanford.edu

**Abstract**

Timing attacks are usually used to attack weak computing devices such as smartcards. We show that timing attacks apply to general software systems. Specifically, we devise a timing attack against OpenSSL. Our experiments show that we can extract private keys from an OpenSSL-based web server running on a machine in the local network. Our results demonstrate that timing attacks against network servers are practical and therefore all security systems should defend against them.

## 1 Introduction

Timing attacks enable an attacker to extract secrets maintained in a security system by observing the time it takes the system to respond to various queries. For example, Kocher [9] designed a timing attack to expose secret keys used for RSA decryption and signing. Until now, these attacks were only applied in the context of hardware security tokens such as smartcards [9, 16, 5]. It is generally believed that timing attacks cannot be used to attack general purpose servers, such as web servers, since decryption times are masked by many concurrent processes running on the system. It is also believed that common implementations of RSA (using Chinese Remainder and Montgomery reductions) are not vulnerable to timing attacks.

We challenge both assumptions by developing a remote timing attack against OpenSSL [13], an SSL library commonly used in web servers and other SSL applications. Our attack client measures the time an OpenSSL server takes to respond to decryption queries. The client is able to extract the private key stored on the server. The attack applies in several environments.

**Local network.** We successfully mounted our timing attack between two machines connected via an Ethernet switch. We were able to extract the SSL private key from common SSL applications such as a web server (Apache mod_SSL) and SSL-tunnel.

**Interprocess.** We successfully mounted the attack between two processes running on the same machine. A hosting center that hosts two domains on the same machine might give management access to the admins of each domain. Since both domain are hosted on the same machine, one admin could use the attack to extract the secret key belonging to the other domain.

**Virtual Machines.** A Virtual Machine Monitor (VMM) is often used to enforce isolation between two Virtual Machines (VM) running on the same processor. One could protect an RSA private key by storing it in one VM and enabling other VM's to make decryption/signing queries. For example, a web server could run in one VM while the private key is stored in a separate VM. This is a natural way of protecting secret keys since a break-in into the web server VM does not expose the private key. Our results show that when using OpenSSL the network server VM can extract the RSA private key from the secure VM, thus invalidating the isolation provided by the VMM. This is especially relevant to VMM projects such as

Microsoft's Palladium architecture. We also note that Palladium enables an application to ask the VMM (aka Nexus) to decrypt (aka unseal) application data. The application could expose the VMM's secret key by measuring the time the VMM takes to respond to such requests.

Many crypto libraries completely ignore the timing attack and have no defenses implemented to prevent it. For example, libgcrypt [6] (used in GNUTLS and GPG) and Cryptlib [7] do not defend against timing attacks. OpenSSL implements a defense against the timing attack as an option. However, common applications such as mod_SSL, the Apache SSL module, does not turn this option on and is therefore vulnerable to the attack. These examples show that timing attacks are a largely ignored vulnerability in many crypto implementations. We hope the results in this paper will convince developers to implement proper defenses (see Section 6). Interestingly, Mozilla's NSS crypto library properly defends against the timing attack. We note that most crypto acceleration cards also implement defenses against the timing attack. Consequently, network servers using these accelerator cards are not vulnerable.

We chose to tailor our timing attack to OpenSSL since it is the most widely used open source SSL library. The OpenSSL implementation of RSA is highly optimized using Chinese Remainder, Sliding Windows, Montgomery multiplication, and Karatsuba's algorithm. These optimizations cause both known timing attacks on RSA [9, 16] to fail in practice. Consequently, we had to devise a new timing attack based on [16] that is able to extract the private key from an OpenSSL-based server. As we will see, the performance of our attack varies with the exact environment in which it is applied. Even the exact compiler optimizations used to compile OpenSSL can make a big difference.

In Sections 2 and 3 we describe OpenSSL's implementation of RSA and the timing attack on OpenSSL. In Section 5 we describe the actual experiments we carried out. We show that using about a million queries we can remotely extract a 1024-bit RSA private key from an OpenSSL server implemented using OpenSSL 0.9.7. The attack takes about two hours. In Section 4 we discuss how these attacks apply to web servers using SSL.

Timing attacks are related to a class of attacks called side-channel attacks. These include power analysis [8] and attacks based on electromagnetic radiation [14]. Unlike the timing attack, these extended side channel attacks require special equipment and physical access to the machine. In this paper we only focus on the timing attack. We also note that our attack targets the implementation of RSA decryption in OpenSSL. We do not use timing attacks on the RSA padding used in SSL and TLS [2] since those attacks are less efficient and are specific to SSL/TLS.

# 2 OpenSSL's Implementation of RSA

We begin by reviewing how OpenSSL implements RSA decryption and signing. We only review the details needed for our attack. OpenSSL closely follows algorithms described in the Handbook of Applied Cryptography [10], where more information is available.

## 2.1 OpenSSL Decryption

At the heart of RSA decryption is a modular exponentiation $m = c^d \bmod N$ where $N = pq$ is the RSA modulus, $d$ is the private decryption exponent, and $c$ is the ciphertext being decrypted. OpenSSL uses the Chinese Remainder Theorem (CRT) to perform this exponentiation. With Chinese remaindering, the function $m = c^d \bmod N$ is computed in two steps. First, evaluate

$m_1 = c^{d_1} \bmod p$ and $m_2 = c^{d_2} \bmod q$ (here $d_1$ and $d_2$ are precomputed from $d$). Then, combine $m_1$ and $m_2$ using CRT to yield $m$.

RSA decryption with CRT gives up to a factor of four speedup, making it essential for competitive RSA implementations. RSA with CRT is not vulnerable to Kocher's original timing attack [9]. Nevertheless, since RSA with CRT uses the factors of $N$, a timing attack can expose these factors. Once the factorization of $N$ is revealed it is easy to obtain the decryption key $d$.

### 2.2 Exponentiation

During an RSA decryption with CRT, OpenSSL computes $c^{d_1} \bmod p$ and $c^{d_2} \bmod q$. Both computations are done using the same code. For simplicity we describe how OpenSSL computes $g^d \bmod q$ for some $g, d$, and $q$.

The simplest algorithm for computing $g^d \bmod q$ is *square and multiply*. The algorithm, squares $g$ approximately $\log_2 d$ times, and performs approximately $\frac{\log_2 d}{2}$ additional multiplications by $g$. After each step, the product is reduced modulo $q$.

OpenSSL uses an optimization of square and multiply called *sliding windows* exponentiation. The central difference in sliding windows is that a block of bits (window) of $d$ are processed at each iteration, where as square-and-multiply processes only one bit of $d$ per iteration. Sliding windows requires precomputation proportional to $2^w$ for a window of size $w$. Thus, there is an optimal window size that balances the time spent during precomputation vs. actual exponentiation. In OpenSSL, for a 1024 bit modulus the optimum window size is five so that $d$ is processed five bits at a time.

For our attack, the key fact about sliding windows is that during the algorithm there are many multiplications by $g$, where $g$ is the input ciphertext. By querying on many inputs $g$ the attacker can expose information about bits of the factor $q$. We note that a timing attack on sliding windows is much harder than a timing attack on square-and-multiply since there are far fewer multiplications by $g$ in sliding windows. As we will see, we had to adapt our sampling techniques to handle sliding windows exponentiation used in OpenSSL.

### 2.3 Montgomery Reduction

The sliding windows exponentiation algorithm performs a few modular multiplications at every step. Given two integers $x, y$, computing $x \cdot y \bmod q$ is done by first multiplying the integers $x \cdot y$ and then reducing the result modulo $q$. We first briefly describe OpenSSL's modular reduction method and then describe its integer multiplication algorithm.

Naively, a reduction modulo $q$ is done via multi-precision division and returning the remainder. This is quite expensive. In 1985 Peter Montgomery discovered a method for implementing a reduction modulo $q$ using a series of operations efficient in hardware and software [12].

Montgomery reduction transforms a reduction modulo $q$ into a reduction modulo some power of 2 denoted by $R$. A reduction modulo a power of 2 is faster than a reduction modulo $q$. However, in order to use Montgomery reduction all variables must first be put into Montgomery form. The Montgomery form of number $x$ is simply $xR \bmod q$. To multiply two numbers $a$ and $b$ in Montgomery form we do the following. First, compute their product as integers: $aR \cdot bR = cR^2$. Then, use the fast Montgomery reduction algorithm to compute $cR^2 * R^{-1} = cR \bmod q$ which is the product of $a$ and $b$ in Montgomery form. At the end of the exponentiation algorithm the output is put back in standard form by multiplying it by $R^{-1} \bmod q$. Note that $R$ and $R^{-1} \bmod N$ are public.

Hence, for the small penalty of converting the input $g$ to Montgomery form, a large gain is incurred during modular reduction. With typical RSA parameters the gain from Montgomery reduction outweighs the cost of initially putting numbers in Montgomery form and converting back at the end of the algorithm.

The key relevant fact about Montgomery reduction is the last step. At the end of each Montgomery reduction one checks if the output $cR$ is greater than $q$. If so, one subtracts $q$ from the output, to ensure that the output $cR$ is in the range $[0, q]$. This extra step is called an *extra reduction* and causes a timing difference for different inputs. Schindler [16] noticed that the probability of an extra reduction during an exponentiation $g^d \bmod q$ is proportional to how close $g$ is to $q$. Schindler showed that the probability for an extra reduction is:

$$\Pr[\texttt{Extra Reduction}] = \frac{g \bmod q}{2R} \tag{1}$$

Consequently, as $g$ approaches either factor $p$ or $q$ from below, the number of extra reductions during the exponentiation algorithm greatly increases. At exact multiples of $p$ or $q$, the number of extra reductions drops dramatically. Figure 1 shows this relationship, with the discontinuities appearing at multiples of $p$ and $q$. By detecting timing differences that result from extra reductions we can tell how close $g$ is to a multiple of one of the factors.

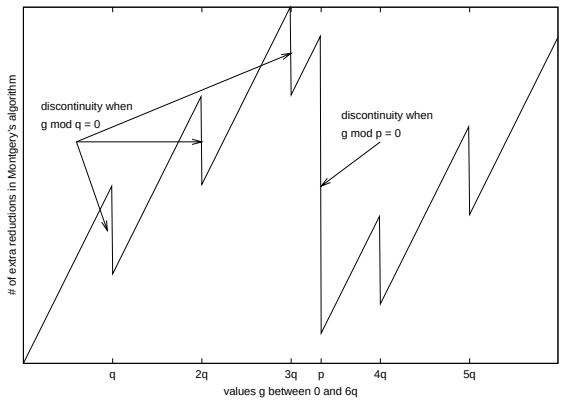



Figure 1: Number of Extra Reductions as a function of $g$.

## 2.4 Multiplication Routines

Montgomery's method makes use of a multi-precision integer multiplication routine. OpenSSL implements two multiplication routines: Karatsuba (sometimes called recursive) and "normal". Multi-precision libraries represent large integers as a sequence of words. OpenSSL uses Karatsuba multiplication when multiplying two numbers with an equal number of words, and runs in time $O(n^{1.58})$. OpenSSL uses normal multiplication, which runs in time $O(nm)$, when multiplying two numbers with an unequal number of words of size $n$ and $m$.

Thus, OpenSSL's integer multiplication routine leaks important timing information. Since Karatsuba is typically faster, multiplication of two unequal size words takes longer than multiplication of two equal size words. Thus, timing information reveals how frequently the operands given to the multiplication routine have the same length. We use this fact to improve the efficiency of the timing attack on OpenSSL.

In both algorithms, multiplication is ultimately done on individual words. The underlying word multiplication algorithm dominates the total time for a decryption. For example, in OpenSSL

the underlying word multiplication routine typically takes $30\% - 40\%$ of the total runtime. The time to multiply individual words depends on the number of bits per word. Therefore, the exact architecture on which OpenSSL runs has an impact on timing measurements used for the attack. In our experiments the word size was 32 bits.

### 2.5 Summary: Comparison of Timing Differences

So far we identified two sources of variance in the time to do an RSA decryption: (1) Schindler's observation on the number of extra reductions in a Montgomery reduction, and (2) the timing difference due to the choice of multiplication routine, i.e. Karatsuba vs. normal. Unfortunately, the effects of these optimizations counteract one another.

Consider a timing attack where we decrypt a ciphertext $g$. As $g$ approaches a multiple of $q$ from below, equation (1) tells us that the number of extra reductions in a Montgomery reduction increases. When we are just over a multiple of $q$, the number of extra reductions decreases dramatically. In other words, decryption of $g$ less than $q$ should be slower than decryption of $g$ greater than $q$.

The choice of Karatsuba vs. normal multiplication has the opposite effect. When $g$ is just below a multiple of $q$, then OpenSSL almost always uses fast Karatsuba multiplication. When $g$ is just over a multiple of $q$ then $g \bmod q$ is small and consequently most multiplications will be of integers with different lengths. Therefore, OpenSSL uses normal multiplication which is slower. In other words, decryption of $g$ less than $q$ should be faster than decryption of $g$ greater than $q$ — the exact opposite of the effect of extra reductions in Montgomery's algorithm. Which effect dominates is determined by the exact environment. Our attack uses both effects, but each effect is used at a different phase of the attack.

## 3 A Timing Attack on OpenSSL

Our attack exposes the factorization of the RSA modulus. Let $N = pq$ with $q < p$. We build approximations to $q$ that get progressively closer as the attack proceeds. We call these approximations guesses. We refine our guess by learning bits of $q$ one at a time, from most significant to least. Thus, our attack can be viewed as a binary search for $q$. After recovering the half-most significant bits of $q$, we can use Coppersmith's algorithm [4] to retrieve the complete factorization.

Initially our guess $g$ of $q$ lies between $2^{512}$ (i.e. $2^{\log_2 N/2}$) and $2^{511}$ (i.e. $2^{\log_2 (N/2)-1}$). We then time the decryption of all possible combinations of the top few bits (typically 2-3). When plotted, the decryption times will show two peaks: one for $q$ and one for $p$. We pick the values that bound the first peak, which in OpenSSL will always be $q$.

Suppose we already recovered the top $i-1$ bits of $q$. Let $g$ be an integer that has the same top $i-1$ bits as $q$ and the remaining bits of $g$ are 0. Then $g < q$. At a high level, we recover the $i$'th bit of $q$ as follows:

- Step 1 - Let $g_{hi}$ be the same value as $g$, with the $i$'th bit set to 1. If bit $i$ of $q$ is 1, then $g < g_{hi} < q$. Otherwise, $g < q < g_{hi}$.
- Step 2 - We measure the time to decrypt both $g$ and $g_{hi}$. Let $t_1 = \text{DecryptTime}(g)$ and $t_2 = \text{DecryptTime}(g_{hi})$.
- Step 3 - We calculate the difference $\Delta = |t_1 - t_2|$. If $g < q < g_{hi}$ then, by Section 2.5, the difference $\Delta$ will be "large", and bit $i$ of $q$ is 0. If $g < g_{hi} < q$, the difference $\Delta$ will be "small", and bit $i$ of $q$ is 1. We use previous $\Delta$ values to know what to consider "large" and "small". Thus we use the value $|t_1 - t_2|$ as an indicator for the $i$'th bit of $q$.

When the $i$'th bit is 0, the "large" difference can either be negative or positive. In this case, if $t_1 - t_2$ is positive then DecryptTime($g$) > DecryptTime($g_{hi}$), and the Montgomery reductions dominated the time difference. If $t_1 - t_2$ is negative, then DecryptTime($g$) < DecryptTime($g_{hi}$), and the multi-precision multiplication dominated the time difference.

Formatting of RSA plaintext, e.g. PKCS 1, does not effect this timing attack. We also do not need the value of the decryption, only how long the decryption takes.

### 3.1 Exponentiation Revisited

We would like $|t_{g_1} - t_{g_2}| \gg |t_{g_3} - t_{g_4}|$ when $g_1 < q < g_2$ and $g_3 < g_4 < q$. Time measurements that have this property we call a strong indicator for bits of $q$, and those that do not are a weak indicator for bits of $q$. Square and multiply exponentiation results in a strong indicator because there are approximately $\frac{\log_2 q}{2}$ multiplications by $g$ during decryption. However, in sliding windows with window size $w$ ($w = 5$ in OpenSSL) the expected number of multiplications by $g$ is only:

$$E[\texttt{\# multiply by g}] = \frac{\log_2 q}{2^{w+1} + 1}$$

resulting in a weaker indicator.

To overcome this we query at a *neighborhood* of values $g, g+1, g+2, ..., g+n$, and use the result as the decrypt time for $g$ (and similarly for $g_{hi}$). The total decryption time for $g$ or $g_{hi}$ is then:

$$T_g = \sum_{i=0}^{n} \texttt{DecryptTime}(g + i)$$

We define $T_g$ as the time to compute $g$ with sliding windows when considering a neighborhood of values. As $n$ grows, $|T_g - T_{g_{hi}}|$ typically becomes a stronger indicator for a bit of $q$ (at the cost of additional decryption queries).

## 4 Real-world scenarios

As mentioned in the introduction there are a number of scenarios where the timing attack applies to networked servers. We discuss an attack on SSL applications, such as stunnel [17] and an Apache web server with mod_SSL [11], and an attack on trusted computing projects such as Palladium.

During a standard full SSL handshake the SSL server performs an RSA decryption using its private key. The SSL server decryption takes place after receiving the CLIENT-KEY-EXCHANGE message from the SSL client. The CLIENT-KEY-EXCHANGE message is composed on the client by encrypting a PKCS 1 padded random bytes with the server's public key. The randomness encrypted by the client is used by the client and server to compute a shared master secret for end-to-end encryption.

Upon receiving a CLIENT-KEY-EXCHANGE message from the client, the server first decrypts the message with its private key and then checks the resulting plaintext for proper PKCS 1 formatting. If the decrypted message is properly formatted, the client and server can compute a shared master secret. If the decrypted message is not properly formatted, the server generates its own random bytes for computing a master secret and continues the SSL protocol. Note that an improperly formatted CLIENT-KEY-EXCHANGE message prevents the client and server from computing the same master secret, ultimately leading the server to send an ALERT message to the client indicating the SSL handshake has failed.

In our attack, the client substitutes a properly formatted CLIENT-KEY-EXCHANGE message with our guess $g$. The server decrypts $g$ as a normal CLIENT-KEY-EXCHANGE message, and then checks the resulting plaintext for proper PKCS 1 padding. Since the decryption of $g$ will not be properly formatted, the server and client will not compute the same master secret, and the client will ultimately receive an ALERT message from the server. The attacking client computes the time difference from sending $g$ as the CLIENT-KEY-EXCHANGE message to receiving the response message from the server as the time to decrypt $g$. The client repeats this process for each value of of $g$ and $g_{hi}$ needed to calculate $T_g$ and $T_{g_{hi}}$.

Our experiments are also relevant to trusted computing efforts such as Palladium. One goal of Palladium is to enable a remote third party to validate the code running on the machine. This is done through the process of attestation: an application running on the user's machine requests the Virtual Machine Monitor (VMM, aka Nexus) to sign the application's binary image. The VMM responds with a signature which the application could then send to the third party. The timing attack shows that by making repeated requests and measuring the time the VMM takes to respond, the application could expose the VMM's secret signing key. Therefore, it is essential that VMM implementations defend against this timing attack.

We note that RSA applications (and subsequently SSL applications using RSA for key exchange) using a hardware crypto accelerator are not vulnerable since most crypto accelerators implement defenses against the timing attack. Our attack applies to software based RSA implementations that do not defend against timing attacks as discussed in section 6.

# 5 Experiments

We performed a series of experiments to demonstrate the effectiveness of our attack on OpenSSL. In each case we obtained the factorization of the RSA modulus $N$. We note that compile-time and source-based optimizations of OpenSSL effect how efficiently we can recover the factorization.

Our experiments consisted of:

1. Testing the effects of increasing the number of decryption requests, both for the same ciphertext and a neighborhood of ciphertexts.
2. Compare the effectiveness of the attack based upon different keys.
3. Compare the effectiveness of the attack based upon common compile-time optimizations.
4. Compare the effectiveness of the attack based upon source-based optimizations.
5. Compare inter-process vs. local network attacks.
6. Compare the effectiveness of the attack against two common SSL applications: an Apache web server with mod_SSL and stunnel.

The first four experiments were carried out inter-process via TCP. The fifth experiment demonstrates our attack works on the local network. The last experiment demonstrates our attack succeeds on the local network against common SSL-enabled applications.

## 5.1 Experiment Setup

Our attack was performed against OpenSSL 0.9.7 (the current version as of this paper). All tests were run under Linux 2.4.18-17 on a 2.4 GHZ Pentium 4 processor, using gcc 2.96 (RedHat), with 1 GB of RAM. All keys were generated at random via OpenSSL's key generation routine.

For the first 5 experiments, we implemented a simple TCP server that read an ASCII string, converted the string to OpenSSL's internal multi-precision representation, then performed the RSA

decryption. The server returned 0 to signify the end of decryption. The TCP client measured the time from writing the ciphertext over the socket to receiving the reply. These first experiments directly characterize the vulnerability in OpenSSL's RSA decryption routines. The remaining experiments show that common SSL-enabled applications, such as Apache mod_SSL and stunnel, are also vulnerable to our attack.

Our timing attack requires a clock with fine resolution. We use the Pentium cycle counter on the attacking machine as such a clock, giving us a time resolution of 2.4 billion ticks per second. The cycle counter increments once per clock tick, regardless of the actual instruction issued. Thus, the decryption time is the cycle counter difference between sending the ciphertext to receiving the reply. The cycle counter is accessible via the "rdtsc" instruction, which returns the 64-bit cycle count since CPU initialization. The high 32 bits are returned into the EDX register, and the low 32 bits into the EAX register. Accessing the cycle counter is not a privileged operation. Other architectures have a similar a counter, such as the UltraSparc %tick register.

OpenSSL generates RSA moduli $N = pq$ where $q < p$. In each case we target the smaller factor, $q$. Once $q$ is known, the RSA modulus is factored and, consequently, the server's private key is exposed.

### 5.2 Experiment 1 - Number of Ciphertexts

This experiment explores the parameters that determine the number of queries needed to break an RSA key. For any particular bit of $q$, we have a guess $g$ that is compromised of two parameters: sample size and neighborhood size. We call the number of values $g, g+1, g+2, ..., g+n$ needed to overcome windowing the neighborhood size, denoted by $n$. We call the number of decryption queries needed for the time to converge for a particular $g+k$ the sample size, denoted by $s$. The total number of queries needed to compute $T_g$ is then $s * n$.

To overcome the effects of a multi-user environment, we repeatedly sample $g+k$ and use the median time value as the effective decryption time. Figure 2(a) shows the difference between median values as sample size increases.

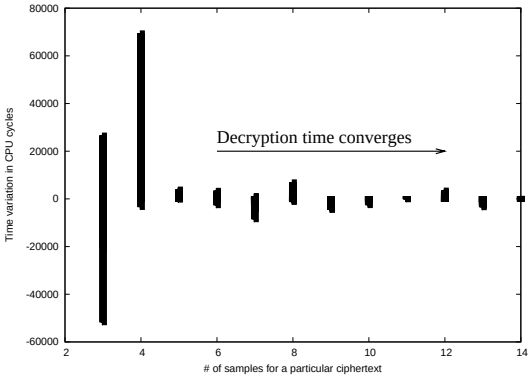
The number of samples required to reach a stable decryption time is surprising small, requiring only 5 samples to give a variation of under 20000 cycles (approximately 8 microseconds), well under that needed to perform a successful attack. To be safe, we used a sample size of 7 in all subsequent experiments.

We call the gap between when a bit of $q$ is 0 and 1 the *zero-one* gap. This gap can be thought of as the difference $T_g - T_{g_{hi}}$. The larger the gap, the stronger the indicator for bit $i$, and the smaller chance of error.

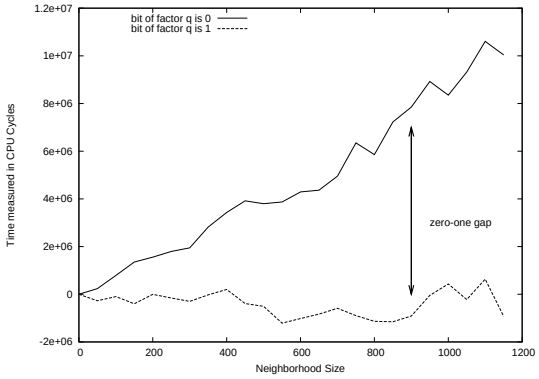
Recall for a particular bit $i$ of $q$, if $g < q < g_{hi}$ then bit $i$ of $q$ is 0. Similarly, if $g < g_{hi} < q$, then bit $i$ of $q$ is 1. Figure 2(b) shows the time difference between these cases. The upper line is when bit $i$ of $q$ is 0, and the lower line is when bit $i$ of $q$ is 1. The distance between these two lines varies for different bits of $q$. This graph shows how for a particular bit of $q$ increasing the neighborhood size tends to increases the time difference $|T_g - T_{g_{hi}}|$ that is the zero-one gap.

### 5.3 Experiment 2 - Different Keys

We attacked several 1024-bit keys, each randomly generated, to determine the ease of breaking different moduli. In each case we were able to recover the factorization of $N$. We used a constant neighborhood size of 400 and sample size of 7 in order to make the differences between keys meaningful. Figure 3(a) shows our results for 3 different keys. For clarity, we include only bits of




(a) Decreasing time variance by repeated sampling of a particular query to increase the accuracy of our time measurement




(b) By increasing the neighborhood size we increase the zero-one gap between two adjacent bits of q where one bit is 1 and the other is 0

Figure 2: Parameters that effect the time to decrypt guess $g$

$q$ that are 0, as bits of q that are 1 are close to $f(x) = 0$. In all our figures the time difference $T_g - T_{g_{hi}}$ is the zero-one gap.

For 2 out of 3 keys, the effects of Montgomery multiplication dominate bits $> 32$, and the zero-one gap is positive. For key 3, the multi-precision multiplication routine dominates from bits 32 to about 187, as indicated by a negative zero-one gap for bits $> 32$, after which Montgomery reductions take over. For the first 32 most significant bits on all keys, the effect of the word-multiplication routine dominate, signified by the case where the zero-one gap is negative. The word-multiplication routine dominates here because the multiplication by $g$ mod $q$ can be more efficiently scheduled in the processor than $g_{hi}$ mod $q$ (see experiment 4.4 for more on resource scheduling).

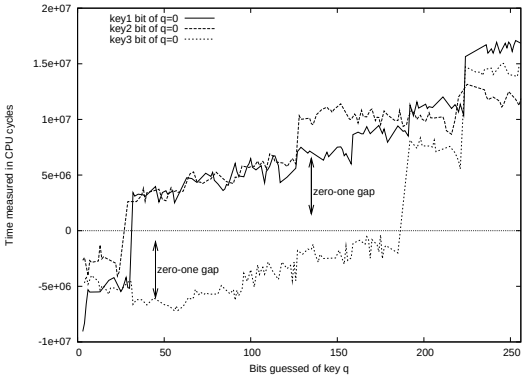
The total number of queries is $2ns{*}\log_2 N/4$, where $s$ is the sample size and $n$ is the neighborhood size. We are sampling two values, $g$ and $g_{hi}$, with neighborhood $n = 400$ and sample size $s = 7$. The entire attack uses 1433600 queries. The total wall clock time spent on the attack depends upon processor speed, but in our test bed was approximately 2 hours. In practice, an effective attack needs far fewer samples, as the neighborhood size can be adjusted dynamically to give a clear zero-one gap in the smallest number of queries.

For key 3, bits 150-200, $|T_g - T_{g_{hi}}|$ is small, resulting in a weak indicator. As discussed previously we can increase the size of the neighborhood to increase $|T_g - T_{g_{hi}}|$, giving a stronger indicator. Figure 3(b) shows the effects of increasing the neighborhood size from 400 to 1000, resulting in a strong enough indicator to mount a successful attack on bits 150-200 on key 3.

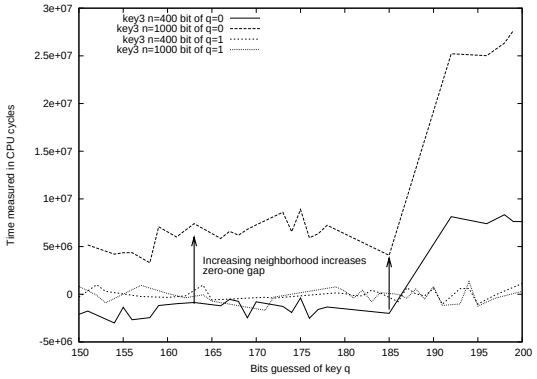
### 5.4 Experiment 3 - Compile-time Optimizations

The total number of decryption queries required for a successful attack depends upon how OpenSSL is compiled. To test the effects of compile-time optimizations, we compiled OpenSSL three different ways:

- **Optimized** (-O3 -fomit-frame-pointer -mcpu=pentium): The default OpenSSL flags for Intel. -O3 is the optimization level, -fomit-frame-pointer omits the frame pointer, thus freeing up an extra register, and -mcpu=pentium enables more sophisticated resource scheduling.




(a) Comparing the zero-one gap for different keys. The plot only includes bits of q that are 0 for clarity. f(x)=0 should be used as a reference for when a bit of q is 1.




(b) Larger neighborhood size increases zero-one gap for hard bits of key 3

Figure 3: Breaking 3 RSA Keys by looking at the zero-one gap time difference

- **No Pentium flag** (-O3 -fomit-frame-pointer): The same as the above, except without -mcpu sophisticated resource scheduling is not done, and an i386 architecture is assumed.
- **Unoptimized** (-g -mcpu=pentium): Enable debugging support.

Each method shows a timing attack is practical, but changes the number of queries necessary and which timing factor dominates. Figure 4 compares the results of each test.

For readability, we only show the difference $T_g - T_{g_{hi}}$ when $g < q < g_{hi}$, i.e. when bit $i$ of $q$ is 0. The case where bit $i = 1$ shows little variance based upon the optimizations. $y = 0$ can be used as reference for the case when bit $i = 1$.

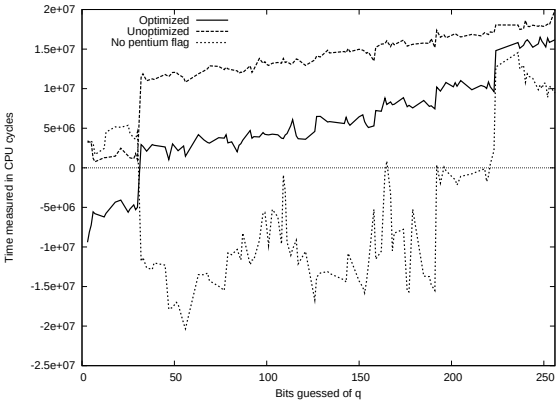



Figure 4: How compile-time flags shift the zero-one gap.

We expected Montgomery reductions to dominate when guessing the first 32 bits, switching to Karatsuba vs. normal multiplication thereafter. When this happens, the zero-one gap is positive for the first 32 bits, and negative elsewhere. In our tests, this only occurred when -mcpu was omitted. This was quite surprising, but can be explained by taking into consideration the enhanced scheduling -mcpu provides the underlying word-based routines.

For the optimized and unoptimized case, the attacker can reduce the neighborhood size as bits are learned to minimize the total overall queries while still maintaining a strong indicator. However, when omitting -mcpu, the attacker must increase the neighborhood size around bit 125, 175, and 200 to receive a strong indicator. The indicator becomes less distinct in the later bits because the effects of Montgomery reduction (and the shorter $g \bmod q$) begin to significantly counteract the effects of Karatsuba vs. normal multiplication.

In these tests again approximately 1.4 million decryption queries were made. We note that without optimizations, separate tests allowed us to recover the factorization with less than 359000 queries. This number could further be reduced by dynamically reducing the neighborhood size as bits of $q$ are learned.

Our tests of OpenSSL 0.9.6g were similar to the results of 0.9.7 with -mcpu omitted. Interestingly, version 0.9.6g uses -m486 instead of -mcpu=pentium. The 486 architecture does not have sophisticated resource scheduling, so the attack against OpenSSL compiled against 486 architecture is similar to using -g on a Pentium.

One conclusion we draw is that users of binary crypto libraries may find it hard to characterize their risk to our attack without complete understanding of the compile-time options and environment. Common flags such as enabling debugging support allow our attack to recover the factors of a 1024-bit modulus in less than 1/3 million queries.

### 5.5 Experiment 4 - Source-based Optimizations

We implemented a minor patch that improves the efficiency of the OpenSSL 0.9.7 CRT decryption check. Our patch has been accepted for future incorporation to OpenSSL (tracking ID 475). After a CRT decryption, OpenSSL re-encrypts the result (mod $N$) and verifies the result is identical to the original ciphertext. This verification step prevents a incorrect CRT decryption from revealing the factors of the modulus [3]. By default, OpenSSL needlessly recalculates both Montgomery parameters $R$ and $R^{-1}$ on every decryption. Our minor patch allows OpenSSL to cache both values between decryptions with the same key. Our patch does not affect any other aspect of the RSA decryption other than caching these values. Figure 5 shows the results of an attack both with and without the patch.

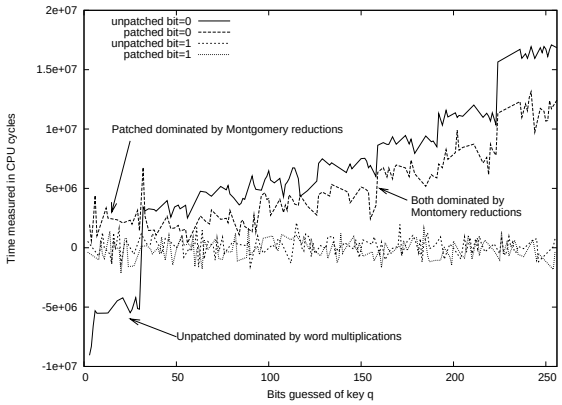



Figure 5: How source-based optimizations change the zero-one gap

Without the patch, OpenSSL spends more time (proportionally to the other experiments) in the first 32 bits doing word-based multiplication, resulting in a negative zero-one gap. With the patch, the time spent doing word-based routines drops, and the effects of Montgomery reduction's

dominate. This illustrates that optimizations can effect the specific attack characteristics, but not the overall vulnerability of exposing the factorization.

### 5.6 Experiment 5 - Interprocess vs. Local Network Attacks

We show that local network timing attacks are practical. We connected two computers via a 10/100 Mb Hawking switch, and compared the results of the attack inter-process vs. inter-network. Figure 6 shows that the network does not diminish the effectiveness of the attack. The noise from the network is eliminated by repeated sampling, giving a similar zero-one gap to inter-process. We note that in our test a zero-one gap of approximately 1 millisecond is a sufficient to receive a strong indicator, enabling a successful attack. Thus, networks with less than 1ms of variance for transmitting the attack packets are vulnerable.

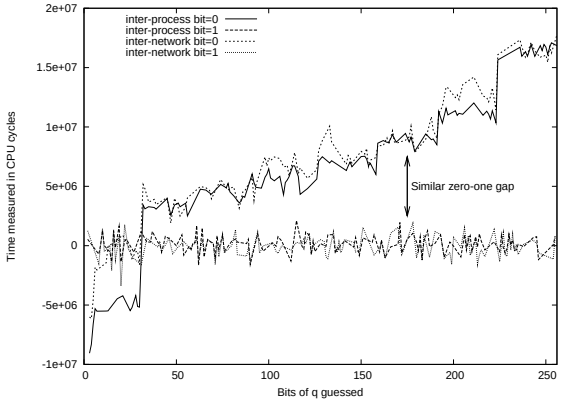



Figure 6: Inter-process vs. Inter-network zero-one gap

Inter-network attacks allow an attacker to also take advantage of faster CPU speeds for increasing the accuracy of timing measurements. Consider machine 1 with a slower CPU than machine 2. Then if machine 2 attacks machine 1, the faster clock cycle allows for finer grained measurements of the decryption time on machine 1. Finer grained measurements should result in fewer queries for the attacker, as the zero-one gap will vary less.

### 5.7 Experiment 6 - Attacking SSL Applications on the Local Network

We show that two common OpenSSL applications are vulnerable to our attack. We compiled Apache 1.3.27 with mod_SSL 2.8.12 and stunnel 4.04 per the respective "INSTALL" files accompanying the software. Apache with mod_SSL is a commonly used secure web server. stunnel allows TCP/IP connections to be tunneled through SSL.

Figure 7 shows the result of attacking stunnel and mod_SSL. For reference, we also include the results for a similar attack against the simple RSA decryption server from the previous experiments. For clarity, we again only show the difference $T_g - T_{g_i}$ when $g < q < g_{hi}$, i.e. when bit $i$ of $q$ is 0. Similar to experiment 3, the line $y = 0$ can be used as reference for the case when bit $i$ of $q$ is 1.

Interestingly, the zero-one gap is different between mod_SSL and stunnel, even though both were linked against the same OpenSSL library files. stunnel exhibits a positive timing difference for most bits of $q$ that are 0, while mod_SSL has a negative difference in similar cases. Both applications have a sufficiently large zero-one gap to be considered vulnerable.

This experiment highlights the difficulty in determining the running-time of our attack against a particular SSL-enabled application. Even though both stunnel and mod_SSL use the exact same

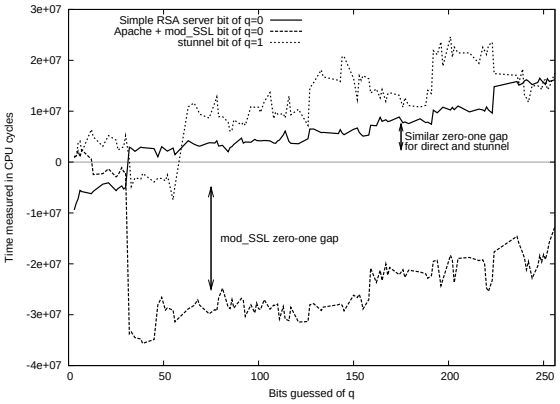



Figure 7: Attacking OpenSSL-enabled applications

OpenSSL libraries and use the same parameters for negotiating the SSL handshake, the run-time and compile-time differences result in different zero-one gaps. Additionally, the larger gap for mod_SSL implies it requires fewer total queries than stunnel to successfully attack.

# 6 Defenses

We discuss three possible defenses. The most widely accepted defense against timing attacks is to perform RSA blinding. The RSA blinding operation calculates $x = r^e g \bmod n$ before decryption, where $r$ is random, $e$ is the RSA encryption exponent, and $g$ is the ciphertext to be decrypted. $x$ is then decrypted as normal, followed by division by $r$, i.e. $x^e/r \bmod n$. Since $r$ is random, $x$ is random and timing the decryption should not reveal information about the key. Note that $r$ should be a new random number for every decryption. According to [15] the performance penalty is $2\% - 10\%$, depending upon implementation. Netscape/Mozilla's NSS library uses blinding. Blinding is available in OpenSSL, but not enabled by default and is not used by applications such as mod_SSL. We hope this paper demonstrates the necessity of enabling the timing defense.

Two other possible defenses are suggested often, but are a second choice to blinding. The first is to try and make all RSA decryptions take the same amount of time. In OpenSSL one would use only one multiplication routine and always carry out the extra reduction in Montgomery's algorithm, as proposed by Schindler in [16]. If an extra reduction is not needed, we carry out a "dummy" extra reduction and do not use the result. We believe this method is difficult to get right and results in code that is hard to maintain. Note that compilers tend to optimize away dummy operations.

Another alternative is to require all RSA computations to be quantized, i.e. always take a multiple of some predefined time quantum. Matt Blaze's quantize library [1] is an example of this approach. Note that *all* decryptions must take the maximum time of *any* decryption, otherwise, timing information can still be used to leak information about the secret key.

Currently, the preferred method for protecting against timing attacks is to use RSA blinding. The immediate drawbacks to this solution is a good source of randomness is essential to prevent attacks on the blinding factor, as well as the small performance degradation. In OpenSSL, neither drawback appears to be a significant problem.

# 7 Conclusion

We devised and implemented a timing attack against OpenSSL – a library commonly used in web servers. Our experiments show that, counter to current belief, the timing attack is effective when carried out between two machines in a local network. Similarly, the timing attack is effective between two processes on the same machine and two Virtual Machines on the same computer. We hope these results will convince designers of crypto libraries to implement defenses against timing attacks as described in the previous section.

# References


[1] Matt Blaze. Quantize wrapper library. `http://islab.oregonstate.edu/documents/People/blaze`.

[2] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. *Lecture Notes in Computer Science*, 1462, 1998.

[3] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults. *Lecture Notes in Computer Science*, 1233:37–51, 1997.

[4] D. Coppersmith. Small solutions to polynomial equations, and low exponenent RSA vulnerabilities. *Journal of Cryptology*, 10:233–260, 1997.

[5] Jean-Francois Dhem, Francois Koeune, Philippe-Alexandre Leroux, Patrick Mestre, Jean-Jacques Quisquater, and Jean-Louis Willems. A practical implementation of the timing attack. In *CARDIS*, pages 167–182, 1998.

[6] GNU. libgcrypt. `http://www.gnu.org/directory/security/libgcrypt.html`.

[7] Peter Gutmann. Cryptlib. `http://www.cs.auckland.ac.nz/~pgut001/cryptlib/`.

[8] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis: Leaking secrets. In *Crypto 99*, pages 388–397, 1999.

[9] Paul Kocher. Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. *Advances in Cryptology*, pages 104–113, 1996.

[10] Alfred Menezes, Paul Oorschot, and Scott Vanstone. *Handbook of Applied Cryptography*. CRC Press, October 1996.

[11] mod_SSL Project. mod_ssl. `http://www.modssl.org`.

[12] Peter Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[13] OpenSSL Project. Openssl. `http://www.openssl.org`.

[14] Rao, Josyula, Rohatgi, and Pankaj. Empowering side-channel attacks. Technical Report 2001/037, 2001.

[15] RSA Press Release. `http://www.otn.net/onthenet/rsaqa.htm`, 1995.

[16] Werner Schindler. A timing attack against RSA with the chinese remainder theorem. In *CHES 2000*, pages 109–124, 2000.

[17] stunnel Project. stunnel. `http://www.stunnel.org`.