Remote Timing Attacks are Practical

David Brumley
Stanford University
dbrumley@cs.stanford.edu

Dan Boneh
Stanford University
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 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 [10] designed a timing attack to expose secret keys used for RSA decryption. Until now, these attacks were only applied in the context of hardware security tokens such as smartcards [4, 10, 18]. 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 [15], 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.

Network. We successfully mounted our timing attack between two machines on our campus network. The attacking machine and the server were in different buildings with three routers and multiple switches between them. With this setup we were able to extract the SSL private key from common SSL applications such as a web server (Apache+mod_SSL) and a 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 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 NGSCB architecture (formerly Palladium). We also note that NGSCB 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 [14] (used in GNUTLS and GPG) and Cryptlib [5] do not defend against timing attacks. OpenSSL 0.9.7 implements a defense against the timing attack as an option. However, common applications such as mod_SSL, the Apache SSL module, do not enable this option and are therefore vulnerable to the attack. These examples show that timing attacks are a largely ignored vulnerability in many crypto implementations. We hope the results of this paper will help 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 [10, 18] to fail in practice. Consequently, we had to devise a new timing attack based on [18, 19, 20, 21, 22] 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 4 we discuss how these attacks apply to SSL. 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 0.9.7 server. The attack takes about two hours.

Timing attacks are related to a class of attacks called side-channel attacks. These include power analysis [9] and attacks based on electromagnetic radiation [16]. 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. Our timing attack does not depend upon the RSA padding used in SSL and TLS.

2 OpenSSL's Implementation of RSA

We begin by reviewing how OpenSSL implements RSA decryption. We only review the details needed for our attack. OpenSSL closely follows algorithms described in the Handbook of Applied Cryptography [11], 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 [10]. 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 by computing $d = e^{-1} \bmod (p-1)(q-1)$.

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. When using sliding windows a block of bits (window) of d are processed at each iteration, where as simple square-and-multiply processes only one bit of d per iteration. Sliding windows requires pre-computing a multiplication table, which takes time proportional to $2^{w-1}+1$ for a window of size w. Hence, there is an optimal window size that balances the time spent during precomputation vs. actual exponentiation. For a 1024-bit modulus OpenSSL uses a window size of five so that about five bits of the exponent d are processed in every iteration.

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 techniques to handle sliding windows exponentiation used in OpenSSL.

2.3 Montgomery Reduction

The sliding windows exponentiation algorithm performs a modular multiplication at every step. Given two integers x, y, computing $xy \bmod q$ is done by first multiplying the integers $x * y$ and then reducing the result modulo q. Later we will see each reduction also requires a few additional multiplications. 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 [13].

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 as many arithmetic operations can be implemented directly in hardware. 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: $aR*bR = cR^2$. Then, use the fast Montgomery reduction algorithm to compute $cR^2 * R^{-1} = cR \bmod q$. Note that the result $cR \bmod q$ is in Montgomery form, and thus can be directly used in subsequent Montgomery operations. At the end of the exponentiation algorithm the output is put back into standard (non-Montgomery) form by multiplying it by $R^{-1} \bmod q$. For our attack, it is equivalent to use R and $R^{-1} \bmod N$, which are public.

Hence, for the small penalty of converting the input g to Montgomery form, a large gain is achieved 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.

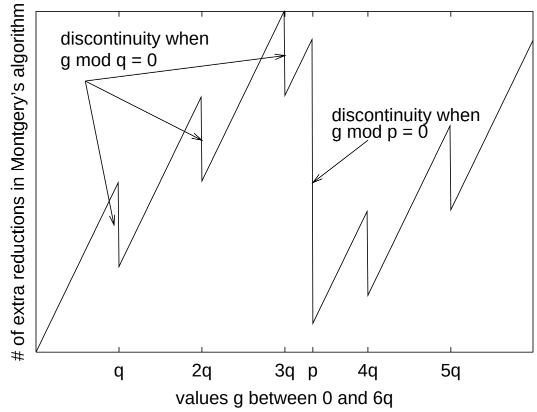


Figure 1: Number of extra reductions in a Montgomery reduction as a function (equation 1) of the input g.

The key relevant fact about a Montgomery reduction is at the end of the 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 noticed that the probability of an extra reduction during an exponentiation $g^d \bmod q$ is proportional to how close g is to q [18]. Schindler showed that the probability for an extra reduction is:

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

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.

2.4 Multiplication Routines

RSA operations, including those using Montgomery's method, must make 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. Karatsuba multiplication takes time $O(n^{\log_2 3})$ which is $O(n^{1.58})$. 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. Hence, for numbers that are approximately the same size (i.e. n is close to m) normal multiplication takes quadratic time.

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. Time measurements will reveal how frequently the operands given to the multiplication routine have the same length. We use this fact in 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. As we will see in experiment 3 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 Comparison of Timing Differences

So far we identified two algorithmic data dependencies in OpenSSL that cause time variance in 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 the factor 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 $g < q$ should be slower than decryption of $g > q$.

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. In this case, OpenSSL uses normal multiplication which is slower. In other words, decryption of $g < q$ should be faster than decryption of $g > q$ — 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 dominant 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 [3] 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 - Compute $u_g = gR^{-1} \bmod N$ and $u_{g_{hi}} = g_{hi}R^{-1} \bmod N$. This step is needed because RSA decryption with Montgomery reduction will calculate $u_g R = g$ and $u_{g_{hi}} R = g_{hi}$ to put u_g and $u_{g_{hi}}$ in Montgomery form before exponentiation during decryption.
- Step 3 We measure the time to decrypt both u_g and $u_{g_{hi}}$. Let t_1 = DecryptTime(u_g) and t_2 = DecryptTime($u_{g_{hi}}$).
- Step 4 - We calculate the difference $\Delta = |t_1 - t_2|$. If $g < q < g_{hi}$ then, by Section 2.5, the difference Δ will be "large", and bit i of q is 0. If $g < g_{hi} < q$, the difference Δ will be "small", and bit i of q is 1. We use previous Δ 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 affect 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 $\frac{\log_2 d}{2}$ 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:

$$E[\texttt{\# multiply by g}] \approx \frac{\log_2 d}{2^{w-1}(w+1)}$$

resulting in a weak 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 [23] and an Apache web server with mod_SSL [12], and an attack on trusted computing projects such as Microsoft's NGSCB (formerly 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 NGSCB. One goal of NGSCB is to provide sealed storage. Sealed storage allows an application to encrypt data to disk using keys unavailable to the user. The timing attack shows that by asking NGSCB to decrypt data in sealed storage a user may learn the secret application key. Therefore, it is essential that the secure storage mechanism provided by projects such as NGSCB defend against this timing attack.

As mentioned in the introduction, 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 show the factorization of the RSA modulus N is vulnerable. We show that a number of factors affect the efficiency of our timing attack.

Our experiments consisted of:

1. Test 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 machine architecture and 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, and directly characterize the vulnerability of OpenSSL's RSA decryption routine. The fifth experiment demonstrates our attack succeeds 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, which does not blind RSA operations by default. All tests were run under RedHat Linux 7.3 on a 2.4 GHz Pentium 4 processor with 1 GB of RAM, using gcc 2.96 (RedHat). 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.

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. As recommended in [7], we use the "cpuid" instruction to serialize the processor to prevent out-of-order execution from changing our timing measurements. Note that cpuid and rdtsc are only used by the attacking client, and that neither instruction is 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.

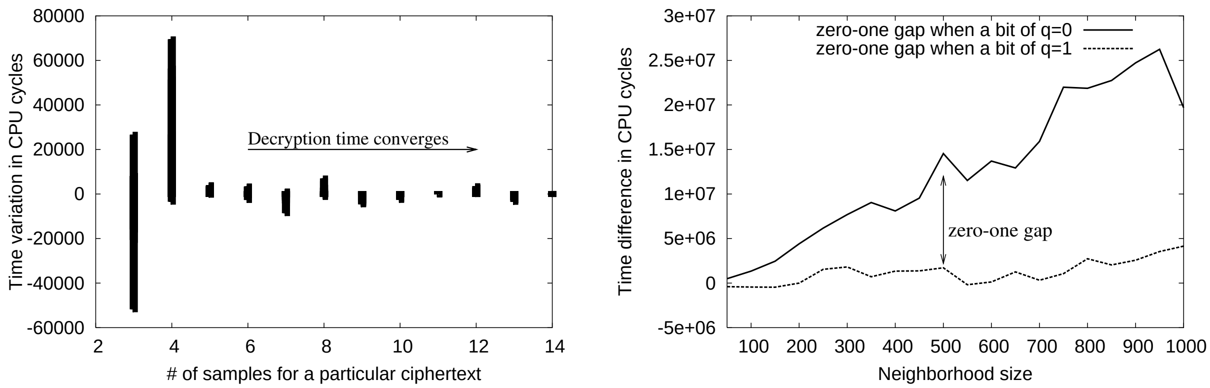
5.2 Experiment 1 - Number of Ciphertexts

This experiment explores the parameters that determine the number of queries needed to expose a single bit of an RSA factor. For any particular bit of q, the number of queries for guess g is determined by two parameters: neighborhood size and sample size.

Neighborhood size. For every bit of q we measure the decryption time for a neighborhood of values $g, g+1, g+2, ..., g+n$. We denote this neighborhood size by n.

Sample size. For each value $g + i$ in a neighborhood we sample the decryption time multiple times and compute the mean decryption time. The number of times we query on each value $g + i$ is called the sample size and is denoted by s.

The total number of queries needed to compute T_g is then $s * n$.



(a) The time variance for decrypting a particular ciphertext decreases as we increase the number of samples taken.

(b) By increasing the neighborhood size we increase the zero-one gap between a bit of q that is 0 and a bit of q that is 1.

Figure 2: Parameters that affect the number of decryption queries of g needed to guess a bit of the RSA factor.

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.

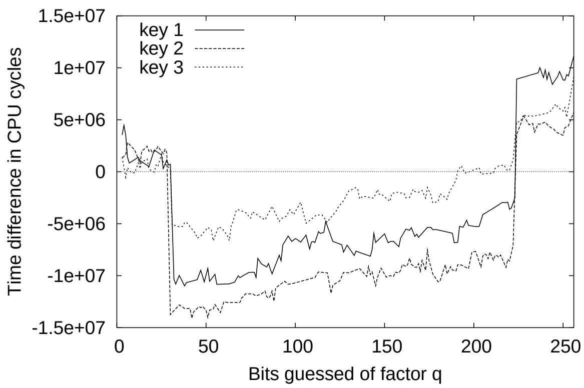
We call the gap between when a bit of q is 0 and 1 the *zero-one* gap. This gap is related to the difference $|T_g - T_{g_{hi}}|$, which we expect to be large when a bit of q is 0 and small otherwise. The larger the gap, the stronger the indicator that bit i is 0, and the smaller chance of error. Figure 2(b) shows that increasing the neighborhood size increases the size of the zero-one gap when a bit of q is 0, but is steady when a bit of q is 1.

The total number of queries to recover a factor is $2ns * \log_2 N/4$, where N is the RSA public modulus. Unless explicitly stated otherwise, we use a sample size of 7 and a neighborhood size of 400 on all subsequent experiments, resulting in 1433600 total queries. With these parameters a typical attack takes approximately 2 hours. In practice, an effective attack may need far fewer samples, as the neighborhood size can be adjusted dynamically to give a clear zero-one gap in the smallest number of queries.

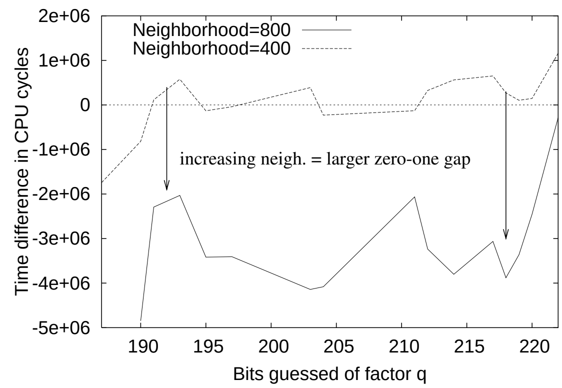
5.3 Experiment 2 - Different Keys

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. Figure 3(a) shows our results for 3 different keys. For clarity, we include only bits of q that are 0, as bits of q that are 1 are close to the x-axis. In all our figures the time difference $T_g - T_{g_{hi}}$ is the zero-one gap. When the zero-one gap for bit i is far from the x-axis we can correctly deduce that bit i is 0.

With all keys the zero-one gap is positive for about the first 32 bits due to Montgomery reductions, since both g and g_{hi} use Karatsuba multiplication. After bit 32, the difference between Karatsuba and normal multiplication dominate until overcome by the sheer size difference between $\log_2(g \bmod q) - \log_2(g_{hi} \bmod q)$. The size difference alters the zero-one gaps because as bits of q are guessed, g_{hi} becomes smaller while g remains $\approx \log_2 q$. The size difference counteracts the effects of Karatsuba vs. normal multiplication. Normally the resulting zero-one gap shift happens around multiples of 32 (224 for key 1, 191 for key 2 and 3), our machine word size. Thus, an attacker should be aware that the zero-one gap may flip signs when guessing bits that are around multiples of the machine word size.



(a) The zero-one gap $T_g - T_{g_{hi}}$ indicates that we can distinguish between bits that are 0 and 1 of the RSA factor q for 3 different randomly-generated keys. For clarity, bits of q that are 1 are omitted, as the x-axis can be used for reference for this case.



(b) When the neighborhood is 400, the zero-one gap is small for some bits in key 3, making it difficult to distinguish between the 0 and 1 bits of q. By increasing the neighborhood size to 800, the zero-one gap is increased and we can launch a successful attack.

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

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 800 to increase the zero-one gap, resulting in a strong enough indicator to mount a successful attack on bits 190-220 of q in key 3.

The results of this experiment show that the factorization of each key is exposed by our timing attack by the zero-one gap created by the difference when a bit of q is 0 or 1. The zero-one gap can be increased by increasing the neighborhood size if hard-to-guess bits are encountered.

5.4 Experiment 3 - Architecture and Compile-Time Effects

In this experiment we show how the computer architecture and common compile-time optimizations can affect the zero-one gap in our attack. Previously, we have shown how algorithmically the number of extra Montgomery reductions and whether normal or Karatsuba multiplication is used results in a timing attack. However, the exact architecture on which decryption is performed can change the zero-one gap.

To show the effect of architecture on the timing attack, we begin by showing the total number of instructions retired agrees with our algorithmic analysis of OpenSSL's decryption routines. An instruction is retired when it completes and the results are written to the destination [8]. However, programs with similar retirement counts may have different execution profiles due to different run-time factors such as branch predictions, pipeline throughput, and the L1 and L2 cache behavior.

We show that minor changes in the code can change the timing attack in two programs: "regular" and "extra-inst". Both programs time local calls to the OpenSSL decryption routine, i.e. unlike other programs presented "regular" and "extra-inst" are not network clients attacking a network server. The "extra-inst" is identical to "regular" except 6 additional nop instructions inserted before timing decryptions. The nop's only change subsequent code offsets, including those in the linked OpenSSL library.

Table 1 shows the timing attack with both programs for two bits of q. Montgomery reductions cause a positive instruction retired difference for bit 30, as expected. The difference between Karatsuba and normal multiplication cause a negative instruction retired difference for bit 32, again as expected. However, the difference $T_g - T_{g_{hi}}$ does not follow the instructions retired difference. On bit 30, there is about a 4 million extra cycles difference between the "regular" and "extra-inst" programs, even though the instruction retired count decreases. For bit 32, the change is even more pronounced: the zero-one gap changes sign between the "normal" and "extra-inst" programs while the instructions retired are similar!

	$g - g_{hi}$ retired	$T_g - T_{g_{hi}}$ cycles
"regular" bit 30	4579248 (0.009%)	6323188 (0.057%)
"extra-inst" bit 30	7641653 (0.016%)	2392299 (0.022%)
"regular" bit 32	-14275879 (-0.029%)	-5429545 (-0.049%)
"extra-inst" bit 32	-13187257 (-0.027%)	1310809 (0.012%)

Table 1: Bit 30 of q for both "regular" and "extra-inst" (which has a few additional nop's) have a positive instructions retired difference due to Montgomery reductions. Similarly, bit 32 has a negative instruction difference due to normal vs. Karatsuba multiplication. However, the addition of a few nop instructions in the "extra-instr" program changes the timing profile, most notably for bit 32. The percentages given are the difference divided by either the total of instructions retired or cycles as appropriate.

Extensive profiling using Intel's VTune [6] shows no single cause for the timing differences. However, two of the most prevalent factors were the L1 and L2 cache behavior and the number of instructions speculatively executed incorrectly. For example, while the "regular" program suffers approximately 0.139% L1 and L2 cache misses per load from memory on average, "extra-inst" has approximately 0.151% L1 and L2 cache misses per load. Additionally, the "regular" program speculatively executed about 9 million micro-operations incorrectly. Since the timing difference detected in our attack is only about 0.05% of total execution time, we expect the runtime factors to heavily affect the zero-one gap. However, under normal circumstances some zero-one gap should be present due to the input data dependencies during decryption.

The total number of decryption queries required for a successful attack also depends upon how OpenSSL is compiled. The compile-time optimizations change both the number of instructions, and how efficiently instructions are executed on the hardware. 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.

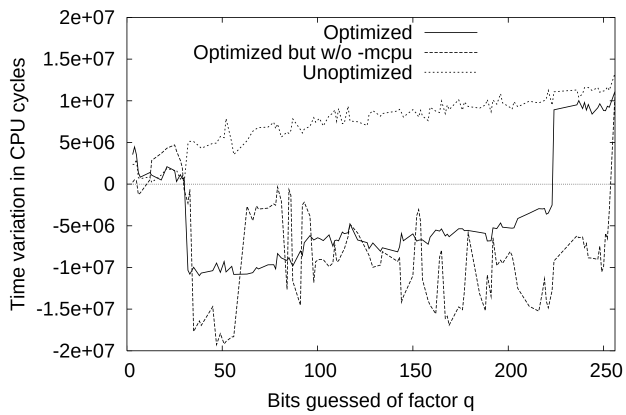


Figure 4: Different compile-time flags can shift the zero-one gap by changing the resulting code and how efficiently it can be executed.

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

Each different compile-time optimization changed the zero-one gap. Figure 4 compares the results of each test. For readability, we only show the difference $T_g - T_{g_{hi}}$ when bit i of q is 0 ($g < q < g_{hi}$). The case where bit $i = 1$ shows little variance based upon the optimizations, and the x-axis can be used for reference.

Recall we expected Montgomery reductions to dominate when guessing the first 32 bits (with a positive zero-one gap), switching to Karatsuba vs. normal multiplication (with a negative zero-one gap) thereafter. Surprisingly, the unoptimized OpenSSL is unaffected by the Karatsuba vs. normal multiplication. Another surprising difference is the zero-one gap is more erratic when the -mcpu flag is omitted.

In these tests we again made about 1.4 million decryption queries. We note that without optimizations (-g), separate tests allowed us to recover the factorization with less than 359000 queries. This number could be reduced further by dynamically reducing the neighborhood size as bits of q are learned. Also, our tests of OpenSSL 0.9.6g were similar to the results of 0.9.7, suggesting previous versions of OpenSSL are also vulnerable.

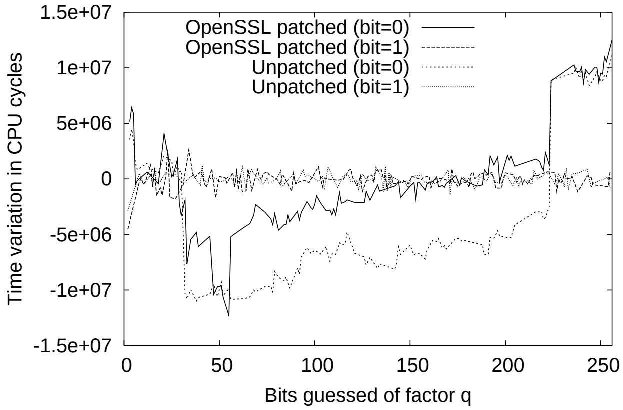


Figure 5: Minor source-based optimizations change the zero-one gap as well. As a consequence, code that doesn't appear initially vulnerable may become so as the source is patched.

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 exact execution environment. Common flags such as enabling debugging support allow our attack to recover the factors of a 1024-bit modulus in about $1/3$ million queries. We speculate that less complex architectures will be less affected by minor code changes, and have the zero-one gap as predicted by the OpenSSL algorithm analysis.

5.5 Experiment 4 - Source-based Optimizations

Source-based optimizations can also change the zero-one gap. RSA library developers may believe their code is not vulnerable to the timing attack based upon testing. However, subsequent patches may change the code profile resulting in a timing vulnerability. To show that minor source changes also affect our attack, 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 an incorrect CRT decryption from revealing the factors of the modulus [2]. By default, OpenSSL needlessly recalculates both Montgomery parameters R and $R^{-1} \bmod N$ 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.

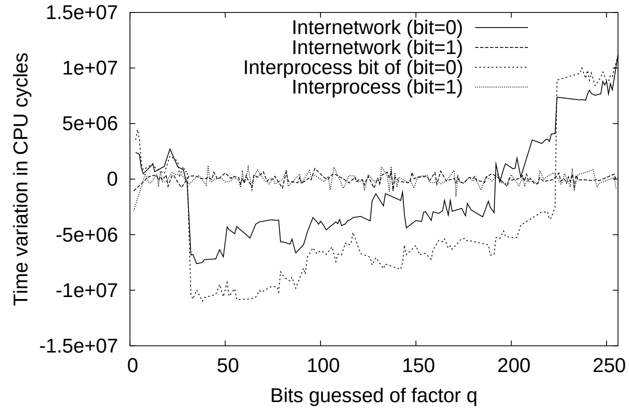
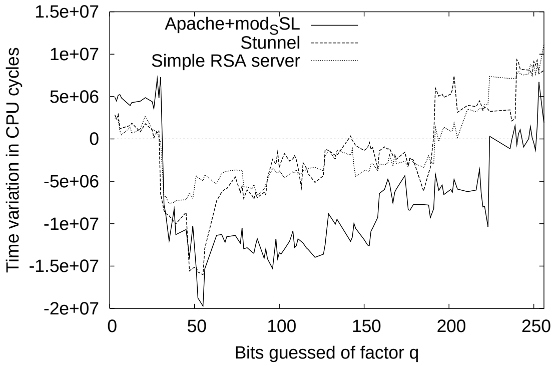


Figure 6: The timing attack succeeds over a local network. We contrast our results with the attack inter-process.

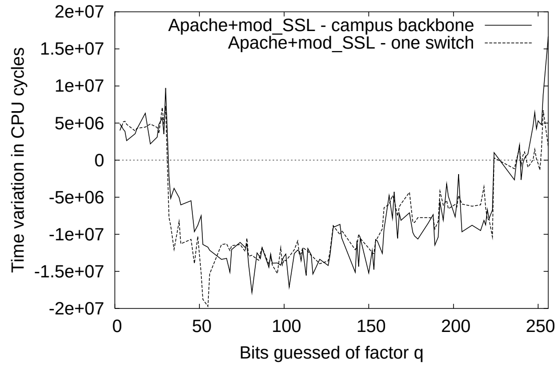
The zero-one gap is shifted because the resulting code will have a different execution profile, as discussed in the previous experiment. While our specific patch decreases the size of the zero-one gap, other patches may increase the zero-one gap. This shows the danger of assuming a specific application is not vulnerable due to timing attack tests, as even a small patch can change the run-time profile and either increase or decrease the zero-one gap. Developers should instead rely upon proper algorithmic defenses as discussed in section 6.

5.6 Experiment 5 - Interprocess vs. Local Network Attacks

To 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 seriously 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 tests a zero-one gap of approximately 1 millisecond is sufficient to receive a strong indicator, enabling a successful attack. Thus, networks with less than 1ms of variance are vulnerable.



(a) The zero-one gaps when attacking Apache+mod_SSL and stunnel separated by one switch.



(b) The zero-one gap when attacking Apache+mod_SSL separated by several routers and a network backbone.

Figure 7: Applications using OpenSSL 0.9.7 are vulnerable, even on a large network.

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 be more distinct.

5.7 Experiment 6 - Attacking SSL Applications on the Local Network

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

We begin by showing servers connected by a single switch are vulnerable to our attack. This scenario is relevant when the attacker has access to a machine near the OpenSSL-based server. Figure 7(a) shows the result of attacking stunnel and mod_SSL where the attacking client is separated by a single switch. For reference, we also include the results for a similar attack against the simple RSA decryption server from the previous experiments.

Interestingly, the zero-one gap is larger for Apache+mod_SSL than either the simple RSA decryption server or stunnel. As a result, successfully attacking Apache+mod_SSL requires fewer queries than stunnel. Both applications have a sufficiently large zero-one gap to be considered vulnerable.

To show our timing attacks can work on larger networks, we separated the attacking client from the Apache+mod_SSL server by our campus backbone. The webserver was hosted in a separate building about a half mile away, separated by three routers and a number of switches on the network backbone. Figure 7(b) shows the effectiveness of our attack against Apache+mod_SSL on this larger LAN, contrasted with our previous experiment where the attacking client and server are separated by only one switch.

This experiment highlights the difficulty in determining the minimum number of queries for a successful attack. Even though both stunnel and mod_SSL use the exact same OpenSSL libraries and use the same parameters for negotiating the SSL handshake, the run-time differences result in different zero-one gaps. More importantly, our attack works even when the attacking client and application are separated by a large network.

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 [17] 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 in versions prior to 0.9.7b. Figure 8 shows that blinding in OpenSSL 0.9.7b defeats our attack. We hope this paper demonstrates the necessity of enabling this defense.

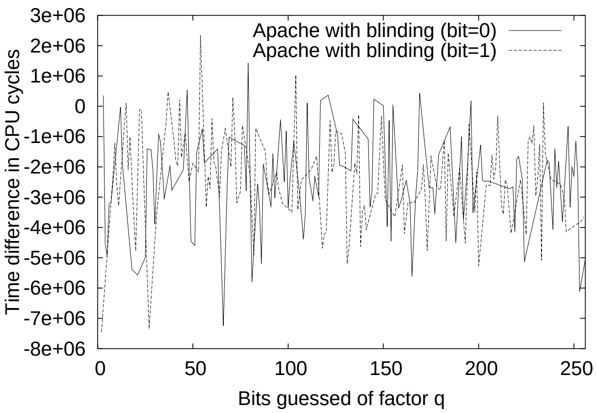


Figure 8: Our attack against Apache+mod_SSL using OpenSSL 0.9.7b is defeated because blinding is enabled by default.

Two other possible defenses are suggested often, but are a second choice to blinding. The first is to try and make all RSA decryptions not dependent upon the input ciphertext. 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 [18]. If an extra reduction is not needed, we carry out a "dummy" extra reduction and do not use the result. Karatsuba multiplication can always be used by calculating $c \bmod p_i * 2^m$, where c is the ciphertext, p_i is one of the RSA factors, and $m = \log_2 p_i - \log_2 (c \bmod p_i)$. After decryption, the result is divided by $2^{md} \bmod q$ to yield the plaintext. We believe it is harder to create and maintain code where the decryption time is not dependent upon the ciphertext. For example, since the result is never used from a dummy extra reduction during Montgomery reductions, it may inadvertently be optimized away by the compiler.

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 that a good source of randomness is needed 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 and other SSL applications. Our experiments show that, counter to current belief, the timing attack is effective when carried out between machines separated by multiple routers. Similarly, the timing attack is effective between two processes on the same machine and two Virtual Machines on the same computer. As a result of this work, several crypto libraries, including OpenSSL, now implement blinding by default as described in the previous section.

8 Acknowledgments

This material is based upon work supported in part by the National Science Foundation under Grant No. 0121481 and the Packard Foundation. We thank the reviewers, Dr. Monica Lam, Ramesh Chandra, Constantine Sapuntzakis, Wei Dai, Art Manion and CERT/CC, and Dr. Werner Schindler for their comments while preparing this paper. We also thank Nelson Bolyard, Geoff Thorpe, Ben Laurie, Dr. Stephen Henson, Richard Levitte, and the rest of the OpenSSL, mod_SSL, and stunnel development teams for their help in preparing patches to enable and use RSA blinding.

References

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

[2] 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.

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

[4] 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.

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

[6] Intel. Vtune performance analyzer for linux v1.1. `http://www.intel.com/software/products/vtune`.

[7] Intel. Using the RDTSC instruction for performance monitoring. Technical report, 1997.

[8] Intel. Ia-32 intel architecture optimization reference manual. Technical Report 248966-008, 2003.

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

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

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

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

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

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

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

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

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

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

[19] Werner Schindler. A combined timing and power attack. *Lecture Notes in Computer Science*, 2274:263–279, 2002.

[20] Werner Schindler. Optimized timing attacks against public key cryptosystems. *Statistics and Decisions*, 20:191–210, 2002.

[21] Werner Schindler, Franois Koeune, and Jean-Jacques Quisquater. Improving divide and conquer attacks against cryptosystems by better error detection/correction strategies. *Lecture Notes in Computer Science*, 2260:245–267, 2001.

[22] Werner Schindler, Franois Koeune, and Jean-Jacques Quisquater. Unleashing the full power of timing attack. Technical Report CG-2001/3, 2001.

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