An Adaptive Idle-Wait Countermeasure Against Timing Attacks on Public-Key Cryptosystems

Carlos Moreno and M. Anwar Hasan

Department of Electrical and Computer Engineering
University of Waterloo, Canada

Abstract

Successful timing attacks against public-key cryptosystems have been demonstrated in many forms, suggesting the use of a technique known as *blinding* as countermeasure to these attacks. An alternative countermeasure has been considered, though somewhat overlooked and less studied in existing literature, consisting of idle-wait to make the decryption time independent of the data. In this work, we propose and implement an optimized form of this countermeasure, making the idle-wait *adaptive*, with the goal of minimizing the performance penalty. We present both analytical and experimental results of simulations designed to evaluate our method's performance and effectiveness, and compare it against alternative countermeasures.

Keywords. Side-channel attack, timing attack, blinding, public-key cryptosystem, RSA.

1 Introduction

Side-channel attacks constitute an important consideration in the implementation of cryptographic systems, including public-key cryptosystems. Attacks in this class attempt to bypass any intrinsic mathematical security of the cryptosystem; instead, they attack the *implementation* of the cryptosystem directly, exploiting observable side-effects of the required operations that may exhibit certain correlation with the secret data. Such correlation constitutes leaked information that may be used to reconstruct the secret data.

Some of the typical side-channel attacks are based on electromagnetic emissions, power consumption, and timing of the operations. In this study, we focus our attention on timing attacks on public-key cryptosystems.

Successful timing attacks against public-key cryptosystems have been demonstrated in many forms [1], [3], [4]. These attacks usually require large amounts of measurements with data, controlled by the attacker, that affects the timing parameters in some way.

The usual countermeasure is a technique known as *blinding*; the actual decryption operation is done on a "randomized" version of the provided ciphertext, thus removing any control over the data from the attacker. This randomization is done in a way that can be reversed after the decryption is completed. This blinding operation, however, has a small performance penalty.

An alternative countermeasure consists of making the time of decryption operations independent of the data through a delay in the form of an *idle wait* after the decryption operation has been completed. Though this technique has been suggested in the existing literature, it has not been as well studied as other countermeasures. From the point of view of performance, this idle-wait technique has a subtle potential advantage, due to the fact that once the CPU is no longer executing a decryption operation, it is available for other tasks to be executed, since the idle wait allows for concurrent tasks to proceed [9]. It also has a potential advantage in applications where power consumption is critical, such as battery-powered devices. This is due to the fact that the performance penalty in this case does not come in term of actual processing, and thus power consumption is reduced, compared to other countermeasures.

In this study, we propose and implement an optimized form of this countermeasure, making the idle-wait *adaptive*, with the goal of minimizing the performance penalty. As part of our study, we evaluate, through simulations, the effectiveness of our proposed technique in terms of performance penalty, verifying that our solution exhibits a smaller performance penalty than the usual blinding countermeasure. Though the countermeasure is in principle valid for most public-key cryptosystems, our experimental setup is focused on RSA decryption operations.

The remaining of this report proceeds as follows: we start by presenting a brief overview of background information, including timing attacks and known countermeasures; we then describe our proposed idle-wait countermeasure and its implementation, presenting some analytical evidence of its effectiveness; finally, we describe our experimental setup and present our results, followed by a brief discussion and suggested work.

2 Timing Attacks on RSA Decryption Operations

The main idea behind side-channel attacks on the timing of RSA decryption operations is the relationship between the data involved in the operations and the variations in the execution time. This data includes both the cryptosystem's parameters — in particular, the secret parameters — and the ciphertext being decrypted.

Depending on the implementation, the cryptosystem's secret parameters may include the decryption exponent d, or the modulus prime factorization, p and q. This depends mainly on the types of optimizations used — e.g., straightforward square-and-multiply vs. Montgomery modular multiplication and the Chinese Remainder Theorem (CRT) to execute two exponentiations modulo each of the factors [7]. It will be clear that the idea presented in this study applies to all of those cases, even if the demonstration presented is for OpenSSL [13].

In most cases, timing attacks use statistical processing on a large number of decryption operations with data controlled by the attacker; this is the case due to two main reasons: the attacker usually has no access to timing measurements of the intermediate operations, and the total amount of time is approximately the same for every decryption. By using statistical processing, the attacker can accurately measure the small variations caused by the data (the ciphertexts). A second reason is that the use of statistical techniques makes

it possible for the attack to get around measurement errors and various random delays contributing to the timing of the operation that are beyond the attacker's control.

Specific attacks are designed for different implementations of the decryption operation; for the general case where modular exponentiations are implemented as a straightforward square-and-multiply, Paul Kocher [1] describes an approach based on guessing one bit at a time, and measuring the variance in the difference between the times measured for the actual decryption operation and the operation "mimicked" by the attacker; when the bit is guessed correctly, we have one additional iteration where both systems are working with the same data, and thus, the measured variance is lower, allowing the attacker to validate the guess for each bit of the decryption exponent.

Schindler [3] presented a more specific timing attack, applicable to implementations that use Montgomery exponentiations combined with the Chinese Remainder Theorem optimization for RSA decryptions. The attack exploits a measurable difference in the decryption time when the ciphertext is close to a multiple of either p or q. This attack was later refined and demonstrated by Brumley and Boneh [4], presenting a successful timing attack on OpenSSL [13], a real-world cryptographic system that is widely used for Internet applications — they showed that these attacks can be effective even when applied to a remote system over the Internet.

An important common theme in these attacks is the ability on the part of the attacker to measure statistical parameters from the collected measurements of the decryption time over a somewhat long period of time, with data controlled by the attacker.

2.1 Standard Countermeasure

The usually recommended countermeasure is a technique called *blinding*, in which the ciphertext is "randomized" before being decrypted. In the case of RSA, with ciphertext y (corresponding to a plaintext x), this operation is done by choosing a random number r_b and obtaining a randomized ciphertext $y' = y \cdot r_b^e$. This y' is then decrypted — we observe that any timing characteristics are now a function of y', over which the attacker has no control or even knowledge.

When decrypting y', we obtain $x' = (y')^d = (y \cdot r_b^e)^d = y^d \cdot r_b^{e \cdot d} = x \cdot r_b$. Since we chose r_b, we compute its inverse mod m so that we can obtain x (the actual, correct result of the decryption for the supplied ciphertext).

As a slight optimization, it is suggested that the "random" values be obtained by squaring the previous ones modulo m — from the point of view of the attacker, this can not be distinguished from using true random values. Thus, r_b, r_b^e, and r_b^{-1} are precomputed, and with every decryption operation, simply square r_b^e and r_b^{-1} — the resulting squares still have the property that when r_b^e is decrypted, it produces the inverse of the other value. Even with this optimization, this countermeasure has a performance penalty.

2.2 Idle-Wait Countermeasures

An alternative countermeasure consists of making the decryption time constant through an *idle wait* ([4] suggests this as a possibility, though not the preferred defense). The

idea is that after completing the decryption operation, the system would not hand out the result immediately (which would reveal the duration of the decryption); instead, the system would wait an additional amount of time, so that the *total* observed decryption time is fixed, regardless of the ciphertext being decrypted.

This idle-wait solution has a subtle additional advantage, overlooked in previous studies: being *idle* wait, as soon as the actual processing part of the decryption is finished, the processor is now free, and if the system has other concurrent tasks that require use of the CPU, they can proceed. This is clearly not the case when using blinding, in which the performance penalty comes in terms of additional actual processing that prevents other tasks from proceeding. Even in cases where multitasking is not an important factor, the fact that idle-wait countermeasures involve a lower amount of actual processing still represent an advantage in terms of power consumption, which could be an important factor on battery-powered devices.

3 Adaptive Idle-Wait Countermeasure

In principle, for an idle-wait countermeasure to be effective, every decryption time must be extended (through the idle-wait) so that the total time exceeds (or at least matches) the decryption time for any other ciphertext. That is, if T_{D_i} denotes the decryption time for ciphertext i, and T_{W_i} denotes the amount of idle-wait applied after decryption of ciphertext i, then it should hold that

$$T_{D_i} + T_{W_i} \geqslant T_{D_j} \quad \forall\, i, j$$

Clearly, the use of idle-wait introduces a performance penalty (putting aside the multitasking aspect, as mentioned in §2.2). However, as this study will show, the performance penalty of idle-wait countermeasures may be comparable and even smaller than that of blinding; not necessarily for all implementations, but we show that this is the case for OpenSSL's RSA implementation.

Additional performance gain can be obtained through an optimized version of this countermeasure. The basic idea for the optimized countermeasure is to try to keep the extra delay (the idle wait) to a minimum, while still hiding any variance or useful patterns in the execution time.

Let T denote the target execution time (i.e., the total observed time, including the decryption time and the idle-wait time). The intuition is that it suffices that the target execution time T be greater than the execution time for *most* ciphertexts. Thus, we can make the parameter T *adaptive* with respect to the observed execution times of the ciphertexts, making it converge to a target value, as a function of the collected statistics of the measured decryption times. After the system has been operating for a while, this parameter will remain virtually constant. This value is specified as a target percentile, which can be an adjustable configuration parameter. This approach is described in the next section.

3.1 Decryption Time Controlled by Target Percentile

We now explore the alternative of making the parameter T converge to a particular percentile of the decryption time. For example, we could require that T corresponds to percentile 99, such that only 1% of ciphertexts produce a decryption time that exceeds the value of T. In other words, we set the value of T such that $F(T) = 0.99$, where $F(\cdot)$ is the Cumulative Distribution Function (CDF) of the random variable representing the actual decryption time.

Since we do not have a priori knowledge or an analytical description of the CDF of the decryption time, we use the decryption times of the requested operations to make the value of T converge to the specified target. Many approaches can be used for this, including the "brute force" solution of storing all decryption times in an ordered sequence, to choose the value T corresponding to the given percentile. This, of course, would be unacceptably inefficient in most cases.

We now present our proposed approach — a somewhat heuristic method which we believe is appropriate for this scenario, given that it is simple to implement and efficient; our results show that the method converges rather rapidly to the target percentile.

The idea is loosely based on the Newton-Raphson method for solving single-variable equations [12]; the equation in our case is $F(T) = P$, where we are trying to solve for the unknown T given the target percentile P [1]. Unlike in the Newton-Raphson method, we can not compute the value of the function or its derivative; instead, we use the numerical approximations given by the statistics collected from the decryption times. In particular, we count the total number of decryptions, and count the number of instances in which the decryption time was below our current approximation of T, denoted T_k (value of T at iteration k). This gives us an approximation for the value of $F(T_k)$. To approximate the derivative, we use two additional thresholds, closely surrounding the value of T (for example, if we have a target percentile of 99, then we could use thresholds corresponding to percentiles 98.5 and 99.5). These thresholds are denoted H_k (*High* threshold value at iteration k) and L_k (*Low* threshold value at iteration k), and we count the number of instances in which the decryption time was below each of these thresholds, to obtain approximations for $F(L_k)$ and $F(H_k)$. The approximation for the derivative at T_k is given by the slope of the straight line between these two surrounding thresholds, with which we obtain our iterative update formula for T:

$$T_{k+1} \;=\; T_k + \frac{(F_T - \hat{F}(T_k))(H_k - L_k)}{\hat{F}(H_k) - \hat{F}(L_k)} \tag{1}$$

where F_T denotes the target percentile (e.g., 0.995), and $\hat{F}$ denotes the approximation for F (given the count of instances for which the decryption time has been below the given argument).

Since the thresholds H and L are also the values corresponding to given percentiles, we also work with approximations for those, and thus, we need to update them as well. The approximation for the derivative at those values is given by the straight line going from each of those points and the point at T_k, with which we obtain the iterative update formulas for

[1] For simplicity, whenever a percentile is needed for formulas or derivations, we use a CDF value between 0 and 1 as a percentile — as opposed to a value between 0 and 100

these thresholds:

$$L_{k+1} = L_k + \frac{(F_{T_L} - \hat{F}(L_k))(T_k - L_k)}{\hat{F}(T_k) - \hat{F}(L_k)} \quad (2)$$

$$H_{k+1} = H_k + \frac{(F_{T_H} - \hat{F}(H_k))(H_k - T_k)}{\hat{F}(H_k) - \hat{F}(T_k)} \quad (3)$$

Like in the Newton-Raphson method, we require an initial estimate not far from the real solution to ensure convergence [12]. In our case, this estimate can be easily obtained by storing all decryption times for an initial sequence of operations; by collecting the first, say, 1000 or 10000 values of decryption times, we can sort them and determine the required percentile approximations. Alternatively, to avoid any storage overhead, one could assume a normal distribution and estimate the average from an initial sequence of values (simply accumulate and count, to avoid using any storage); then, with the average, estimate the variance from a second sequence of values (again, accumulating and counting, avoiding storage). From these two parameters, we obtain an approximation for any required target percentile [11].

3.2 Resistance to Any Possible Timing Attacks

Clearly, if the target percentile is set to 100, then the countermeasure will defeat *any* timing attacks — known or otherwise. This follows directly from the fact that timing measurements in such case only reveal a constant value added to random measurement noise, both of which are independent of the ciphertext and the decryption key. As our results show, a setup using percentile 100 is feasible at a performance penalty comparable to that incurred by blinding in OpenSSL. This result validates the idea of idle-wait as countermeasure to timing attacks, and also provides a conservative setup for our proposed method.

Better performance can be obtained if we want to defend against specific timing attacks that are known, and for which a given percentile could be sufficient to hide all the timing information that the specific attack requires. We will show an example of this scenario (§4.3), in which we experimentally verify that our method, with a percentile below 100, defeats the attack presented in [4].

3.3 Effect on Attacks Based on Mutual Information Analysis

In this section, we discuss the effect of our countermeasure on a broad class of attacks, based on Mutual Information Analysis [5], [6]. We will evaluate the decrease of mutual information between the data producing the leakage and the measurement taken through the side channel (decryption time, in this case) in the presence of our countermeasure.

Let X denote the measurement taken by the attacker with no countermeasure present, and let Y denote the data producing the leakage to the side-channel; the mutual information is given by

$$I(X;Y) = H(X) - H(X \mid Y)$$

where $H(\cdot)$ denotes the entropy of the given variable [8].

If we assume both X and $X \mid Y$ to follow normal distributions, we can obtain the associated entropies. Let $X \sim \mathcal{N}(\mu, \sigma^2)$, with probability density function denoted $\Phi(x)$. Then,

$$\begin{aligned} H(X) &= -\int_{-\infty}^{\infty} \Phi(x) \ln \Phi(x)\, dx \\ &= -\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}} \ln\left(\frac{1}{\sqrt{2\pi\sigma^2}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}}\right) dx \\ &= \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}} \left(\frac{1}{2}\ln\left(2\pi\sigma^2\right) + \frac{(x-\mu)^2}{2\sigma^2}\right) dx \\ &= \frac{\ln\left(2\pi\sigma^2\right)+1}{2} \end{aligned} \tag{4}$$

Let X_p denote the measurement with the countermeasure present at percentile p. We evaluate its entropy; in this case, the integral is evaluated from T to ∞, where T is the threshold corresponding to percentile p, and thus $\int_T^{\infty} \Phi(x)\, dx \;=\; 1-p$

$$\begin{aligned} H(X_p) &= -\int_T^{\infty} \Phi(x) \ln \Phi(x)\, dx \;= \int_T^{\infty} \Phi(x) \left(\frac{1}{2}\ln\left(2\pi\sigma^2\right) + \frac{(x-\mu)^2}{2\sigma^2}\right) dx \\ &= \frac{\ln\left(2\pi\sigma^2\right)}{2} \int_T^{\infty} \Phi(x)\, dx \;+\; \frac{1}{2\sigma^2}\int_T^{\infty} (x-\mu)^2 \Phi(x)\, dx \\ &= \frac{\ln\left(2\pi\sigma^2\right)}{2}(1-p) \;+\; \frac{1}{2\sigma^2}\int_T^{\infty} (x-\mu)^2 \Phi(x)\, dx \end{aligned} \tag{5}$$

We apply integration by parts to the integral in Eq. (5), with $f(x) = (x-\mu)$ and $g'(x) = (x-\mu)\Phi(x) \;\Rightarrow\; g(x) = -\sigma^2\Phi(x)$, to obtain

$$\begin{aligned} \int_T^{\infty} (x-\mu)^2 \Phi(x)\, dx \;&=\; -\sigma^2(x-\mu)\Phi(x)\Big|_T^{\infty} + \;\sigma^2 \int_T^{\infty} \Phi(x) dx \\ &=\; \sigma^2(T-\mu)\Phi(T) \;+\; \sigma^2(1-p) \end{aligned} \tag{6}$$

Combining equations (4), (5), and (6), we finally obtain

$$\begin{aligned} H(X_p) &= \frac{\ln\left(2\pi\sigma^2\right)}{2}(1-p) + \frac{1}{2}(1-p) + \frac{1}{2}(T-\mu)\Phi(T) \\ &= (1-p)H(X) \;+\; \underbrace{\frac{1}{2}(T-\mu)\Phi(T)}_{=\,O(Te^{-T^2})} \;\approx\; (1-p)H(X) \end{aligned} \tag{7}$$

Following a similar analysis, we obtain $H(X_p \mid Y) \;\approx (1-p)H(X \mid Y)$; even though the threshold for the given percentile is not necessarily the same, we expect it to be approximately the same in the general case [2]. This means that

$$I(X_p; Y) \;\approx\; (1-p)H(X) - (1-p)H(X \mid Y) \;=\; (1-p)I(X;Y)$$

[2]Otherwise, we would be talking about a much more specific attack, and thus this analysis does not apply; for such cases, the countermeasure can still be used, adjusting the percentile to the appropriate level (see §4.3 for an example).

That is, in the general case, the countermeasure with target percentile p leads to a reduction of the mutual information by a factor close to $1 - p$. We recall that p is in principle chosen to be a value close to 1, and thus $1 - p$ is a small value. This means that we can expect any attacks based on Mutual Information Analysis to be slowed down by a reasonably large factor for a typical value of p.

3.4 A Note on Implementing Idle-Wait Countermeasures

A naively implemented idle-wait countermeasure could be vulnerable to attacks that could entirely bypass the idle-wait. Indeed, as mentioned in §2.2, as soon as a decryption is completed, the processor is now free, and thus available for other (concurrent) tasks to proceed; if these tasks involve decryption operations requested concurrently by the attacker, then it would be possible to measure the time from the beginning of one operation to the beginning of the next operation, revealing the *actual processing* time, and thus entirely bypassing the idle-wait.

Figure 1 demonstrates this aspect — solid lines denote an active decryption operation (i.e., processing is taking place); dashed lines indicate an idle wait, and crosses indicate that the decryption operation is ready to begin, but can not yet, since the processor is being used by another concurrent operation.

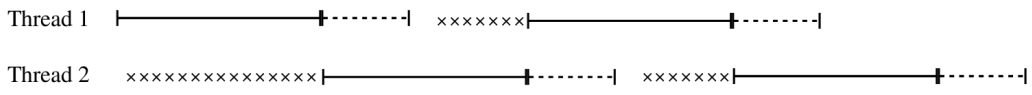


Figure 1: Throughput of Concurrent Decryption Operations.

We clearly see that, as long as the system is flooded with requests, so that at all times there is one decryption operation taking place (assuming a single-processor), then the sum of the frequencies of operations measured at each thread of execution is the frequency of execution of the operations, or throughput. Given N_T threads of concurrent execution, let f be the total frequency or throughput, N the total number of decryptions executed, f_i the frequency of decryption executions at thread i, N_i the number of decryptions completed by thread i, and T_D the time it takes for all threads to complete. Then

$$f \;=\; \frac{N}{T_D} \;=\; \frac{1}{T_D}\sum_{i=1}^{N_T} N_i \;=\; \sum_{i=1}^{N_T} \frac{N_i}{T_D} \;=\; \sum_{i=1}^{N_T} f_i \tag{8}$$

This allows us to determine the decryption time t (the actual time of execution, not including the idle-wait stage) given the measurements t_i for each thread i (notice that t_i does include the decryption time and the idle-wait):

$$t \;=\; \frac{1}{f} = \frac{1}{\sum_{i=1}^{N_T} f_i} \;=\; \frac{1}{\sum_{i=1}^{N_T} \frac{1}{t_i}} \tag{9}$$

We also see that if the idle wait is long, more than two threads would be needed; specifically, the minimum number of threads is given by

$$\#Threads \;=\; \left\lceil \frac{T_P(x) + T_I(x)}{T_P(x)} \right\rceil \tag{10}$$

where $T_P(x)$ is the processing time (the time that it takes to complete the actual operation), and $T_I(x)$ is the idle wait time. It is straightforward for the attacker to determine this figure: an estimate of the average processing time is done locally; and $T(x) = T_P(x) + T_I(x)$ is measured by requesting decryption operations using a single thread of execution.

All of the above applies to multi-core processors as well — the minimum number of threads is now minimum number *per processor core*. It is also straightforward to see that if more threads are used, the frequency of operations observed on each thread will be lower, but the sum of the frequencies will still be equal to the frequency of execution of operations, or throughput.

Naturally, the attack in the presence of a poorly implemented idle-wait countermeasure is slower than the original attack, since the threads scheduling introduces additional randomness in the timing of the operations, and thus, additional measurement noise. Still, it is clear that the countermeasure should be considered ineffective if it is vulnerable to this attack on the throughput.

3.4.1 Correct Implementation of Idle-Wait Countermeasures

The vulnerability described in this section can be easily avoided; a correct implementation of an idle-wait countermeasure (adaptive or otherwise) should prevent any decryption from starting as long as any other decryption — *including its idle-wait phase* — is still in progress. This means that the throughput of (concurrent) operations is being forced to be equal to the decryption time as measured directly (i.e., including the idle-wait), which defeats the timing attack if properly adjusted.

To this end, a *mutex* or some suitable synchronization mechanism could be used [9]. An interesting aspect is that this can be done while still maintaining the advantage of leaving the processor free for other concurrent tasks to proceed [9].

4 Simulation Setup and Experimental Results

In this section, we present and discuss the experimental part of our study. The results are based exclusively on simulations. The reason for this is that the idle-wait timer resolution required for our proposed method is higher than currently available for typical software implementations; and a hardware implementation is somewhat overkill for the purpose of evaluating the effectiveness of our method.

4.1 Setup

The basic idea for the simulations is to take timing measurements by running the actual decryption operation and then updating the value of T, using the technique described in §3.1. We simulate the decryption followed by idle-wait, and measure the performance penalty of our proposed method. To this end, we produce a sequence of randomly generated values for the ciphertext, and measure the actual processing time for those.

The decryption operation is done through invocation of the appropriate OpenSSL API function call, with blinding disabled [13], using `/dev/urandom` as the source of randomness for the ciphertexts [10]. With this actual processing time, the simulation can now determine the amount of idle-wait necessary, and with this, the average overhead is obtained.

To optimize the simulations, the processing was split into two independent programs; one program "profiles" the execution speed — generate random ciphertexts, measure the execution time of the decryption, and store these execution times in data files. A second program now uses the stored data, thus avoiding redundant invocations of decryptions, and optimizing the process.

4.2 Results

The test platform was an AMD Quad-Core Phenom Processor, 64-bit at 2.5GHz, running Ubuntu Linux 8.04, with gcc/g++ 4.2.4, and OpenSSL 0.9.8g. For the profiling, all graphical interface and networking was shut down, to avoid disrupting the measurements. For the simulation part, this was not critical, as the processing time had been already measured, and the program simply reads the values from a data file. The simulations were done with a set of 1,000,000 (one million) values of ciphertext, with decryptions done with a 1024 bits key, and the measurements done using the CPU clock cycle counter (sub-nanosecond resolution).

4.2.1 Performance Penalty

The measured blinding overhead for OpenSSL, using 1024-bit keys, is 3.1%. This is consistent with the reported range of 2 to 10% [4]. For the performance penalty of our proposed method, we adjusted the target percentile between 99.5 and 100 [3]. Figure 2 shows the results for all the simulations.

4.3 Resistance to Timing Attacks

Figure 2 shows a crucial piece of evidence in favor of our proposed method regarding resistance to timing attacks, as discussed in §3.2: for percentile 100, the performance penalty is only 3.45%; as already discussed, setting the target percentile at 100 defeats any possible

[3] For percentile 100, a different method to update T was used, since the general iterative method would fail; also, obtaining the value for percentile 100 is much simpler than for the general case.

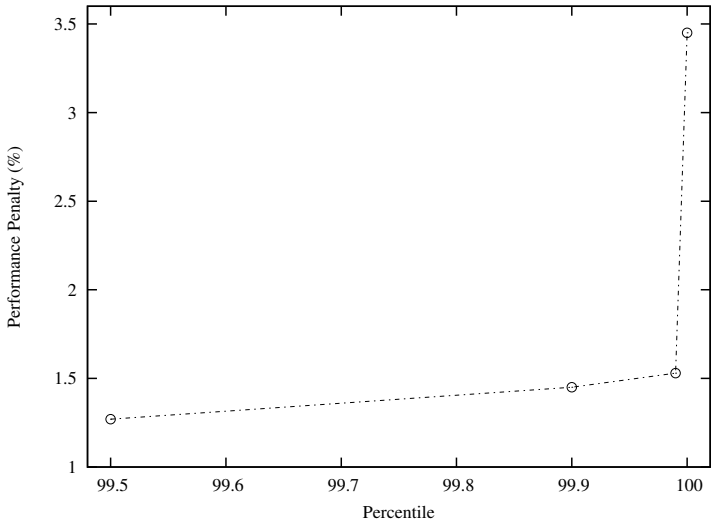


Figure 2: Performance Penalty (in %) of Our Proposed Method.

attack based on measurement of time, since the observed decryption time would be independent of the ciphertext and the decryption key. The corresponding performance penalty is within the lower half of the reported range, and barely above the figure that we measured for our test platform.

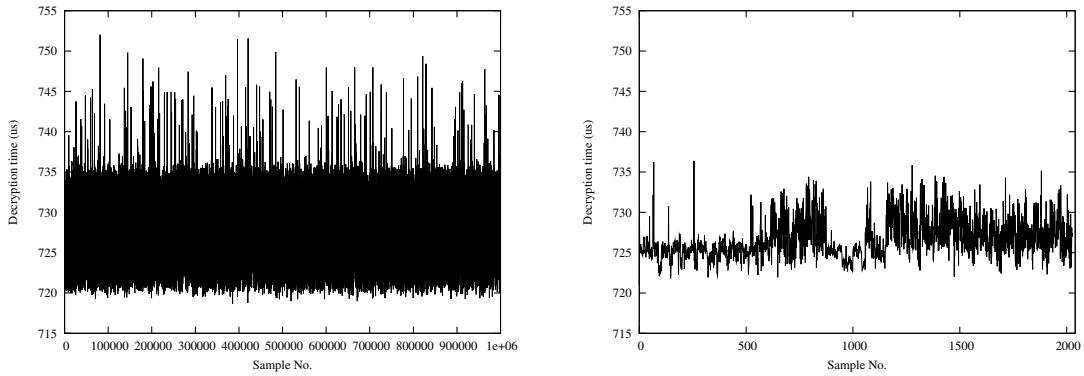
It is important, however, to notice that this is the case for RSA, where the decryption exponent is fixed; since the performance penalty is clearly related to the variance of the decryption time, it is to be expected that this figure will be higher for techniques where the decryption exponent is variable, such as Diffie-Hellman and ElGamal. We did not include these measurements in our study.

4.3.1 Case-Study: Resistance Against a Concrete Attack

As additional evidence in favor of our method, we evaluated the resistance against a known and effective timing attack, with the goal of demonstrating that additional efficiency can be obtained if we need to defend against known attacks only — not an unreasonable design criterion, given that the method can be easily readjusted, should new, more effective attacks were discovered after it was deployed.

The attack is that presented in [4]; we simulated the attack by generating all the ciphertexts that would be required (see [4] for details); since we only want to verify that all the decryption times fall within a certain percentile, we made use of the actual values of p and q, instead of executing the attack to guess the bits — clearly, a successful attack would need to use precisely these ciphertexts, as any incorrectly guessed bit would cause the attack to be ineffective for all the following bits. Thus, it makes sense and it is fair to make use of the known factorization $\{p, q\}$, even if a real attack would not have such information.

We verified that the attack is successfully defeated with a target percentile of 99.99 (with a corresponding performance penalty of 1.53%); with these settings, we get a value of $T = 736.4\mu s$, which covers the decryption times for the attack ciphertexts. Figure 3 shows



(a) Random Ciphertexts

(b) Ciphertexts for Attack on OpenSSL

Figure 3: Decryption Times – OpenSSL API without Blinding.

the decryption times for the profiling data (randomly selected ciphertexts) and for the attack ciphertexts.

Even though strictly speaking, we need to cover *all* ciphertexts to guarantee that we defeat the attack, in a practical sense, the attack would be defeated with a percentile of 99.9, with a corresponding performance penalty of 1.45%; it is not clearly visible in figure 3, but only two ciphertexts exceed percentile 99.9. It is reasonable to assume that an attack like this one, statistical in nature, would be defeated if only a negligible fraction of the required measurements survives the barrier imposed by the countermeasure.

5 Discussion and Suggested Work

The results from our experimental setup confirm the validity and show important aspects of the proposed technique and its implementation. We verified the technique's resistance against known attacks, and we see that it does offer good performance, in that it exhibits a performance penalty below that of the blinding countermeasure for a reasonably high security level. In the highest security setting, where we guarantee that any possible timing attack would be defeated, the performance penalty is comparable to that of blinding.

We emphasize again that even for comparable performance penalties, our method has an additional advantage, at least for software implementations: during the idle-wait phase, the processor is available for other tasks to proceed; this is not the case for blinding, in which the performance penalty is given in terms of actual processing that prevents other concurrent tasks from proceeding. This could be an important advantage on systems where multi-tasking is a requirement for the server where the cryptosystem is operating. Though blinding could have an advantage in cases where other side-channel attacks such as Power Analysis [2] are a threat, for many situations, remote timing attacks may be the only plausible side-channel attack, and thus our countermeasure would be perfectly suitable.

On a more theoretical side, we presented analytical evidence that the countermeasure should be effective against attacks based on Mutual Information Analysis; we showed that any such attacks would be slowed down by a reasonably high factor at a reasonable performance

penalty. These types of attacks would of course be completely defeated if we use percentile 100, in which case the mutual information would be reduced to *zero*.

Perhaps one important aspect that could require improvement relates to the practical issue of availability of idle-wait facilities with the required accuracy for software implementations; we observe that there is no fundamental reason why these can not be available; it is simply a matter that at the present time, they are not for the common software platforms. It is perhaps worth noting that a busy-wait countermeasure would not be unreasonable for a software implementation, which would address this issue; we could use the CPU cycle counter on modern processors — with sub-nanosecond resolution — as the basis for a busy-wait loop after the decryption operation; we sacrifice the benefit of the idle-wait in terms of multitasking performance, but we do keep in mind that the performance penalty is still below the performance penalty with blinding (or comparable, for the highest security setting).

Furthermore, if software implementations used currently available idle-wait facilities — which have lower resolution than required for our method — we notice that the security level would be unaffected, since the measured time is now the required time plus a random value that is uncorrelated to the amount of idle-wait that we introduce. Performance could be negatively affected, depending on how these lower-accuracy idle-wait facilities operate; but again, this would be compensated by the multi-tasking aspect of idle-wait countermeasures.

6 Conclusions

In this work, an alternative countermeasure against timing attacks on public-key cryptosystems, namely adaptive idle-wait, has been presented and evaluated. Given the results of our simulations, we can conclude that the method may be more effective, in terms of performance, than blinding solutions. The benefit is twofold, given that the performance penalty in our case comes in terms of idle wait, and thus the processor is available for other tasks to proceed. We also highlighted the additional advantage that with less actual processing required, the system is more efficient in terms of power consumption — a potentially important issue for, e.g., battery-powered devices.

The method was also shown to be effective in terms of security, in that it was verified that the countermeasure could defeat any possible timing attack, with a performance penalty comparable to that of blinding; it was also shown that the countermeasure defeats a known timing attack with a performance penalty considerably below that of blinding.

For software implementations, we must also consider the issue that standard software platforms do not feature idle-wait or timer facilities with the required accuracy. However, there is nothing that fundamentally prevents our method to be implemented in software. It is simply a matter of the required tools not being currently available in standard software platforms. Timers and idle-wait facilities with high enough accuracy could become mainstream in standard software platforms in the near future.

Acknowledgements

The first author would like to acknowledge his fruitful discussions with Dr. Ian Goldberg, who contributed with valuable ideas helping to give a better direction to this study. We would also like to thank Dr. Alfred Menezes for valuable feedback and comments on earlier versions of the manuscript.

This work was supported in part through an NSERC grant awarded to Dr. Hasan.

References

[1] Paul C. Kocher: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. Advances in Cryptology (1996)

[2] Paul Kocher, Joshua Jaffe, and Benjamin Jun: Differential Power Analysis. Advances in Cryptology (1999)

[3] Werner Schindler: A Timing Attack Against RSA with the Chinese Remainder Theorem. Workshop on Cryptographic Hardware and Embedded Systems (2000)

[4] David Brumley, Dan Boneh: Remote Timing Attacks are Practical. 12th Usenix Security Symposium (2003)

[5] Benedikt Gierlichs, Lejla Batina, Pim Tuyls: Mutual Information Analysis. CHES 2008 (2008)

[6] Nicolas Veyrat-Charvillon, François-Xavier Standaert: Mutual Information Analysis: How, When and Why? CHES 2009 (2009)

[7] Alfred Menezes, Paul Van Oorschot, Scott Vanstone: Handbook of Applied Cryptography. CRC Press (2001)

[8] T. M. Cover and Joy A. Thomas: Elements of Information Theory, Second Edition. Wiley-Interscience (2006)

[9] David R. Butenhof: Programming with POSIX Threads. Addison-Wesley (1997)

[10] John Viega, Gary McGraw: Building Secure Software. Addison-Wesley (2002)

[11] Athanasios Papoulis, S. Unnikrishna Pillai: Probability, Random Variables and Stochastic Processes, Fourth Edition. McGraw-Hill (2002)

[12] W. Press, S. Teukolsky, W. Vetterling, B. Flannery: Numerical Recipes in C, Second Edition. Cambridge University Press (1992)

[13] OpenSSL. `http://www.openssl.org` online documentation.