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

Carlos Moreno

Outline:

- Overview of Timing Attacks
- Blinding (standard countermeasure)
- Idle-Wait Countermeasures
- Adaptive Idle-Wait Countermeasure
- Experimental Results
- Conclusions

Timing Attacks:

- Use variations in the measured decryption time
  - Overall time is approximately constant; only small variations due to data-dependent optimizations are exploitable.
  - Make use of relationship between the secret parameters, the data being decrypted (which the attacker controls), and decryption time (which the attacker measures)

Example – Attack on OpenSSL

- OpenSSL uses Montgomery multiplication and CRT optimization — exponentiations are done mod p and mod q:
  - Montgomery exponentiation involves a conditional subtraction, which occurs with high probability when the decrypted value is slightly above one of the factors.

Standard Countermeasure — Blinding:

- The idea is to remove control over the data being decrypted from the attacker.
- We decrypt a "randomized" version of the ciphertext — randomized in a way that we can "undo" the randomization after decryption.

  (Trick borrowed from blind signatures)
- **Inconvenience: Performance penalty!** (reported to be 2 to 10%)

An alternative Countermeasure — Idle-Wait:

- If we wait after the decryption is completed, to make the observed decryption time constant, we effectively counter the key detail for the attacks: Decryption time is now independent of the data.

- Potential advantage: Making the decryption time constant through an idle-wait after the modular exponentiation allows other tasks to proceed (with blinding, the performance penalty comes in terms of actual processing).

Adaptive Idle-Wait Countermeasure:

- Intuitively, one might expect that making the total decryption time larger than the decryption time for any possible ciphertext would likely involve a high performance penalty — turns out that this is not the case!

- Performance penalty can be further reduced if we make the total decryption time larger than the decryption time for a given fraction of all possible ciphertexts — a given *percentile*.

Adaptive Idle-Wait Countermeasure:

- The “adaptive” part is given by the fact that we adjust the value of the total decryption time to follow the collected statistics of the decryption times, trying to set the target decryption time at the value corresponding to the target percentile.

Adaptive Idle-Wait Countermeasure:

- Clearly, the brute-force solution is ridiculously inefficient.
- Our proposed approach is roughly based on the same idea as the Newton-Raphson method for solving non-linear equations.
- The equation we're trying to solve is $F(T) = P$ where F is the CDF, P is the target percentile (e.g., 0.99), and T is the decryption time for that percentile.

Adaptive Idle-Wait Countermeasure:

- Unlike with the Newton-Raphson method, we do not have a closed-form description of the function or its derivative.
- Instead, we can have an approximation, given by counting values below certain thresholds.
- These thresholds correspond to approximations of the target percentiles.
- We use three thresholds — one at the target percentile, and two surrounding it.

Adaptive Idle-Wait Countermeasure:

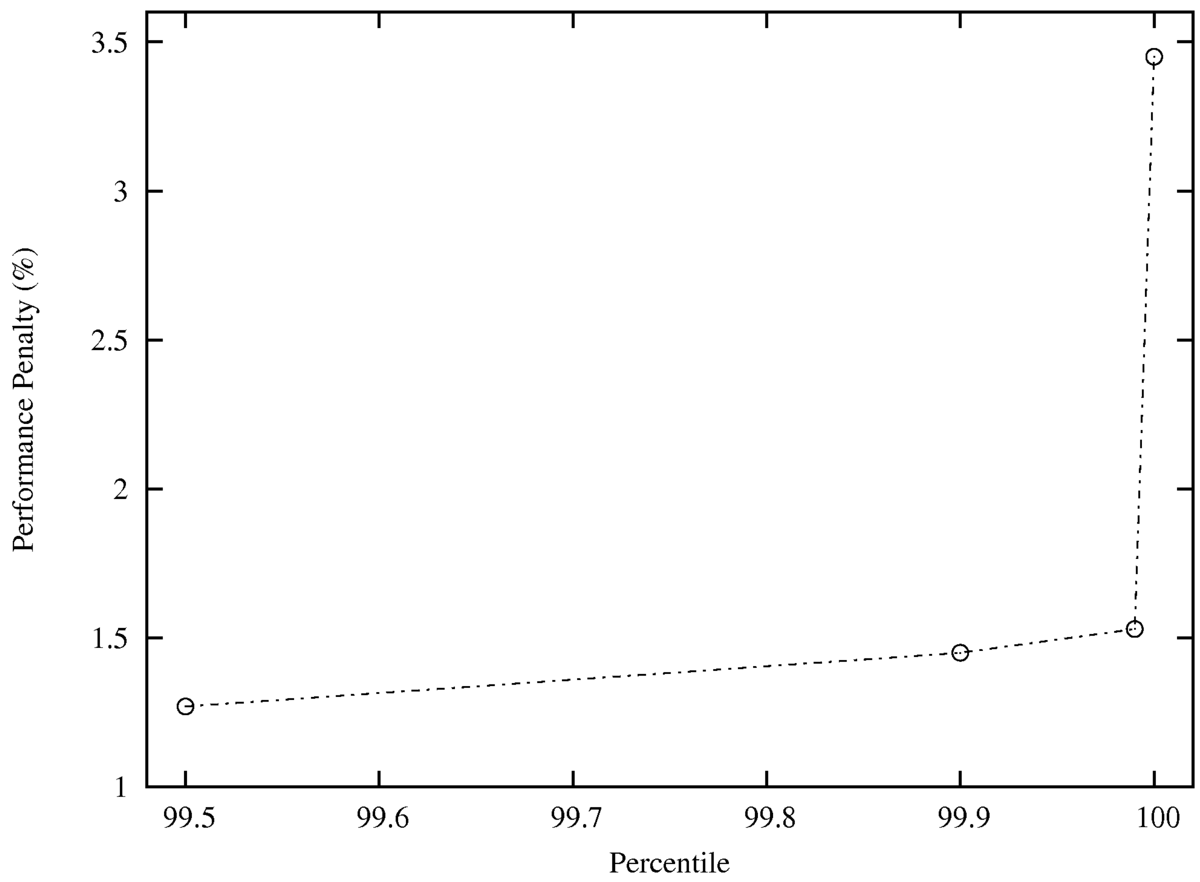
- For example, if we want a target percentile 99, (i.e., $P = 0.99$), then we could have thresholds corresponding to 0.985, 0.99, 0.995.
- The two surrounding thresholds provide an approximation for $F'(T)$, which we use to refine our estimate of T at each iteration.

Experimental Results:

- Based on simulations — simulate addition of this countermeasure to OpenSSL by running the decryption operation on a set of ciphertexts (randomly chosen), and use those times to adjust the required amount of idle-wait.

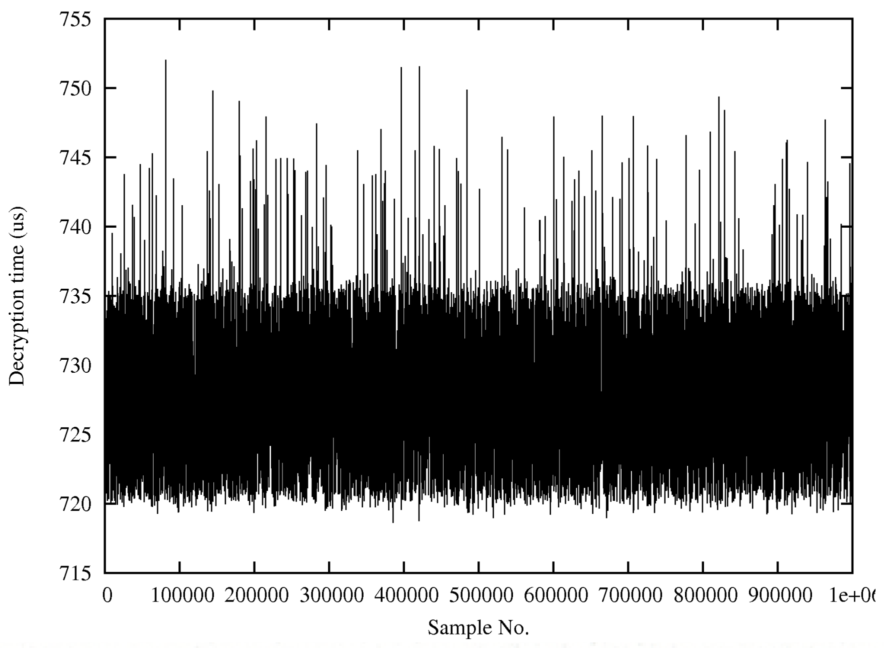
- This allows us to measure the performance penalty (given by the difference between the decryption time and the parameter T).

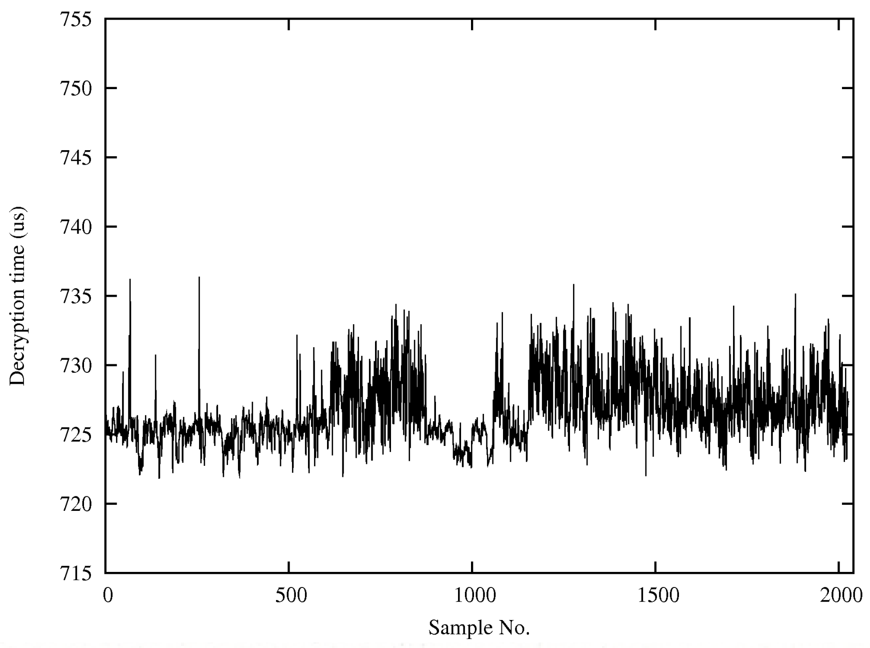
Experimental Results:



Experimental Results:

Defeated attack on OpenSSL with percentile 99.99 (performance penalty 1.53%)





Conclusions:

- Adaptive Idle-Wait method more effective, in terms of performance, than Blinding.
- Demonstrated effectiveness against known attack, at a reasonable performance penalty.
- Effective against any possible timing attack, at a still reasonable performance penalty.
- For software implementations where multi-tasking is a requirement, benefit is twofold.

Questions?

Blinding (details):

- Given ciphertext y corresponding to plaintext x, instead of decrypting y itself, we decrypt $y{\cdot}r^e$, where r is randomly chosen (and kept secret).
- Instead of x (the required result), we obtain:
  $$(y{\cdot}r^e)^d \bmod m = y^d \cdot (r^e)^d \bmod m = x{\cdot}r \bmod m$$
- After the modular exponentiation, we just multiply by r^{-1} to obtain the correct result x.

Adaptive Idle-Wait Countermeasure:

- We adjust the "main" threshold and then the two surrounding thresholds:

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

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

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

Idle-Wait — Implementation Requirements:

- If poorly/naively implemented, this counter-measure can be *trivially* bypassed — key detail: the idle-wait period allows for other concurrent tasks to proceed.
- These tasks could include other decryptions, allowing the attacker to measure the decryption time through the *throughput* of operations.

Idle-Wait — Implementation Requirements:

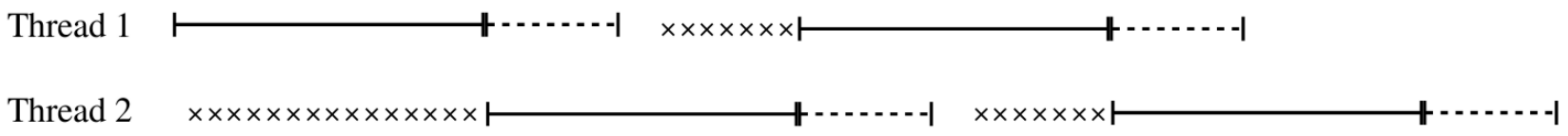


Fig. 1. Throughput of Concurrent RSA Decryption Operations.

Idle-Wait — Implementation Requirements:

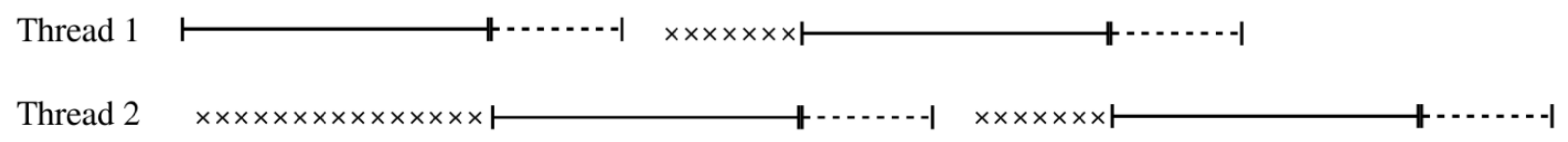


Fig. 1. Throughput of Concurrent RSA Decryption Operations.

Solution: Control concurrency, not allowing a decryption operation to begin while any other operation, including its idle-wait phase, is still in progress.

Idle-Wait Countermeasures Against Timing-Attacks on RSA Decryptions

Carlos Moreno