Timing Attacks

Vitaly Shmatikov

Reading

- Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems" (CRYPTO 1996).
- Brumley and Boneh. "Remote Timing Attacks Are Practical" (Best Paper Award, USENIX Security 2003).

Attacking Cryptographic Schemes

- Cryptanalysis
  - Find mathematical weaknesses in constructions
  - Statistical analysis of plaintext / ciphertext pairs
- Side channel attacks
  - Exploit characteristics of <u>implementations</u>
  - Power analysis
  - Electromagnetic radiation analysis
  - Acoustic analysis
  - Timing analysis

Timing Attack

- Basic idea: learn the system's secret by observing how long it takes to perform various computations
- Typical goal: extract private key
- Extremely powerful because isolation doesn't help
  - Victim could be remote
  - Victim could be inside its own virtual machine
  - Keys could be in tamper-proof storage or smartcard
- Attacker wins simply by measuring response times

RSA Cryptosystem

- Key generation:
  - Generate large (say, 512-bit) primes p, q
  - Compute $n=pq$ and $\varphi(n)=(p-1)(q-1)$
  - Choose small e, relatively prime to $\varphi(n)$
    - Typically, $e=3$ (may be vulnerable) or $e=2^{16}+1=65537$ (why?)
  - Compute unique d such that $ed = 1 \bmod \varphi(n)$
  - Public key = (e,n); private key = d
    - Security relies on the assumption that it is difficult to compute roots modulo n without knowing p and q
- Encryption of m (simplified!): $c = m^e \bmod n$
- Decryption of c: $c^d \bmod n = (m^e)^d \bmod n = m$

How Does RSA Decryption Work?

- RSA decryption: compute y^x mod n
  - This is a modular exponentiation operation
- Naïve algorithm: square and multiply

```
Let s_0 = 1.
For k = 0 upto w − 1:
   If (bit k of x) is 1 then
      Let R_k = (s_k · y) mod n.
   Else
      Let R_k = s_k.
   Let s_{k+1} = R_k^2 mod n.
EndFor.
Return (R_{w−1}).
```

Kocher's Observation

Whether iteration takes a long time depends on the k^th bit of secret exponent

```
Let s_0 = 1.
For k = 0 upto w − 1:
  If (bit k of x) is 1 then
    Let R_k = (s_k · y) mod n.
  Else
    Let R_k = s_k.
  Let s_{k+1} = R_k^2 mod n.
EndFor.
Return (R_{w−1}).
```

This takes a while to compute

This is instantaneous

Outline of Kocher's Attack

- Idea: guess some bits of the exponent and predict how long decryption will take
- If guess is correct, will observe correlation; if incorrect, then prediction will look random
  - This is a signal detection problem, where signal is timing variation due to guessed exponent bits
  - The more bits you already know, the stronger the signal, thus easier to detect (error-correction property)
- Start by guessing a few top bits, look at correlations for each guess, pick the most promising candidate and continue

RSA in OpenSSL

- OpenSSL is a popular open-source toolkit
  - mod_SSL (in Apache = 28% of HTTPS market)
  - stunnel (secure TCP/IP servers)
  - sNFS (secure NFS)
  - Many more applications
- Kocher's attack doesn't work against OpenSSL
  - Instead of square-and-multiply, OpenSSL uses CRT, sliding windows and two different multiplication algorithms for modular exponentiation
    - CRT = Chinese Remainder Theorem
    - Secret exponent is processed in chunks, not bit-by-bit

Chinese Remainder Theorem

- $n = n_1 n_2 \ldots n_k$

  where $\gcd(n_i, n_j) = 1$ when $i \neq j$
- The system of congruences

  $x = x_1 \bmod n_1 = \ldots = x_k \bmod n_k$
  - Has a simultaneous solution x to all congruences
  - There exists exactly one solution x between 0 and n-1
- For RSA modulus n=pq, to compute x mod n it's enough to know x mod p and x mod q

RSA Decryption With CRT

- To decrypt c, need to compute $m=c^d \bmod n$
- Use Chinese Remainder Theorem (why?)
  - $d_1 = d \bmod (p-1)$
  - $d_2 = d \bmod (q-1)$
  - $qinv = q^{-1} \bmod p$

  these are precomputed

  - Compute $m_1 = c^{d1} \bmod p$; $m_2 = c^{d2} \bmod q$
  - Compute $m = m_2 + (qinv*(m_1-m_2) \bmod p)*q$

Attack this computation in order to learn q.
This is enough to learn private key **(why?)**

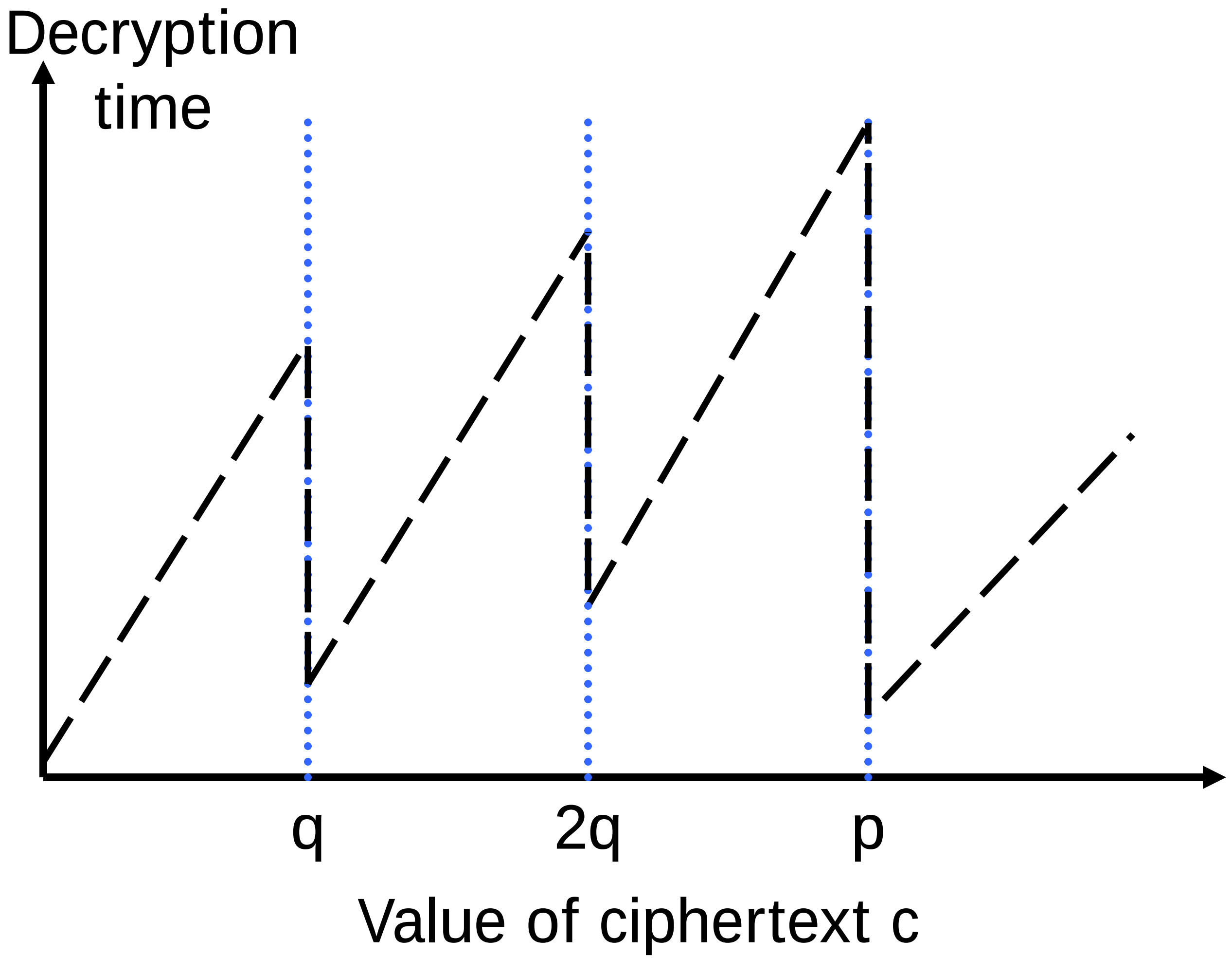
Montgomery Reduction

- Decryption requires computing $m_2 = c^{d2} \bmod q$
- This is done by repeated multiplication
  - Simple: square and multiply (process d_2 1 bit at a time)
  - More clever: sliding windows (process d_2 in 5-bit blocks)
- In either case, many multiplications modulo q
- Multiplications use Montgomery reduction
  - Pick some $R = 2^k$
  - To compute $x*y \bmod q$, convert x and y into their Montgomery form $xR \bmod q$ and $yR \bmod q$
  - Compute $(xR * yR) * R^{-1} = zR \bmod q$
    - Multiplication by R^{-1} can be done very efficiently

Schindler's Observation

- At the end of Montgomery reduction, if zR > q, then need to subtract q
  - Probability of this extra step is proportional to c mod q
- If c is close to q, a lot of subtractions will be done
- If c mod q = 0, very few subtractions
  - Decryption will take longer as c gets closer to q, then become fast as c passes a multiple of q
- By playing with different values of c and observing how long decryption takes, attacker can guess q!
  - Doesn't work directly against OpenSSL because of sliding windows and two multiplication algorithms

Reduction Timing Dependency



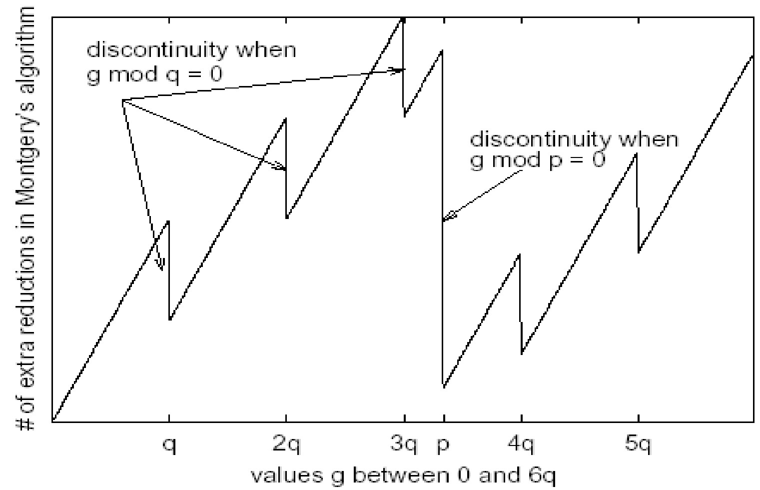
Integer Multiplication Routines

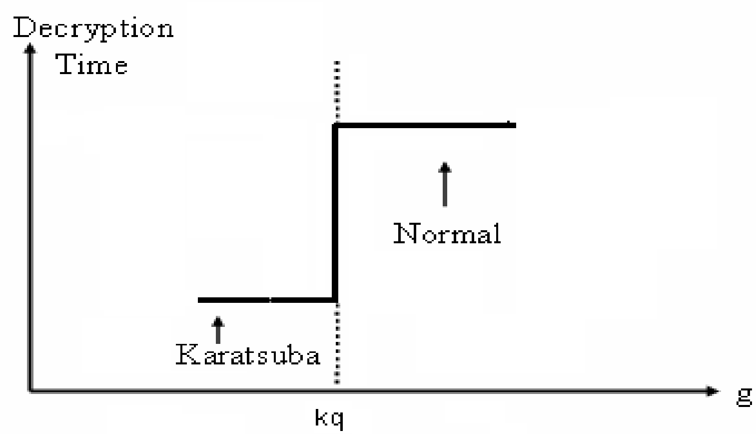
- 30-40% of OpenSSL running time is spent on integer multiplication
- If integers have the same number of words n, OpenSSL uses Karatsuba multiplication
  - Takes $O(n^{\log_2 3})$
- If integers have unequal number of words n and m, OpenSSL uses normal multiplication
  - Takes O(nm)

Summary of Time Dependencies

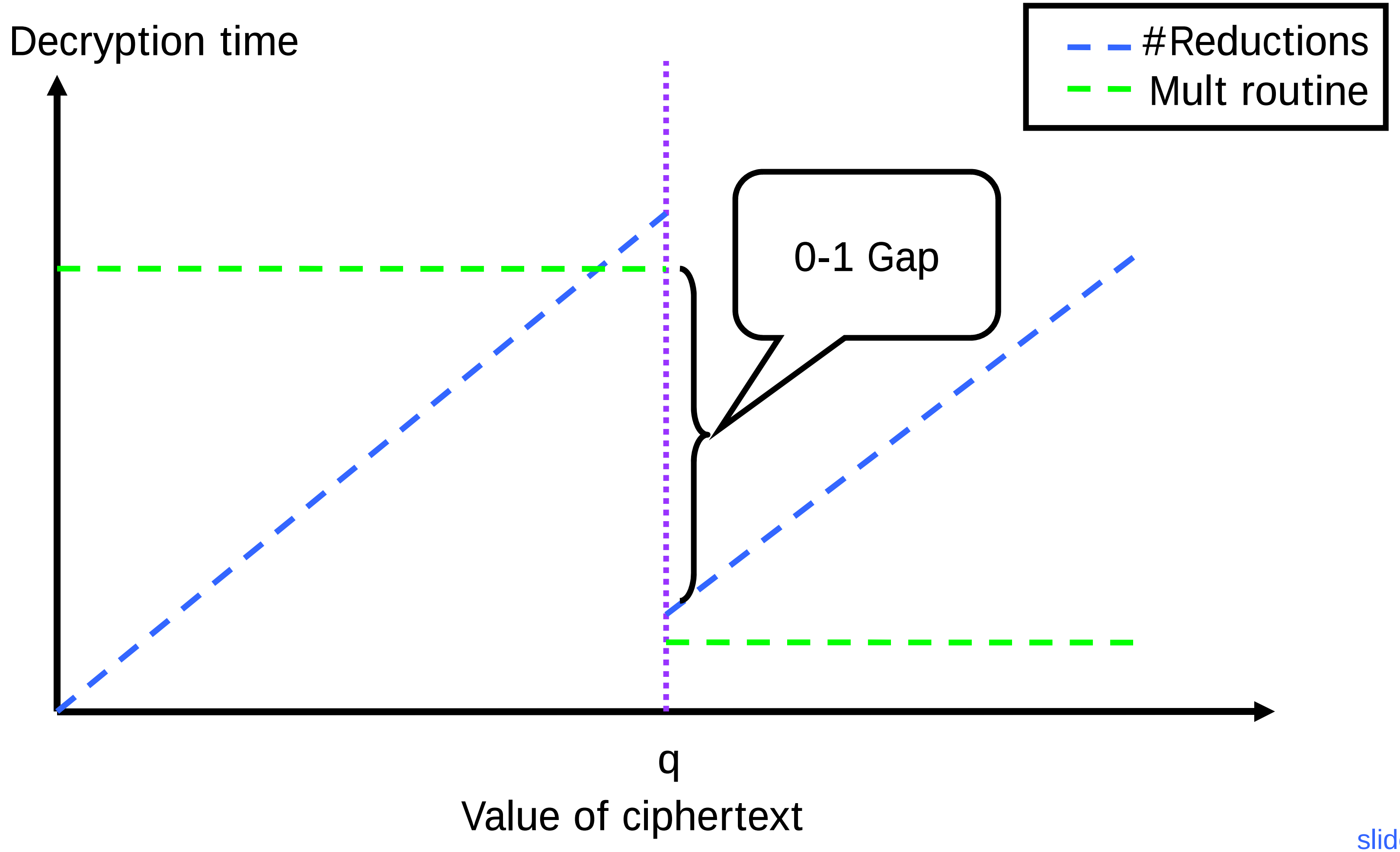
	g<q	g>q
Montgomery effect	Longer	Shorter
Multiplication effect	Shorter	Longer

g is the decryption value (same as c)
Different effects… but one will always dominate!



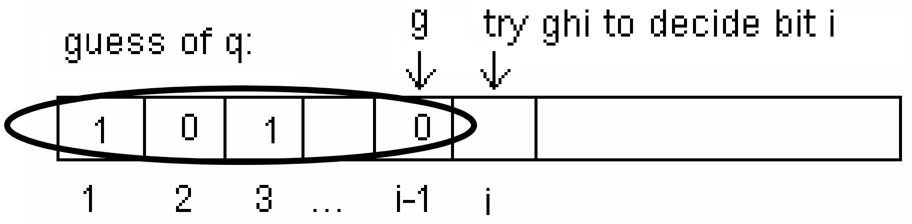


Attack Is Binary Search

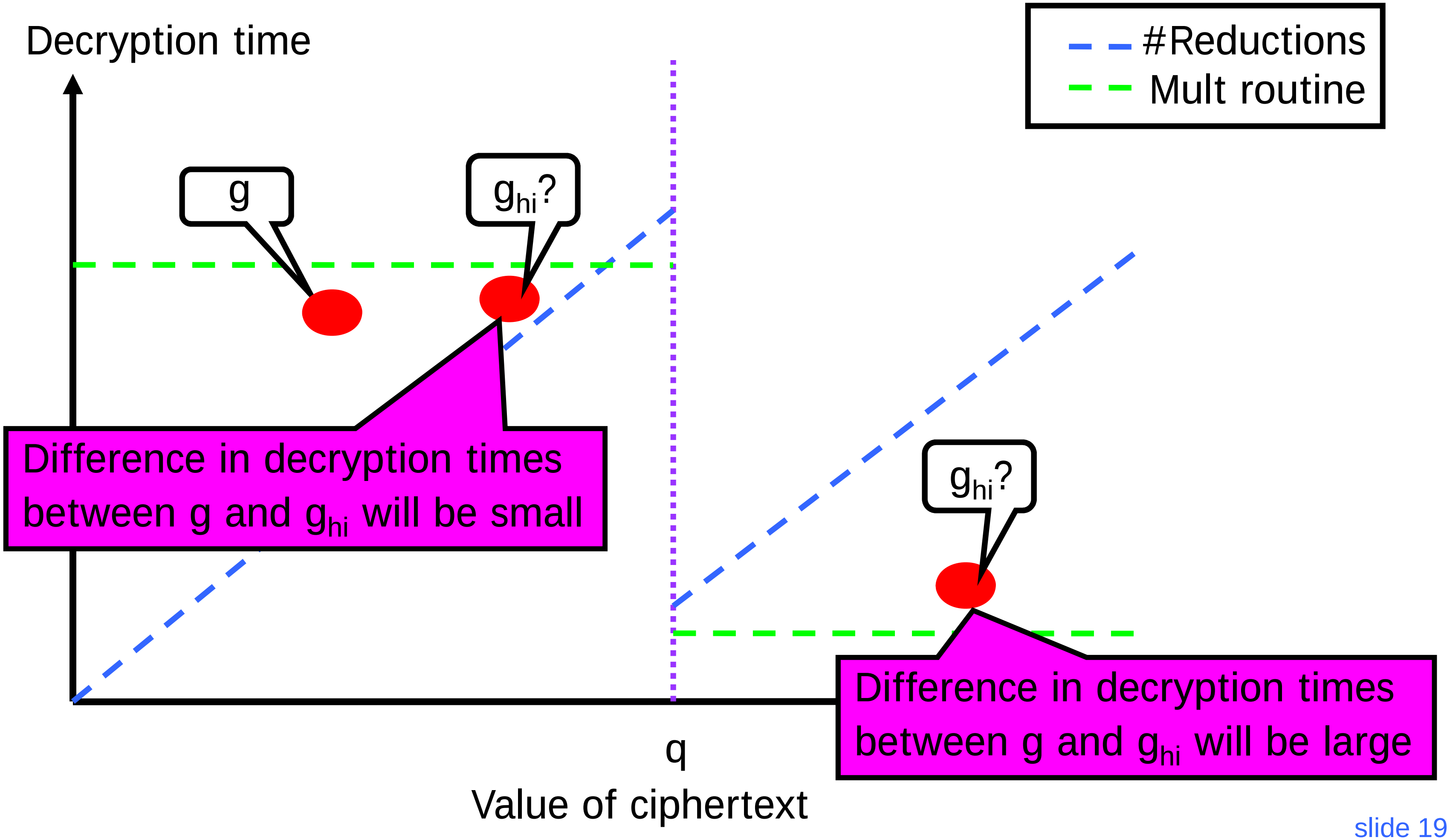


Attack Overview

- Initial guess g for q between 2^{511} and 2^{512} (why?)
- Try all possible guesses for the top few bits
- Suppose we know i-1 top bits of q. Goal: i^{th} bit
  - Set g =...known i-1 bits of q...000000
  - Set g_{hi}=...known i-1 bits of q...100000 (note: $g<g_{hi}$)
    - If $g<q<g_{hi}$ then the i^{th} bit of q is 0
    - If $g<g_{hi}<q$ then the i^{th} bit of q is 1
- Goal: decide whether $g<q<g_{hi}$ or $g<g_{hi}<q$



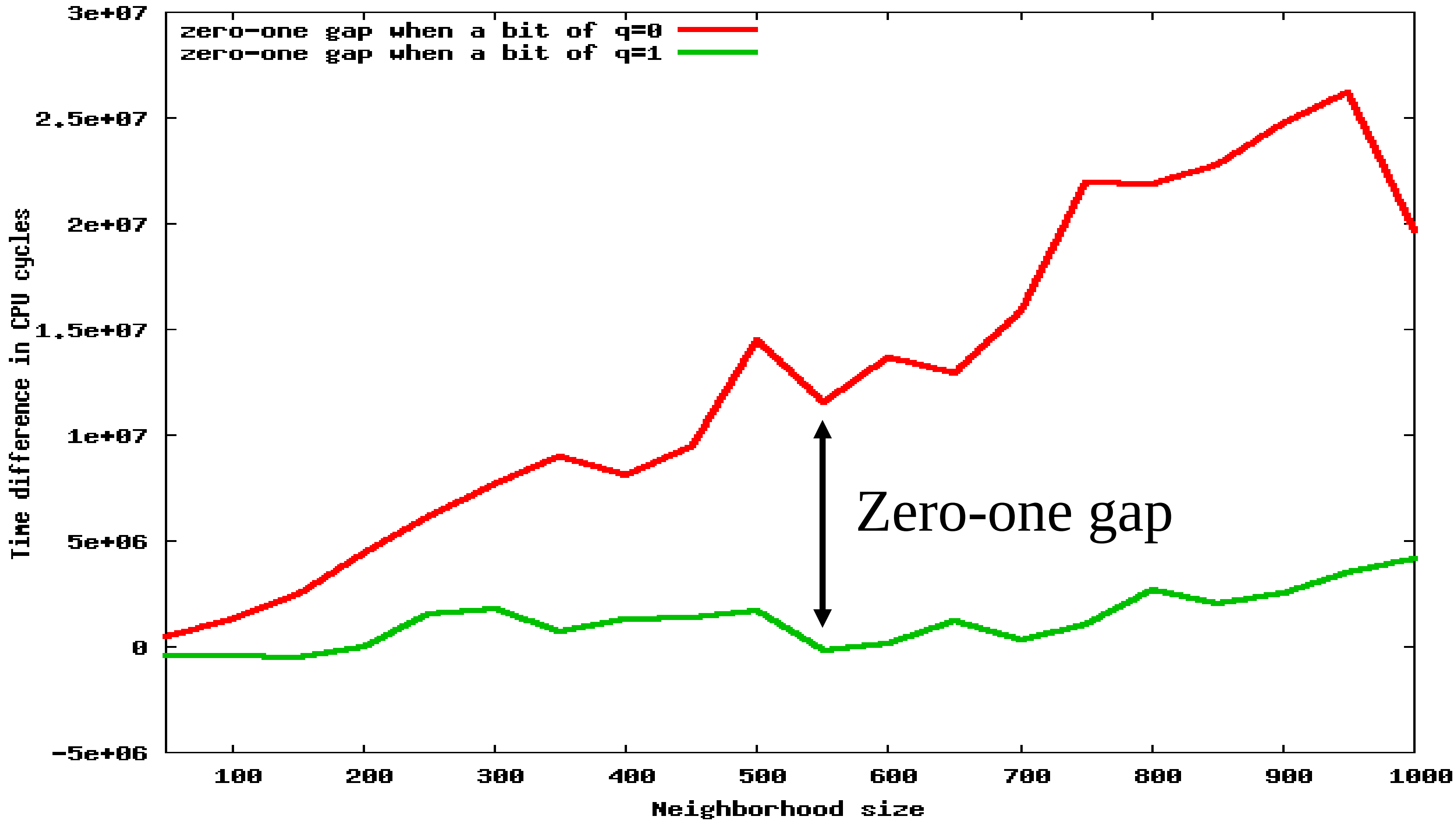
Two Possibilities for g_{hi}



Timing Attack Details

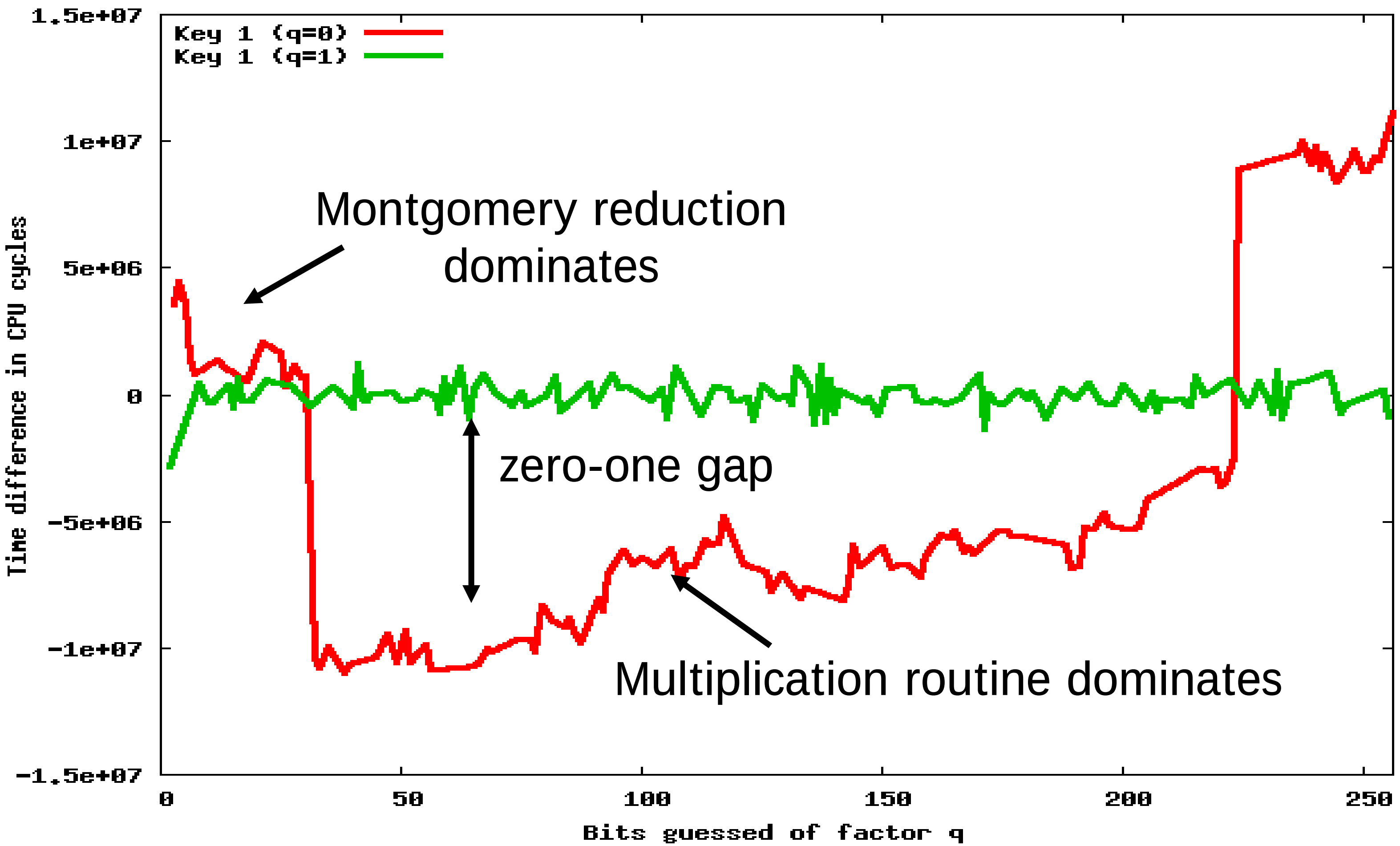
- What is "large" and "small"?
  - Know from attacking previous bits
- Decrypting just g does not work because of sliding windows
  - Decrypt a neighborhood of values near g
  - Will increase difference between large and small values, resulting in larger 0-1 gap
- Attack requires only 2 hours, about 1.4 million queries to recover the private key
  - Only need to recover most significant half bits of q

The 0-1 Gap

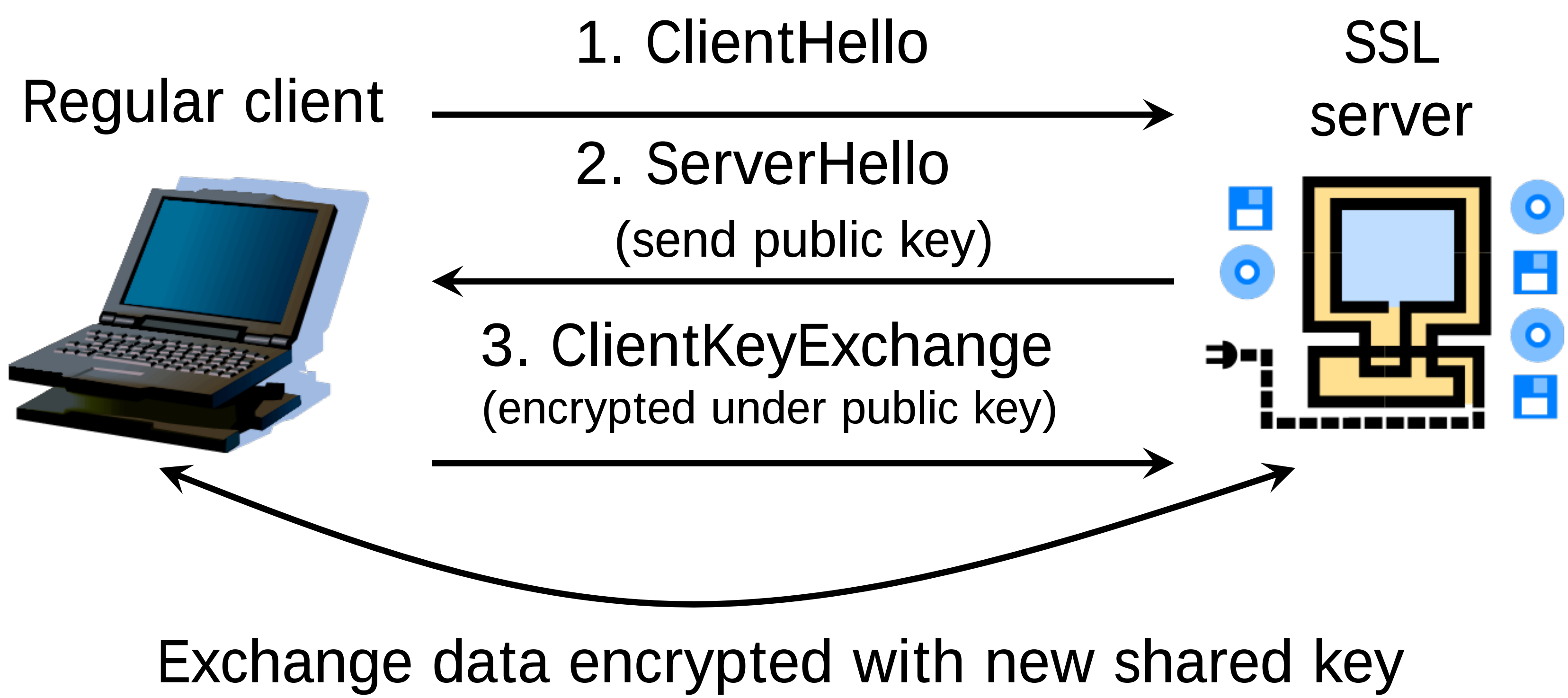


zero-one gap when a bit of q=0
zero-one gap when a bit of q=1
3e+07
2.5e+07
2e+07
1.5e+07
1e+07
5e+06
0
-5e+06
Time difference in CPU cycles
100
200
300
400
500
600
700
800
900
1000
Neighborhood size
Zero-one gap

Extracting RSA Private Key



Normal SSL Handshake



Attacking SSL Handshake

Attacker

1. ClientHello

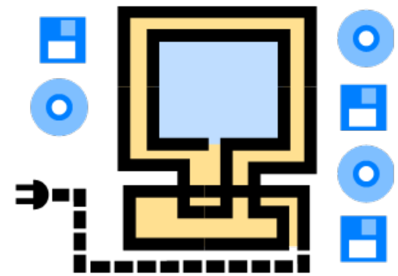
2. ServerHello
   (send public key)

3. Record time t_1
   Send guess g or g_{hi}

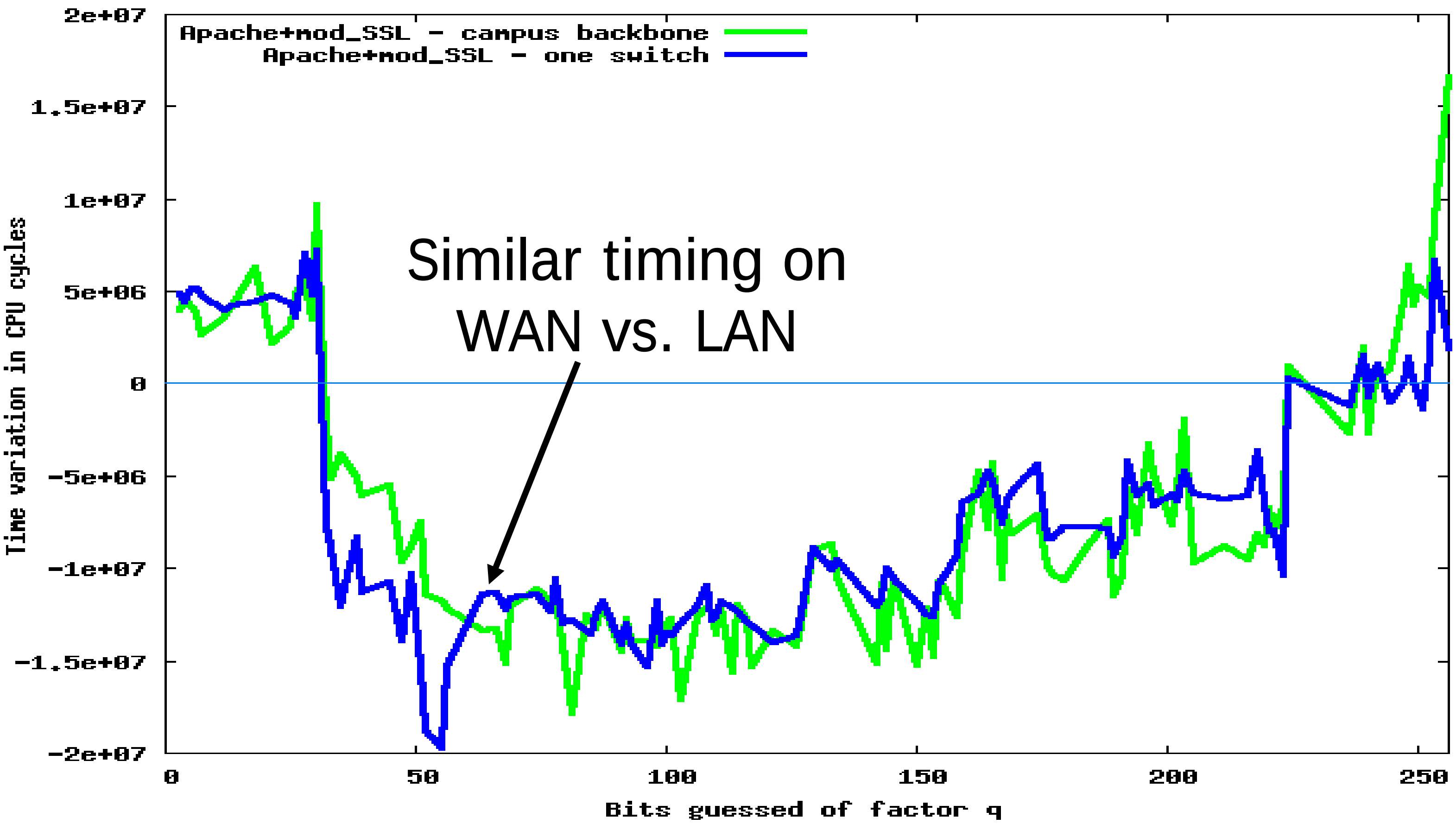
4. Alert

5. Record time t_2
   Compute $t_2 – t_1$

SSL server



Works On The Network



Defenses

- Good: Use RSA blinding
- Worse: require statically that all decryptions take the same time
  - For example, always do the extra “dummy” reduction
  - ... but what if compiler optimizes it away?
- Worse: dynamically make all decryptions the same or multiples of the same time “quantum”
  - Now all decryptions have to be as slow as the slowest decryption

RSA Blinding

- Instead of decrypting ciphertext c, decrypt a random ciphertext related to c
  - Compute x' = $c*r^e$ mod n, r is random
  - Decrypt x' to obtain m'
  - Calculate original plaintext m = m'/r mod n
- Since r is random, decryption time is random
- 2-10% performance penalty

Blinding Works