Side channel attacks: historically worried about EM signals leaking.

- Ref: http://cryptome.org/nsa-tempest.pdf
- Broadly, systems may need to worry about many unexpected ways in which information can be revealed.

Example setting: a server (e.g., Apache) has an RSA private key.

- Server uses RSA private key (e.g., decrypt message from client).
- Something about the server’s computation is leaked to the client.

Many information leaks have been looked at:

- How long it takes to decrypt.
- How decryption affects shared resources (cache, TLB, branch predictor).
- Emissions from the CPU itself (RF, audio, power consumption, etc).

Side-channel attacks don’t have to be crypto-related.

- E.g., operation time relates to which character of password was incorrect.
- Or time related to how many common friends you + some user have on Facebook.
- Or how long it takes to load a page in browser (depends if it was cached).
- Or recovering printed text based on sound from dot-matrix printer.
  - Ref: https://www.usenix.org/conference/usenixsecurity10/acoustic-side-channel-attacks-printers
- But attacks on passwords or keys are usually the most damaging.

Adversary can analyze information leaks, use it to reconstruct private key.

- Currently, side-channel attacks on systems described in the paper are rare.
  - E.g., Apache web server running on some Internet-connected machine.
  - Often some other vulnerability exists and is easier to exploit.
  - Slowly becoming a bigger concern: new side-channels (VMs), better attacks.
- Side-channel attacks are more commonly used to attack trusted/embedded hw.
  - E.g., chip running cryptographic operations on a smartcard.
  - Often these have a small attack surface, not many other ways to get in.
  - As paper mentions, some crypto coprocessors designed to avoid this attack.

What’s this paper’s contribution?

- Timing attacks known for a while.
- This paper: possible to attack standard Apache web server over the network.
- Uses lots of observations/techniques from prior work on timing attacks.
- To understand how this works, first let’s look at some internals of RSA...
RSA: high level plan
- Pick two random primes, p and q. Let n = p*q.
- A reasonable key length, i.e., |n| or |d|, is 2048 bits today.
- Euler’s function \( \phi(n) \): number of elements of \( \mathbb{Z}_n \) relatively prime to n.
  - Theorem [no proof here]: \( a^{\phi(n)} = 1 \mod n \), for all \( a \) and \( n \).
- So, how to encrypt and decrypt?
  - Pick two exponents \( d \) and \( e \), such that \( m^{e*d} = m \mod n \), which means \( e*d = 1 \mod \phi(n) \).
  - Encryption will be \( c = m^e \mod n \); decryption will be \( m = c^d \mod n \).
- How to get such \( e \) and \( d \)?
  - For \( n=pq \), \( \phi(n) = (p-1)(q-1) \).
  - Easy to compute \( d=1/e \), if we know \( \phi(n) \).
  - Extended Euclidean algorithm.
  - In practice, pick small \( e \) (e.g., 65537), to make encryption fast.
- Public key is \( (n, e) \).
- Private key is, in principle, \( (n, d) \).
  - Note: \( p \) and \( q \) must be kept secret!
  - Otherwise, adversary can compute \( d \) from \( e \), as we did above.
  - Knowing \( p \) and \( q \) also turns out to be helpful for fast decryption.
  - So, in practice, private key includes \( (p, q) \) as well.

RSA is tricky to use "securely" -- be careful if using RSA directly!
- Ciphertexts are multiplicative
  - \( E(a)*E(b) = a^e * b^e = (ab)^e \).
  - Can allow adversary to manipulate encryptions, generate new ones.
- RSA is deterministic
  - Encrypting the same plaintext will generate the same ciphertext each time.
  - Adversary can tell when the same thing is being re-encrypted.
- Typically solved by "padding" messages before encryption.
  - Take plaintext message bits, add padding bits before and after plaintext.
  - Encrypt the combined bits (must be less than \(|n| \) bits total).
  - Padding includes randomness, as well as fixed bit patterns.
  - Helps detect tampering (e.g. ciphertext multiplication).

How to implement RSA?
- Key problem: fast modular exponentiation.
  - In general, quadratic complexity.
- Multiplying two 1024-bit numbers is slow.
- Computing the modulus for 1024-bit numbers is slow (1024-bit division).

Optimization 1: Chinese Remainder Theorem (CRT).
- Recall what the CRT says:
if \( x = a_1 \pmod{p} \) and \( x = a_2 \pmod{q} \), where \( p \) and \( q \) are relatively prime, then there's a unique solution \( x = a \pmod{pq} \). (and, there's an efficient algorithm for computing \( a \))

- Suppose we want to compute \( m = c^d \pmod{pq} \).
- Can compute \( m_1 = c^d \pmod{p} \), and \( m_2 = c^d \pmod{q} \).
- Then use CRT to compute \( m = c^d \pmod{n} \) from \( m_1, m_2 \); it's unique and fast.
- Computing \( m_1 \) (or \( m_2 \)) is \(~4x\) faster than computing \( m \) directly (~quadratic).
- Computing \( m \) from \( m_1 \) and \( m_2 \) using CRT is ~negligible in comparison.
- So, roughly a 2x speedup.

Optimization 2: Repeated squaring and Sliding windows.
- Naive approach to computing \( c^d \): multiply \( c \) by itself, \( d \) times.
- Better approach, called repeated squaring:
  - \( c^{(2x)} = (c^x)^2 \)
  - \( c^{(2x+1)} = (c^x)^2 \times c \)
  - To compute \( c^d \), first compute \( c^{(\text{floor}(d/2))} \), then use above for \( c^d \).
  - Recursively apply until the computation hits \( c^0 = 1 \).
  - Number of squarings: \( |d| \)
  - Number of multiplications: number of 1 bits in \( d \)
- Better yet (sometimes), called sliding window:
  - \( c^{(32x+1)} = (c^x)^{32} \times c \)
  - \( c^{(32x+3)} = (c^x)^{32} \times c^3 \)
  - \( ... \)
  - \( c^{(32x+z)} = (c^x)^{32} \times c^z \), generally [where \( z \leq 31 \)]
  - Can pre-compute a table of all necessary \( c^z \) powers, store in memory.
  - The choice of power-of-2 constant (e.g., \( 32 \)) depends on usage.
    - Costs: extra memory, extra time to pre-compute powers ahead of time.
    - Note: only pre-compute odd powers of \( c \) (use first rule for even).
    - OpenSSL uses 32 (table with 16 pre-computed entries).

Optimization 3: Montgomery representation.
- Reducing mod \( p \) each time (after square or multiply) is expensive.
  - Typical implementation: do long division, find remainder.
  - Hard to avoid reduction: otherwise, value grows exponentially.
- Idea (by Peter Montgomery): do computations in another representation.
  - Shift the base (e.g., \( c \)) into different representation upfront.
  - Perform modular operations in this representation (will be cheaper).
  - Shift numbers back into original representation when done.
  - Ideally, savings from reductions outweigh cost of shifting.
- Montgomery representation: multiply everything by some factor \( R \).
  - \( a \pmod{q} \leftrightarrow aR \pmod{q} \)
  - \( b \pmod{q} \leftrightarrow bR \pmod{q} \)
  - \( c = a*b \pmod{q} \leftrightarrow cR \pmod{q} = (aR * bR)/R \pmod{q} \)
Each mul (or sqr) in Montgomery-space requires division by $R$.

Why is modular multiplication cheaper in montgomery rep?

○ Choose $R$ so division by $R$ is easy: $R = 2^{|q|}$ (2^512 for 1024-bit keys).
○ Because we divide by $R$, we will often not need to do mod $q$.

- $|aR| = |q|$
- $|bR| = |q|$
- $|aR \times bR| = 2|q|$
- $|aR \times bR \mod R| = |q|$

○ How do we divide by $R$ cheaply? Only works if lower bits are zero.
○ Observation: since we care about value mod $q$, multiples of $q$ don’t matter.
○ Trick: add multiples of $q$ to the number being divided by $R$, make low bits 0.

- For example, suppose $R=2^4$ (10000), $q=7$ (111), divide $x=26$ (11010) by $R$.
  - $x+2q = (binary) 101000$
  - $x+2q+8q = (binary) 1100000$
- Now, can easily divide by $R$: result is binary 110 (or 6).
- Generally, always possible:
  - Low bit of $q$ is 1 ($q$ is prime), so can "shoot down" any bits.
  - To "shoot down" bit $k$, add $2^k \times q$
  - To shoot down low-order bits $l$, add $q \times (l \times (-q^{-1}) \mod R)$
  - Then, dividing by $R$ means simply discarding low zero bits.

○ One remaining problem: result will be $< R$, but might be $> q$.
  ○ If the result happens to be greater than $q$, need to subtract $q$.
  ○ This is called the "extra reduction".
  ○ When computing $x^d \mod q$, $Pr[extra\ reduction] = (x \mod q) / 2R$.
    ▪ Here, $x$ is assumed to be already in Montgomery form.
    ▪ Intuition: as we multiply bigger numbers, will overflow more often.

Optimization 4: Efficient multiplication.

○ How to multiply 512-bit numbers?
○ Representation: break up into 32-bit values (or whatever hardware supports).
○ Naive approach: pair-wise multiplication of all 32-bit components.
  ○ Same as if you were doing digit-wise multiplication of numbers on paper.
  ○ Requires $O(nm)$ time if two numbers have $n$ and $m$ components respectively.
  ○ $O(n^2)$ if the two numbers are close.
○ Karatsuba multiplication: assumes both numbers have same number of components.
  ○ $O(n^{\log_2 3}) = O(n^{1.585})$ time.
  ○ Split both numbers ($x$ and $y$) into two components ($x_1, x_0$ and $y_1, y_0$).
    ▪ $x = x_1 \times B + x_0$
    ▪ $y = y_1 \times B + y_0$
- E.g., B=2^32 when splitting 64-bit numbers into 32-bit components.
  - Naive: x*y = x1y1 * B^2 + x0y1 * B + x1y0 * B + x0y0
    - Four multiplies: O(n^2).
  - Faster:
    \[ x*y = x1y1 \cdot (B^2+B) - (x1-x0)(y1-y0) \cdot B + x0y0 \cdot (B+1) = x1y1 \cdot B^2 + (- (x1-x0)(y1-y0) + x1y1 + x0y0) \cdot B + x0y0 \]
    - Just three multiplies, and a few more additions.
  - Recursively apply this algorithm to keep splitting into more halves.

- Meaningfully faster (no hidden big constants)
  - For 1024-bit keys, "n" here is 16 (512/32).
  - \( n^2 = 256 \)
  - \( n^{1.585} = 81 \)

- Multiplication algorithm needs to decide when to use Karatsuba vs. Naive.
- Two cases matter: two large numbers, and one large + one small number.
- OpenSSL: if equal number of components, use Karatsuba, otherwise Naive.
- In some intermediate cases, Karatsuba may win too, but OpenSSL ignores it, according to this paper.

How does SSL use RSA?
- Server's SSL certificate contains public key.
- Server must use private key to prove its identity.
- Client sends random bits to server, encrypted with server's public key.
- Server decrypts client's message, uses these bits to generate session key.
  - In reality, server also verifies message padding.
  - However, can still measure time until server responds in some way.

Figure of decryption pipeline on the server:

<table>
<thead>
<tr>
<th>CRT</th>
<th>Mod</th>
<th>Montgomery</th>
<th>Modular exp</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>c_0 = c mod q</td>
<td>c'_0 = c_0*R mod q</td>
<td>m'_0 = (c'_0)^d mod q</td>
</tr>
</tbody>
</table>

Use sliding window for bits of the exponent d

Use sliding window for bits of the exponent d

Karatsuba if c'_0 and q have same number of 32-bit parts

Extra reductions proportional to \(((c'_0)^z mod q) / 2R; z comes from sliding window

Then, compute m_0 = m'_0/R mod q.
Then, combine m_0 and m_1 using CRT to get m.
Then verify padding in m.
Finally, use payload in some way (SSL, etc).

Setup for the attack described in Brumley's paper.
• Victim Apache HTTPS web server using OpenSSL, has private key in memory.
• Connected to Stanford’s campus network.
• Adversary controls some client machine on campus network.
• Adversary sends specially-constructed ciphertext in msg to server.
  o Server decrypts ciphertext, finds garbage padding, returns an error.
  o Client measures response time to get error message.
  o Uses the response time to guess bits of q.
• Overall response time is on the order of 5 msec.
  o Time difference between requests can be around 10 usec.
• What causes time variations? Karatsuba vs normal; extra reductions.
• Once guessed enough bits of q, can factor n=p*q, compute d from e.
• About 1M queries seem enough to obtain 512-bit p and q for 1024-bit key.
  o Only need to guess the top 256 bits of p and q, then use another algorithm.

Attack from Brumley’s paper.
• Let $q = q_0 \ldots q_N$, where $N = |q|$ (say, 512 bits for 1024-bit keys).
• Assume we know some number j of high-order bits of q ($q_0$ through $q_j$).
• Construct two approximations of q, guessing $q_{j+1}$ is either 0 or 1:
  o $g = q_0 \ldots q_j 0 0 0 0$
  o $g_{hi} = q_0 \ldots q_j 1 0 0 0$
• Get the server to perform modular exponentiation ($g^d$) for both guesses.
  o We know g is necessarily less than q.
  o If g and $g_{hi}$ are both less than q, time taken shouldn't change much.
  o If $g_{hi}$ is greater than q, time taken might change noticeably.
    ▪ $g_{hi}$ mod q is small.
    ▪ Less time: fewer extra reductions in Montgomery.
    ▪ More time: switch from Karatsuba to normal multiplication.
  o Knowing the time taken can tell us if 0 or 1 was the right guess.
• How to get the server to perform modular exponentiation on our guess?
  o Send our guess as if it were the encryption of randomness to server.
  o One snag: server will convert our message to Montgomery form.
  o Since Montgomery's R is known, send $(g/R \mod n)$ as message to server.
• How do we know if the time difference should be positive or negative?
  o Paper seems to suggest it doesn't matter: just look for large diff.
  o Figure 3a shows the measured time differences for each bit's guess.
  o Karatsuba vs normal multiplication happens at 32-bit boundaries.
  o First 32 bits: extra reductions dominate.
  o Next bits: Karatsuba vs normal multiplication dominates.
  o At some point, extra reductions start dominating again.
• What happens if the time difference from the two effects cancels out?
  o Figure 3, key 3.
  o Larger neighborhood changes the balance a bit, reveals a non-zero gap.
• How does the paper get accurate measurements?
  o Client machine uses processor's timestamp counter (rdtsc on x86).
Measure several times, take the median value.
  - Not clear why median; min seems like it would be the true compute time.
  - One snag: relatively few multiplications by $g$, due to sliding windows.
  - Solution: get more multiplications by values close to $g$ (+ same for $g_{hi}$).
  - Specifically, probe a "neighborhood" of $g$ ($g, g+1, ..., g+400$).
- Why probe a 400-value neighborhood of $g$ instead of measuring $g$ 400 times?
  - Consider the kinds of noise we are trying to deal with.
  - Noise unrelated to computation (e.g. interrupts, network latency).
    - This might go away when we measure the same thing many times.
    - See Figure 2a in the paper.
  - "Noise" related to computation.
    - E.g., multiplying by $g^3$ and $g_{hi}^3$ in sliding window takes diff time.
    - Repeated measurements will return the same value.
    - Will not help determine whether mul by $g$ or $g_{hi}$ has more reductions.
    - See Figure 2b in the paper.
  - Neighborhood values average out 2nd kind of noise.
  - Since neighborhood values are nearby, still has ~same # reductions.

How to avoid these attacks?
- Timing attack on decryption time: RSA blinding.
  - Choose random $r$.
  - Multiply ciphertext by $r^e \mod n$: $c' = c*r^e \mod n$.
  - Due to multiplicative property of RSA, $c'$ is an encryption of $m*r$.
  - Decrypt ciphertext $c'$ to get message $m'$.
  - Divide plaintext by $r$: $m = m'/r$.
  - About a 10% CPU overhead for OpenSSL, according to Brumley's paper.
- Make all code paths predictable in terms of execution time.
  - Hard, compilers will strive to remove unnecessary operations.
  - Precludes efficient special-case algorithms.
  - Difficult to predict execution time: instructions aren't fixed-time.
- Can we take away access to precise clocks?
  - Yes for single-threaded attackers on a machine we control.
  - Can add noise to legitimate computation, but attacker might average.
  - Can quantize legitimate computations, at some performance cost.
  - But with "sleeping" quantization, throughput can still leak info.

How worried should we be about these attacks?
- Relatively tricky to develop an exploit (but that's a one-time problem).
- Possible to notice attack on server (many connection requests).
  - Though maybe not so easy on a busy web server cluster?
- Adversary has to be close by, in terms of network.
  - Not that big of a problem for adversary.
• Can average over more queries, co-locate nearby (Amazon EC2), run on a nearby bot or browser, etc.

• Adversary may need to know the version, optimization flags, etc of OpenSSL.
  o Is it a good idea to rely on such a defense?
  o How big of an impediment is this?

• If adversary mounts attack, effects are quite bad (key leaked).

Other types of timing attacks.
• Page-fault timing for password guessing [Tenex system]
  o Suppose the kernel provides a system call to check user's password.
    ▪ Checks the password one byte at a time, returns error when finds mismatch.
  o Adversary aligns password, so that first byte is at the end of a page, rest of password is on next page.
  o Somehow arrange for the second page to be swapped out to disk.
    ▪ Or just unmap the next page entirely (using equivalent of mmap).
  o Measure time to return an error when guessing password.
    ▪ If it took a long time, kernel had to read in the second page from disk.
    ▪ [ Or, if unmapped, if crashed, then kernel tried to read second page. ]
    ▪ Means first character was right!
  o Can guess an N-character password in 256*N tries, rather than 256^N.

• Cache analysis attacks: processor's cache shared by all processes.
  o E.g.: accessing one of the sliding-window multiples brings it in cache.
  o Necessarily evicts something else in the cache.
  o Malicious process could fill cache with large array, watch what's evicted.
  o Guess parts of exponent (d) based on offsets being evicted.

• Cache attacks are potentially problematic with "mobile code".
  o NaCl modules, Javascript, Flash, etc running on your desktop or phone.

• Network traffic timing / analysis attacks.
  o Even when data is encrypted, its ciphertext size remains ~same as plaintext.
  o Recent papers show can infer a lot about SSL/VPN traffic by sizes, timing.
  o E.g., Fidelity lets customers manage stocks through an SSL web site.
    ▪ Web site displays some kind of pie chart image for each stock.
    ▪ User's browser requests images for all of the user's stocks.
    ▪ Adversary can enumerate all stock pie chart images, knows sizes.
    ▪ Can tell what stocks a user has, based on sizes of data transfers.
  o Similar to CRIME attack mentioned in guest lecture earlier this term.

References:
• http://www.cs.unc.edu/~reiter/papers/2012/CCS.pdf
• http://ed25519.cr yp.to/