UNIVERSITY OF
WATERLOO

Department of Electrical
and Computer Engineering

Side-Channel Analysis

ECE-458 – Computer Security
(Guest lecture – 2013-02-05)

Carlos Moreno

cmoreno@uwaterloo.ca

EIT-4103

During today's lecture:

- Side-channel analysis or side-channel attacks
  - Preliminaries (some math to understand how these attacks work)
  - Timing attacks
    - Countermeasures
  - Power analysis
    - Simple Power Analysis (SPA)
      - Countermeasures
      - As if things weren't bad enough — fault attacks
    - Differential Power Analysis (DPA)
      - Countermeasures

Preliminaries:

- Public-key cryptosystems:
  - Diffie-Hellman key exchange protocol (1976)
    - Relies on the difficulty to solve the discrete logarithm problem for large numbers
  - RSA public-key and Digital signatures (1977)
    - Relies on the difficulty to factor large numbers
  - Both use modular exponentiation as a fundamental operation.

Preliminaries:

- Modular exponentiation: $x^e \bmod m$

  Smallest non-negative integer y such that $0 \leqslant y < m$ and $y = x^e - k\,m$ for some integer k

- In other words .... Remainder of the division by m (in this case, remainder of the division $x^e \div m$)

Preliminaries:

- Modular exponentiation: $x^e \bmod m$

- Can be computed efficiently (keep in mind that the values of the exponent are very large)
- We take advantage of the binary representation of the exponent — if e is an N bits number

$$e = b_{N-1} b_{N-2} \cdots b_2 b_1 b_0$$

- We traverse the bits of the exponent from left to right (most-significant to least-significant), and keep a partial result.

Preliminaries:

- Modular exponentiation: $x^e \bmod m$

- At some point, we have the value x^a and want to obtain $x^{a'}$, where a' has an additional bit b to the right. That is:

$$a' = 2a + b$$

$$\Rightarrow \quad x^{a'} = x^{2a+b} = x^{2a} \cdot x^b = \left(x^a\right)^2 \cdot x^b$$

Preliminaries:

- Modular exponentiation: $x^e \bmod m$

- The square-and-multiply algorithm:

```
R ← 1
For each bit of the exponent (left to right):
    R ← R² mod m
    If bit is 1 then
        R ← R·x mod m
    end if
end for
```

Preliminaries:

- Modular exponentiation: $x^e \bmod m$

- Detail: we can do modular reduction (the mod m part of the formula) before or after:

$$a \cdot b \bmod m = (a \bmod m) \cdot (b \bmod m) \bmod m$$

Since:

$$(A + k_1 m) \cdot (B + k_2 m) = A \cdot B + (k_1 B + k_2 A + k_1 k_2) m$$

Preliminaries:

- Back to public-key cryptosystems:
  - Diffie-Hellman key exchange protocol:
  - Alice and Bob want to communicate in secret, but have never communicated before (to exchange a secret key that they can both use with a cipher such as AES)

Preliminaries:

- Diffie-Hellman key exchange protocol:
  - Setup phase:
    - Alice and Bob agree on a large prime number p and a value g ($0 \leq g < p$)
  - Operation phase (key exchange protocol):
    - Alice picks a random number a, and computes g^a mod p
    - Bob picks a random number b, and computes g^b mod p
    - They transmit the results to each other
    - Bob receives g^a and raises it to the power b.
    - Alice receives g^b and raises it to the power a.
    - Bottom line: $(g^a)^b = (g^b)^a = g^{ab}$ (everything mod p)

Preliminaries:

- Diffie-Hellman key exchange protocol:
  - Why is it secure?
    - Discrete Logarithm problem:
      - Given g, a prime p, and g^x mod p, determine the value of x.
    - Turns out that no efficient method is known to solve this problem — hard problem: makes sense that a cryptographic protocol relies on it !
  - Key detail: Only works if numbers are really large!

Preliminaries:

- RSA public-key cryptosystem:
  - Allows sender (say, Alice) to encrypt with recipient's (Bob's) *public* key, in a way that only Bob (only the owner of the corresponding *private* key) can decrypt it (no-one else can).

Preliminaries:

- RSA public-key cryptosystem:
  - Without going into more details (there are some details at the end of the slides set, for those who may be curious), this is how it works:
  - Given a modulus *m* obtained as the product of two large primes *p* and *q*, we choose an encryption exponent *e*.
  - Bob's public key is the pair of values (*e*,*m*), with encryption function $\mathrm{E}(x) = x^e \bmod m$

Preliminaries:

- RSA public-key cryptosystem:
  - Turns out that if the factorization of m is known, it is easy to find a “magic” decryption exponent *d* to “undo” the exponentiation with exponent *e*.
  - Bob's private key is the decryption exponent *d*, with decryption function $\mathrm{D}(y) = y^d \bmod m$

Preliminaries:

- Bottom line — and key detail for what we're going to see today:
  - Both cryptosystems rely on (modular) exponentiation with large values and large exponents; and in both cases, the security of the system relies on the secrecy of the exponent:
    - For DH, Alice must keep *a* secret, and Bob must keep *b* secret (otherwise, an attacker/eavesdropper can determine the value g^{ab}, which is supposed to be Alice and Bob's shared secret)
    - For RSA, the decryption exponent *d* is precisely the *private* key!

Side-Channel Analysis:

- Basic idea:
  - Entirely bypass the intrinsic “mathematical” security of a cryptosystem (or in general the security of a certain software system) by observing side-effects or by-products of the computations.
  - The hope/goal is that these side-effects exhibit a correlation with the secret data (e.g., encryption keys) that can be exploited to break the system without having to break the math.
  - These types of attacks are typically suitable when an attacker (or potential attacker) has physical access to the device performing the cryptographic computations (e.g., credit or debit cards, smartphones, etc.)

Side-Channel Analysis:

- Typical side-channels:
  - Timing
  - Power consumption
  - Electromagnetic emanations (techniques are very similar to power analysis)
  - Noise
  - Heat / Infrared Thermal Imaging
  - Cache

- We'll cover some of the basic timing and power analysis existing techniques.

Side-Channel Analysis:

- Timing attacks:
  - Consider the following password validation function in C:

```
bool password_ok (char * username, char * pwd)
{
    char * actual_pwd = get_pwd_from_db (username);
    return strcmp (actual_pwd, pwd) == 0;
}
```

  - What is its *exact* execution time?
  - It depends on the user's password .... Big oops !!!

Side-Channel Analysis:

- Timing attacks:
  - The *real* cause of the problem: data-dependent optimization; things take different amounts of time depending on the (secret) data!!
  - How does an attacker exploits this to obtain the password of a given user *in a reasonable amount of time*?
  - Attacker has:
    - Access to the system that validates a password (realistic — anyone can attempt to login as root on a given machine, or to some given user's Gmail or Facebook account, etc.)
    - Unlimited amount of failed attempts to log in (not necessarily the case in real systems — web sites block access after a certain number of consecutive failed login attempts)

Side-Channel Analysis:

- Timing attacks:
  - Procedure (just to simplify the explanation, let's assume that the password contains only lowercase letters):
    - Start with a 1-character password, trying all possible letters (26 possibilities).
    - For the correct character, the password validation function will take a little longer than for the other 25 choices (one extra iteration of the strcmp function).
      - This reveals the first character of the actual password — we only have to measure execution times for the 26 passwords.
    - Now, we repeat for the second character — try 2-character passwords, fixing the first character to the correct letter that we know from the previous iteration.
    - Successively repeat the above until finding the password.

Side-Channel Analysis:

- Timing attacks:
  - **Exercise:**

    The previous procedure fails if the password validation function happens to check the lengths first, and reject if the lengths are different without bothering to check the characters.

    Modify the procedure so that the timing attack succeeds even if the password validation function does that.

Side-Channel Analysis:

- Timing attacks:
  - Runtime analysis (ECE-250 flashback!)
    - If the attacked password has *N* characters, and it takes *L* attempts to be able to measure with enough precision (say, *L* could be in the order of several thousands, or several millions), then:
    - Guessing each character takes L attempts.
    - Thus, obtaining the password takes $L\,N = O(N)$ login attempts (linear time in the size of the password).

    - A brute force attack takes $O(26^N)$ login attempts!!

Side-Channel Analysis:

- Timing attacks:
  - Though this example is not an attack on a cryptographic system, it nicely illustrates the principle:
    - Exploit data-dependent optimizations that lead to differences in execution time related to the (secret) data

Side-Channel Analysis:

- Timing attacks:
  - Countermeasures?

Side-Channel Analysis:

- Timing attacks:
  - Countermeasures?
  - Don't use strcmp to check passwords — use a custom-made string comparison function that checks every character and decides at the end.
  - Even better:
    - Don't compare passwords, but compare a cryptographic hash of the password (storing a hash of the password in the database protects against data theft, so it is a good idea to do that anyway)

Side-Channel Analysis:

- Timing attacks on modular exponentiations:
  - Goal: obtain the exponent through measurements of execution time of the exponentiation.
  - Roughly the same assumptions in terms of what the attacker has and can do.
    - Important detail: like in the previous case, it's not really the actual amount of time what matters, but *variations* in the amount of time for different, carefully selected data.

Side-Channel Analysis:

- Timing attacks on modular exponentiations:
  - General idea:
    - Guess one bit of the exponent at a time (left-to-right), and validate that guess by comparing the timing against a “mimicked” exponentiation operation done by the attacker (on identical hardware)
    - Keep in mind Kerchoff's principle (rephrased by Claude Shannon as “The attacker knows the system”) — we assume that the attacker knows the exact exponentiation algorithm and knows the exact hardware that it is running on.

Side-Channel Analysis:

- Timing attacks on modular exponentiations:
  - Recall our square-and-multiply exponentiation algorithm:

```
R ← 1
For each bit of the exponent (left to right):
    R ← R² mod m
    If bit is 1 then
        R ← R·x mod m
    end if
end for
```

Side-Channel Analysis:

- Timing attacks on modular exponentiations:
  - There are small variations in execution time related to the data — optimizations in the algorithms at very basic levels, hardware aspects such as pipeline / branch predictions, instruction scheduling, etc.
  - For the most part, those variations are random(ish); but they're definitely *correlated* to the data. (and this is the big oops in this case)

Side-Channel Analysis:

- Timing attacks on modular exponentiations:
  - Let's define the attack recursively — for an *N*-bits exponent:
    - Suppose the attacker has managed to obtain the *G* left-most bits of the exponent.
    - Attacker chooses an *N*-bit exponent consisting of the already-guessed *G* bits at the left, and then *N*−*G* random bits to the right and locally execute exponentiation with that exponent.
    - With the same value of x:
      - Measure exponentiation times for this local ("mimicked") execution.
      - Measure the execution time for the system under attack (on which exponentiation is done with the *actual* exponent)
      - For the first G iterations in the square-and-multiply, both systems are operating *with the exact same data*!
      - What about the $(G+1)^{th}$ iteration?

Side-Channel Analysis:

- Timing attacks on modular exponentiations:
  - Given the first *G* bits already determined, we try both possibilities for bit *G*+1, and we validate by computing the variance of the difference of times:

$$\mathrm{Var}\left(T_{\mathrm{A}} - T_{\mathrm{M}}\right)$$

where T_{A} denotes the execution time for the *actual* exponentiation, and T_{M} denotes the execution time for the *mimicked* exponentiation.

  - For the correct guess of bit *G*+1, we have one extra iteration that is done with the same data, so the variance is smaller!

Side-Channel Analysis:

- Timing attacks on modular exponentiations:
  - Countermeasures?

Side-Channel Analysis:

- Timing attacks on modular exponentiations:
  - Countermeasures?
  - A very obvious (but somewhat ineffective) countermeasure is adding a delay at the end of the exponentiation to make all exponentiation times equal (notice that if we add a random amount of time, we gain nothing — we average a number of measurements and we still obtain the real decryption time!)
    - Can be bypassed by requesting concurrent decryption operations (since it is idle time, then additional decryptions will proceed — the throughput would reveal the actual exponentiation time)

Side-Channel Analysis:

- Timing attacks on modular exponentiations:
  - Countermeasures?
  - Basic idea: remove control over partial results from the attacker — how?
  - Randomize the input data — how? (by that, we mean, how do we randomize the input data in such a way that we still can obtain the correct result?)

Side-Channel Analysis:

- Timing attacks on modular exponentiations:
  - A very clever technique known as *blinding* goes as follows for RSA:
    - Pick a random number r.
    - Compute r^e (recall that e is the encryption exponent)
    - Compute the inverse of r modulo m (r^{-1} mod m)
    - To compute x^d (d being the decryption exponent), we use $x \cdot r^e$ as the base for the exponentiation, instead of x, to obtain:

$$\left(x \cdot r^e\right)^d = x^d \cdot \left(r^e\right)^d = x^d \cdot r$$

    - Now, we just multiply that results times r^{-1} mod m to cancel out the factor r and obtain the required result, x^d.

Side-Channel Analysis:

- Back in 2003, David Brumley and Dan Boneh (yes, the same Dan Boneh that Prof. Ganesh has mentioned) published a successful *remote* timing attack, in which they recovered a web server's SSL private key in a matter of a few weeks.
- As a result, they convinced the OpenSSL team to enable blinding by default (it was a configuration option, but it was disabled by default).
  - If we have time, we'll go over the details (they used a different trick that made the attack much more efficient)

Side-Channel Analysis:

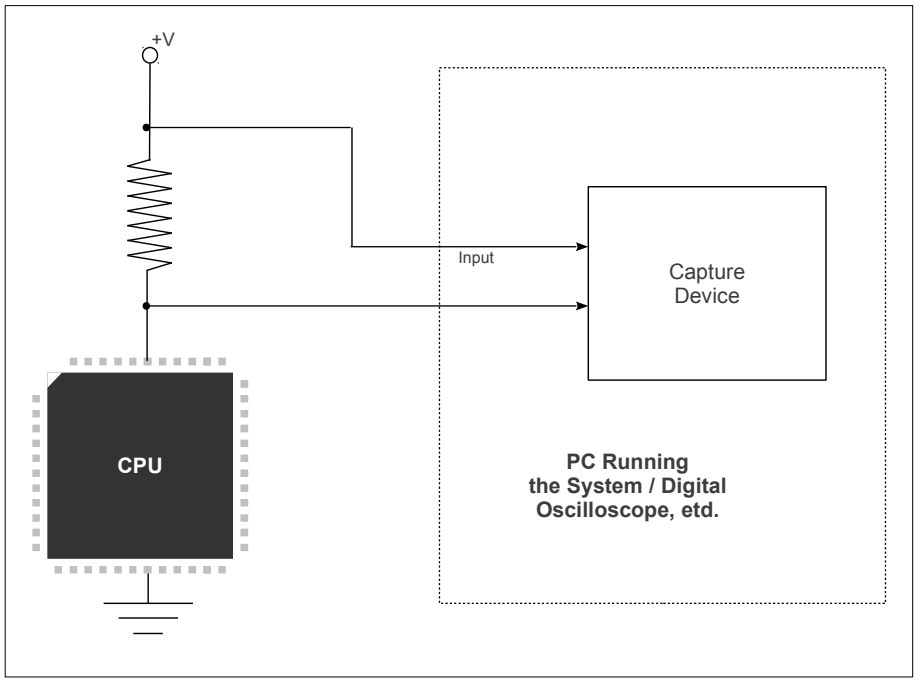
- Power Analysis:
  - Exploit relationship between the operations a CPU is doing and the data it is operating on, with power consumption.
  - Two categories:
    - Simple Power Analysis (SPA): a single power trace suffices to recover the secret data (feasible when there are data-dependent optimizations at a coarse-scale level)
    - Differential Power Analysis (DPA): multiple traces, along with statistical processing is used to “amplify” small variations in power consumption due to the data (typically applicable to SPA-resistant algorithms that execute essentially the same sequence of instructions regardless of the exponent bits)

Side-Channel Analysis:

- Power Analysis:
  - How to obtain a power trace? (i.e., a “plot” of power consumption as a function of time)

Side-Channel Analysis:

- Power Analysis:
  - How to obtain a power trace? (i.e., a “plot” of power consumption as a function of time)



Side-Channel Analysis:

- Power Analysis – SPA:
  - From Kocher et. al *Differential Power Analysis* paper (1999):

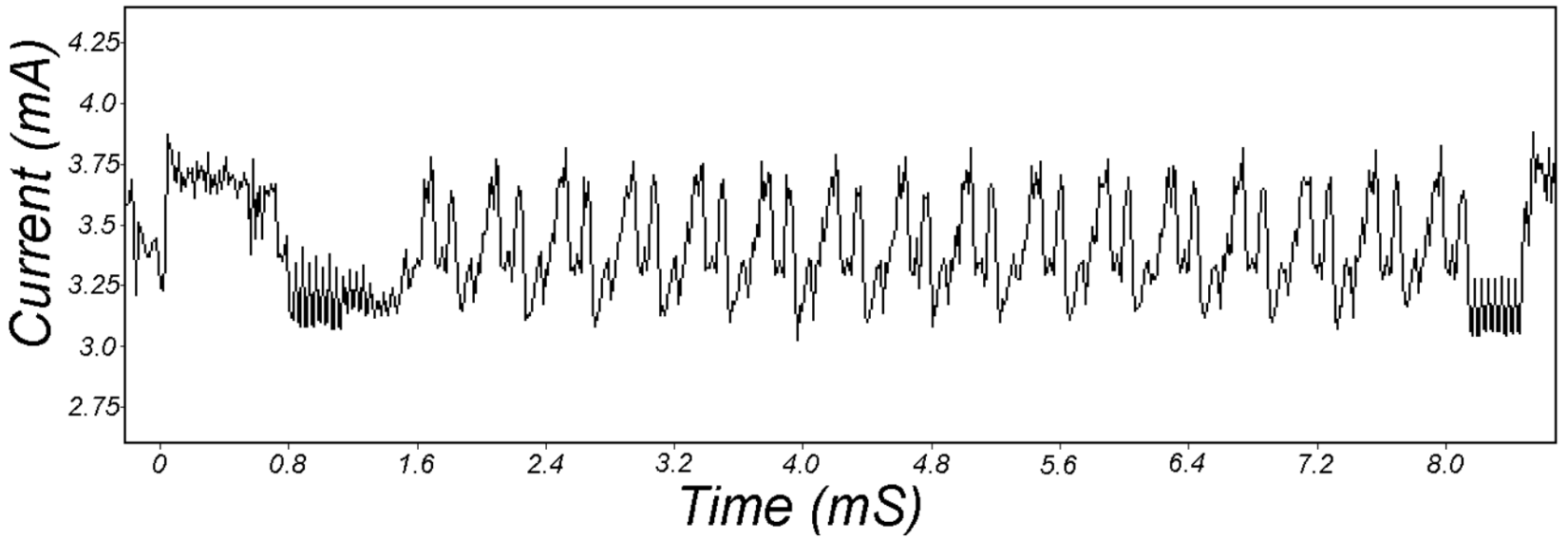


Figure 1: SPA trace showing an entire DES operation.

Side-Channel Analysis:

- Power Analysis – SPA:
  - A much more obvious example is a straightforward implementation of exponentiation (square-and-multiply):

```
R ← 1
For each bit of the exponent (left to right):
    R ← R² mod m
    If bit is 1 then
        R ← R·x mod m
    end if
end for
```

Side-Channel Analysis:

- Power Analysis – SPA:
  - A much more obvious example is a straightforward implementation of exponentiation (square-and-multiply):
  - A power trace will reveal the exponent bits in a very obvious way:
    - When the exponent bit is 0, we see a squaring operation.
    - When the exponent bit is 1, we see a squaring followed by a multiplication
    - (we assume that the operations are different, so multiplication takes longer and likely consume more power)

Side-Channel Analysis:

- Power Analysis – SPA:
  - Countermeasures?

Side-Channel Analysis:

- Power Analysis – SPA:
  - Countermeasures?
  - A quick parenthesis: does blinding work against SPA?

Side-Channel Analysis:

- Power Analysis – SPA:
  - Countermeasures?
  - A quick parenthesis: does blinding work against SPA?
  - Clearly, no — the exponent bits are the ones being observed; randomizing (blinding) the data changes nothing.

Side-Channel Analysis:

- Power Analysis – SPA:
  - Countermeasures?

Side-Channel Analysis:

- Power Analysis – SPA:
  - Countermeasures?
  - Like the old joke of the guy that goes to the doctor:
    - Patient: Doctor, it hurts when I do that
    - Doctor: Well, don't do that !!
  - Our obvious countermeasure uses the same logic:
    - Oops, I reveal the exponent bits when I conditionally execute the multiplication!
    - Well, *do not* conditionally execute the multiplication!!
      - As in: execute it unconditionally, and discard the result when the exponent bit is 0.

Side-Channel Analysis:

- Power Analysis – SPA:
  - Countermeasure: square-and-always-multiply — has to be done carefully, though (in particular, have to avoid any single if statements):

```
R ← 1
For each bit of the exponent (left to right):
    TMP[0] ← R² mod m
    TMP[1] ← TMP[0]·x mod m
    R ← TMP[bit]
end for
```

Side-Channel Analysis:

- Power Analysis – SPA:
  - Big drawback with this square-and-always-multiply?

Side-Channel Analysis:

- Power Analysis – SPA:
  - Big drawback with this square-and-always-multiply?
  - Strong performance penalty — for an *N*-bit exponent, an average of *N* / 2 unnecessary multiplications are executed!
    - And we keep in mind that multiplications are the more expensive operations in the mix !!
  - A far less obvious, but far worse drawback comes when we consider fault attacks (specifically, *induced* fault attacks)
    - Recall that we're working with situations where the attacker has physical access to the device — in many cases, the device relies on an external power supply (certainly the case with smart cards)

Side-Channel Analysis:

- (Induced) Fault Attacks:
  - The idea is that as a consequence of a hardware or software fault in a cryptographic operation, the incorrect result could end up leaking information about the secret parameters.
  - Having physical access to a device allows us to induce a hardware fault — how?
    - One typical approach consists of manipulating the power supply.
    - For example, drop the power or send a power surge for a microsecond or so, such that you scramble the bits in the ALU, or flip some bits that makes the CPU skip some operations, etc.

Side-Channel Analysis:

- (Induced) Fault Attacks:
  - This idea of fault attacks gets mixed with SPA if we naively implement the square-and-always-multiply countermeasure:
  - At the times where the multiplication is being executed, the attacker induces a fault.
  - As a consequence, the result of the multiplication will be incorrect.
    - If the exponent bit is 0, then it won't matter, because the result is being discarded.
    - Big oops: the attacker can tell whether the result was discarded (how? Encrypt the result and see if it matches the supplied data)
    - If the attacker can determine whether the result was discarded, then they know whether that particular exponent bit was 0 or 1. (so, they repeat this procedure for each exponent bit and *voilà* !)

Side-Channel Analysis:

- (Induced) Fault Attacks:
  - The approach described in the previous slide is realistic:
    - Recall again Shannon's Maxim: *The enemy knows the system*.
    - In particular, the attacker knows the exact clock rate and the exact times at which each operation happens.

Side-Channel Analysis:

- Fault Attacks in general can be more sophisticated than this — if we have time, we'll go over one of the interesting attacks (proposed by Dan Boneh et al. back in 1997).
  - That fault attack (on RSA digital signatures) uses, among other things, the same idea as the remote timing attack that we mentioned earlier.

Side-Channel Analysis:

- Back to power analysis — we'll now look at DPA.

Side-Channel Analysis:

- Differential Power Analysis (DPA).
- Basic idea:
  - Similar to the timing attack we saw earlier: power consumption is *approximately* the same regardless of the exponent (if we have an SPA-resistant exponentiation).
  - We want to exploit the small variations correlated to the data that the system is working with.
  - Like the timing attack, we guess one bit at a time (from left to right) and validate that guess through statistics:
    - Unlike the timing attack, we use correlation between power traces directly, instead of variance of an auxiliary measurement.

Side-Channel Analysis:

- Differential Power Analysis (DPA).
- Defined recursively:
  - Assuming that the attacker has gained knowledge of the first *G* bits of the exponent, and wants to determine the value of bit *G*+1:
  - The attacker can determine all of the intermediate results up until the G^{th} iteration of the square-and-multiply.
    - In particular (and this is the key detail here!), if we execute multiple times the exponentiation with the same data and different exponents with the same first *G* bits, then the power traces will be identical (fully correlated) up until iteration G, and completely uncorrelated after that.

Side-Channel Analysis:

- Differential Power Analysis (DPA).
  - A less obvious aspect: if we execute multiple exponentiations with the same exponent and select the traces for which some arbitrary bit (say, the least-significant bit) of the result has a particular value, there will be *some* correlation between the traces.
    - Same thing holds for intermediate results — if we select the traces for which a certain bit is 1 at the end of the G^{th} iteration, there will be some correlation if the first G exponent bits in all exponentiations are the same.

Side-Channel Analysis:

- Differential Power Analysis (DPA).
  - Putting the pieces together: to determine bit *G*+1 of the exponent (given that the attacker has already determined the first *G* bits), the attacker guesses a value for that bit.
  - Based on that guess, the attacker computes (predicts) the intermediate result at the end of iteration *G*+1 (doesn't have to be in identical hardware — as long as the *algorithm* is the same, things will work)
    - Partition the power traces based on the *predicted* value of the LSB of the intermediate result at the end of iteration *G*+1.
    - Multiple power traces are taken, with *random* input data, and the prediction has to be done for each operation.

Side-Channel Analysis:

- Differential Power Analysis (DPA).
  - If the guess (for the exponent bit) was correct, then the intermediate results that the attacker predicted will match *exactly* the values being computed inside the target system.
  - Thus, the power traces will exhibit some correlation during iteration G+1.
  - If the guess was incorrect, then the values predicted by the attacker are essentially “random” numbers uncorrelated from the values being computed inside the target system.
    - Thus, the partition made by the attacker is essentially a random selection of traces — there will be no correlation between the power traces that the attacker puts together.

Examples and Exercises:

Examples and Exercises:

- Diffie-Hellman key exchange protocol:
  - Toy example: p = 100 and g = 17 (what is incorrect about these parameters?)
  - Alice picks, say, 15 (but Bob doesn't know that), and Bob picks, say, 19 (and Alice doesn't know that).
  - Alice computes 17^{15} = 286242305150981579 3
  - Bob computes 17^{19} = 2390724356851513248471 53
  - Alice transmits 93 and Bob transmits 53
  - Bob computes 93^{19} = 251869756626442693778398337792895099 57
  - Alice computes 53^{15} = 731371518890286197244881 57
  - Shared secret is 57

Examples and Exercises:

- Diffie-Hellman key exchange protocol:
  - To try it yourself (for self-assessment purposes, as it is still a TOY example), you can use this silly demo I created:

    https://ece.uwaterloo.ca/~cmoreno/dhp.html

    (sorry if it the instructions sound a bit “patronizing” — I created it a long time ago for a non-technical audience :-))

Examples and Exercises:

- RSA public-key cryptosystem:
  - Allows sender to encrypt with recipient's *public* key, in a way that only the owner of the corresponding *private* key can decrypt it (no-one else can).
  - Basic idea (in general, for public-key cryptography):
    Need an operation with the mathematical property that it is easy to apply in one direction, but hard to apply the inverse function *unless* you have some additional piece of information.
    - The function is the public key; the additional piece of information is the private key.

Examples and Exercises:

- RSA public-key cryptosystem:
  - Each recipient generates their own public / private key pair.
  - For RSA:
    - Randomly select two large prime numbers, p and q
      - There are efficient algorithms to test whether a number is prime (even if the number is large).
    - Compute the modulus $m = p \cdot q$
    - Compute Euler's function $\varphi(m) = (p-1)(q-1)$
      - Note: Euler's function $\varphi(n)$ is the number of numbers relatively prime to n in the range $[1,n)$ — it's easy to see that for the product of two primes, the formula is the above.

Examples and Exercises:

- RSA public-key cryptosystem:
  - Each recipient generates their own public / private key pair.
  - For RSA:
    - Select an encryption exponent e (relatively prime to φ(m))
      - In general, the value $2^{16} + 1 = 65537$ is chosen — efficient to do exponentiation: only 17 bits, and only two of them are 1.
    - Compute the inverse of e modulo φ(m). That is, a value d such that $e \cdot d \bmod \varphi(m) = 1$
      - There's an easy and efficient procedure to do this (the so-called *Extended Euclidean Algorithm*)
    - This value d is the decryption exponent (the private key!)

Examples and Exercises:

- RSA public-key cryptosystem:
  - Toy example: $p = 7,\ q = 13$
  - $m = p \cdot q = 91$; $\varphi(91) = 6 \cdot 12 = 72$
  - We'll pick encryption exponent $e = 5$ ($2^2 + 1$ — it's prime, so it is relatively prime to 72)
  - The decryption exponent d is the inverse of 5 modulo 72 (a number between 1 and 71 such that $5 \cdot d \bmod 72 = 1$)

    We can easily verify that that number is 29 — $5 \cdot 29 = 145$, and 145 mod 72 is 1.
  - Public-key is the pair (5,91) — $E(x) = x^5 \bmod 91$
  - Private-key is 29 — $D(y) = y^{29} \bmod 91$

Examples and Exercises:

- RSA public-key cryptosystem:
  - $E(x) = x^5 \bmod 91$; $D(y) = y^{29} \bmod 91$
  - Let's try a few encryptions/decryptions:
    - Plaintext = 15:
      - $E(15) = 15^5 \bmod 91 = 759375 \bmod 91 = 71$
      - $D(71) = 71^{29} \bmod 91 =$ $48583870762480666770881138170405337679268897592532343 1 \bmod 91 = 15$
  - Try some other values (hopefully your calculator handles these large number of digits — recent versions of the calculator that comes with Linux certainly does)

Summary

- During today's lecture, we:
  - Looked at modular exponentiation, a fundamental operation for public-key cryptosystems
  - Briefly looked at Diffie-Hellman and RSA systems
  - Talked about side-channel analysis
    - Some vulnerabilities caused by data-dependent optimizations
    - Some are caused by correlation between data and leaked signals
    - Statistical processing techniques used to measure small variations, and to get around measurement noise.
  - Talked about Timing Analysis and Power Analysis (SPA and DPA)
  - Discussed some *basic* countermeasures