# Practical Cryptographic Systems

## Protocols / Side Channels I

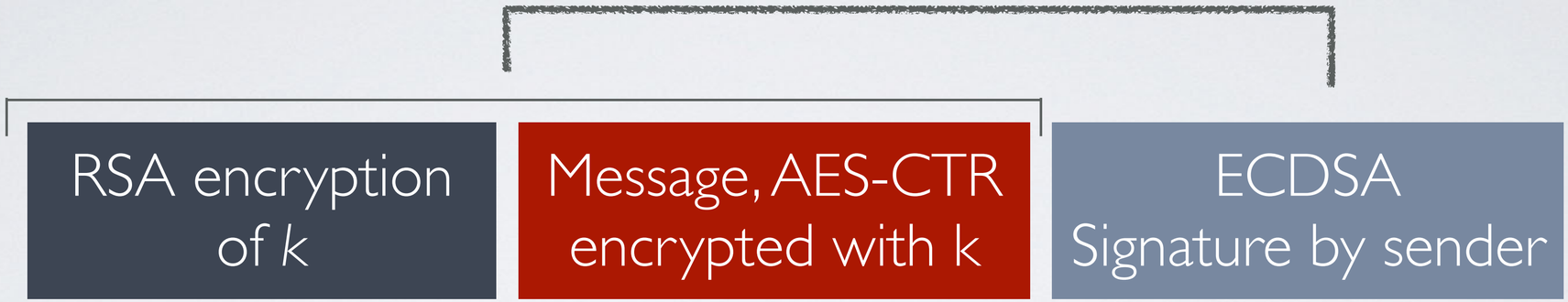
Instructor: Matthew Green

# Housekeeping

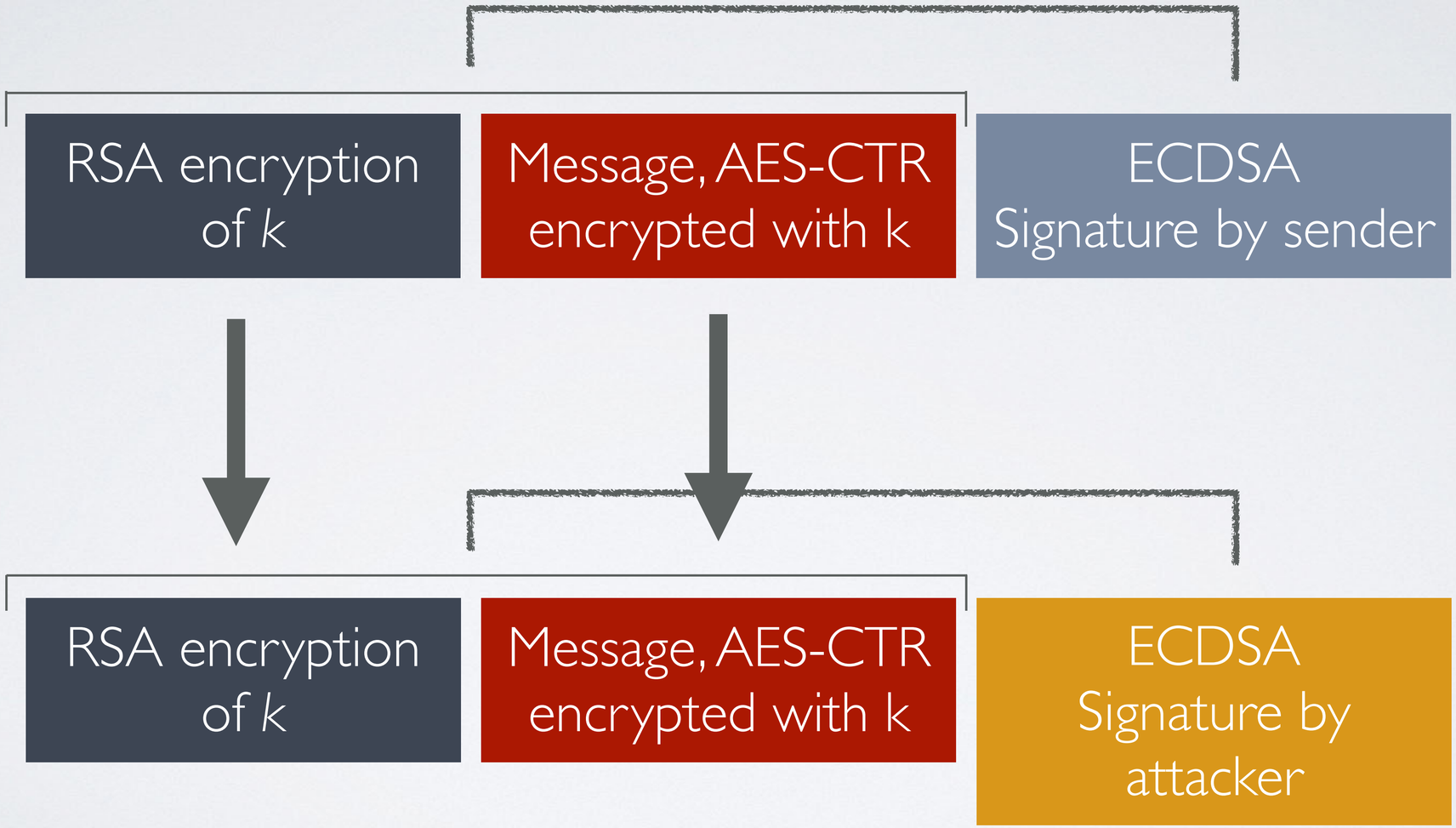
- Reading assignment (short!) out tonight
  - Really!
- **Midterm exam on March 13**
  - This will cover everything through the previous class

# News?

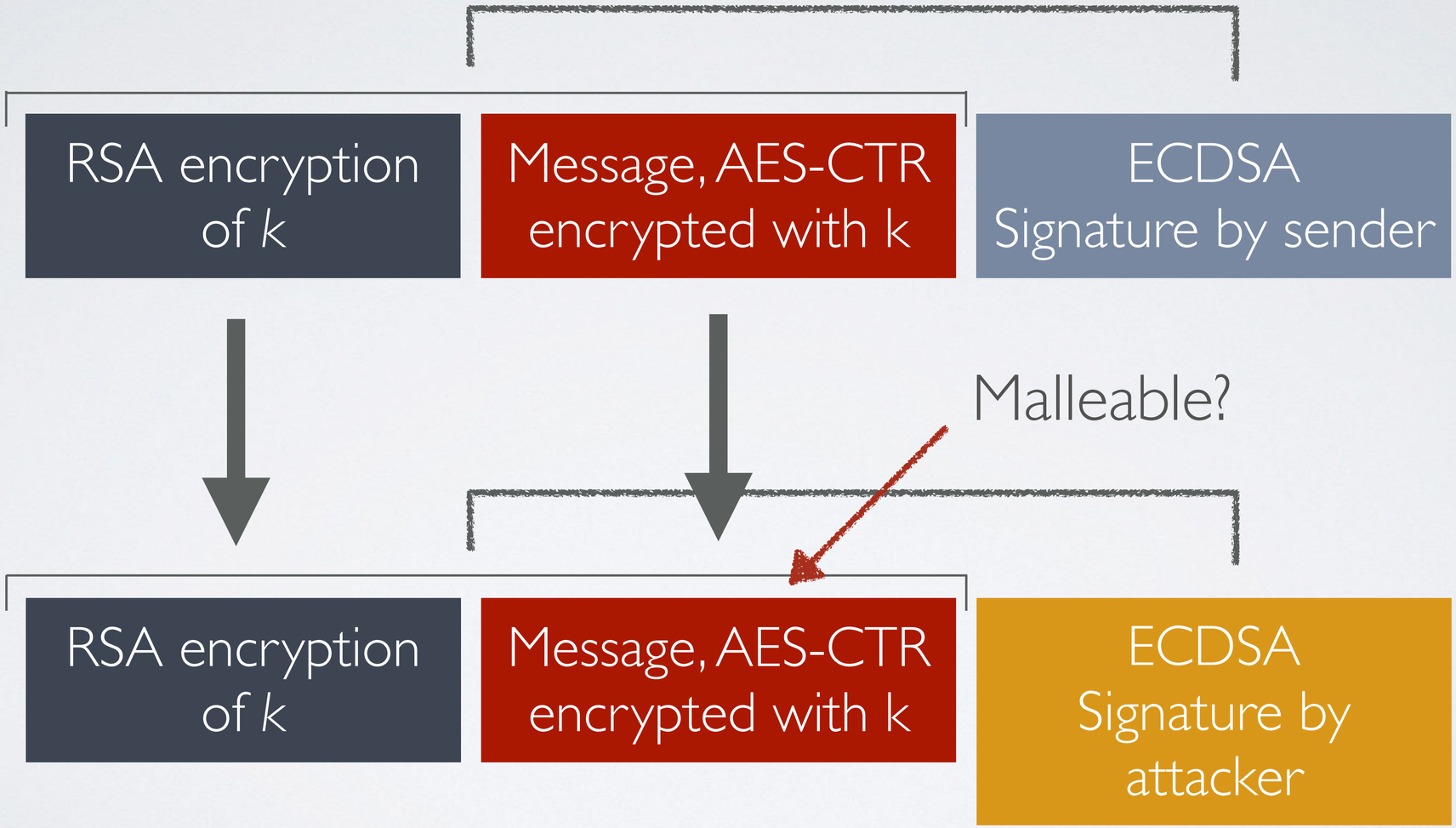
# iMessage: Encryption



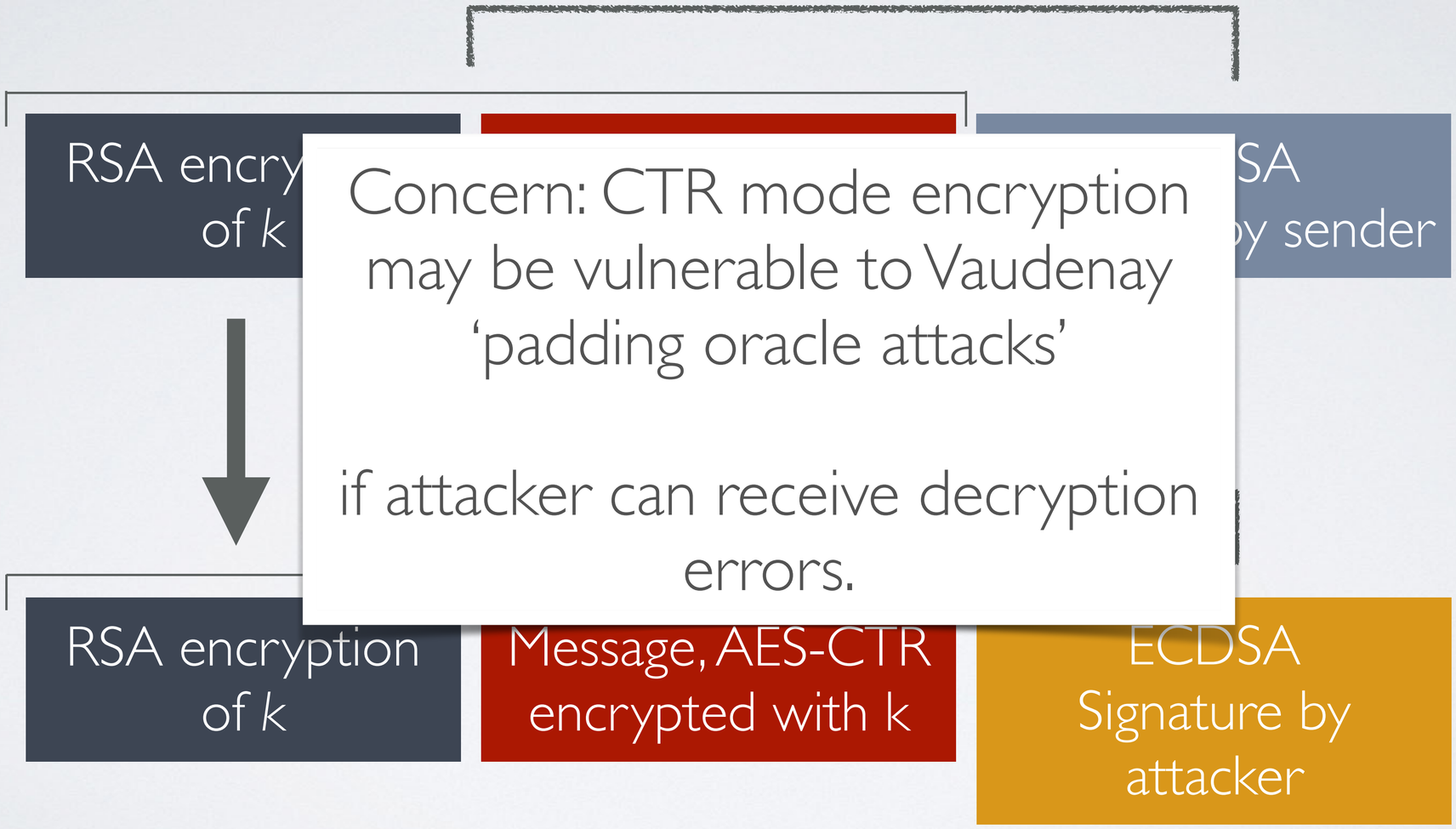
# iMessage: Encryption



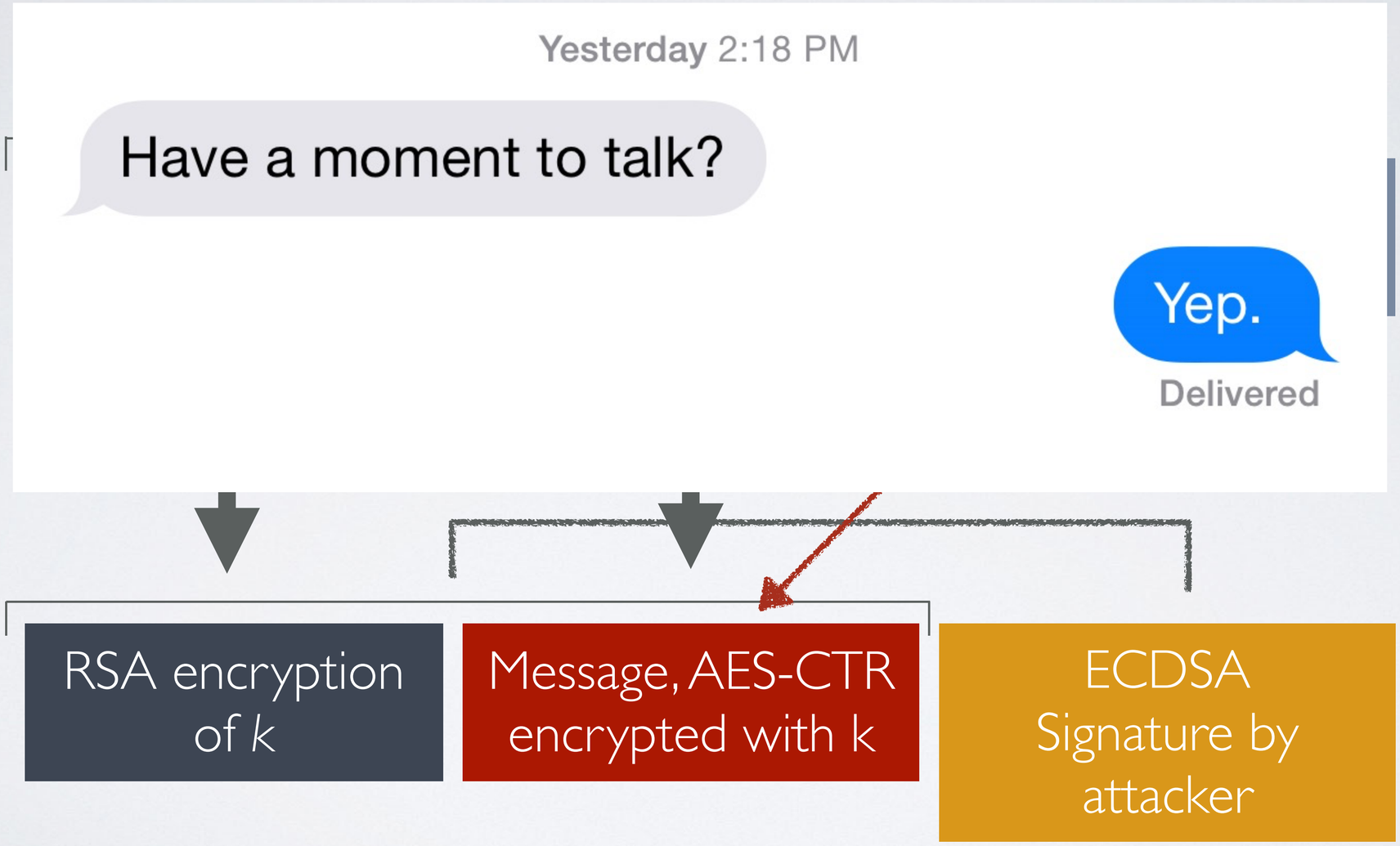
# iMessage: Encryption



# iMessage: Encryption



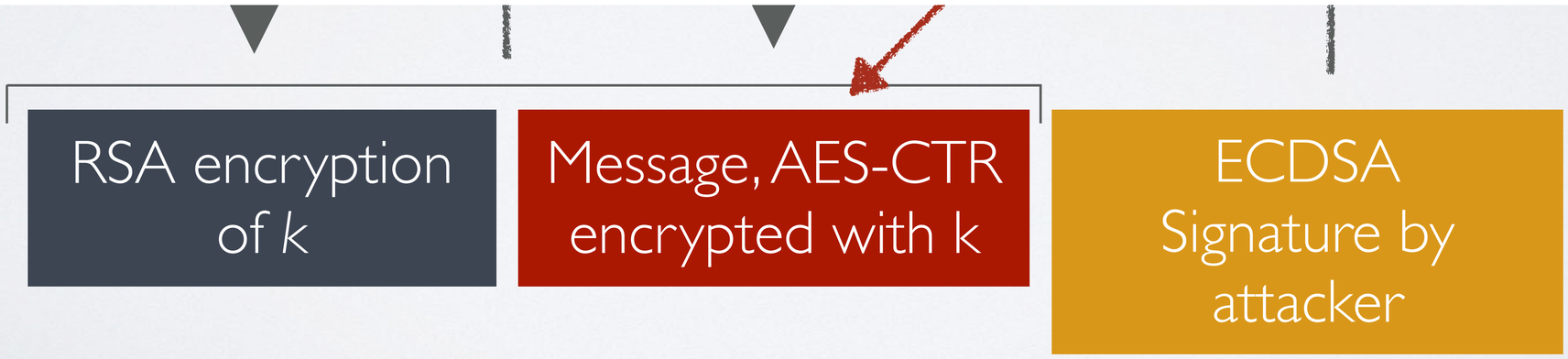
# iMessage: Encryption



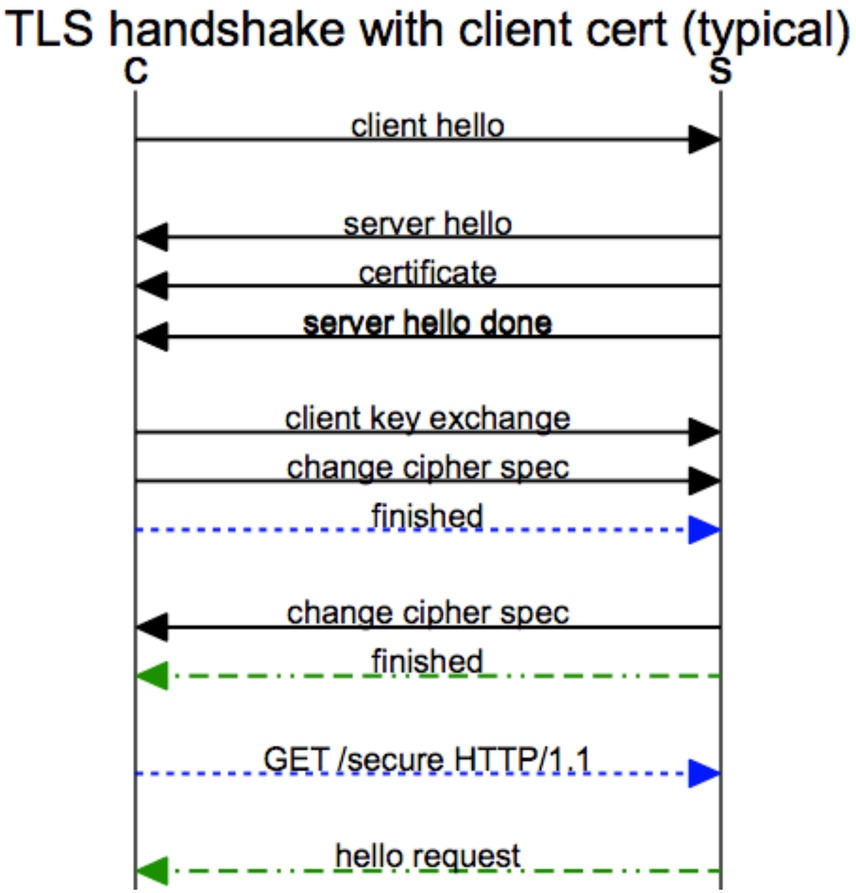
# iMessage: Encryption

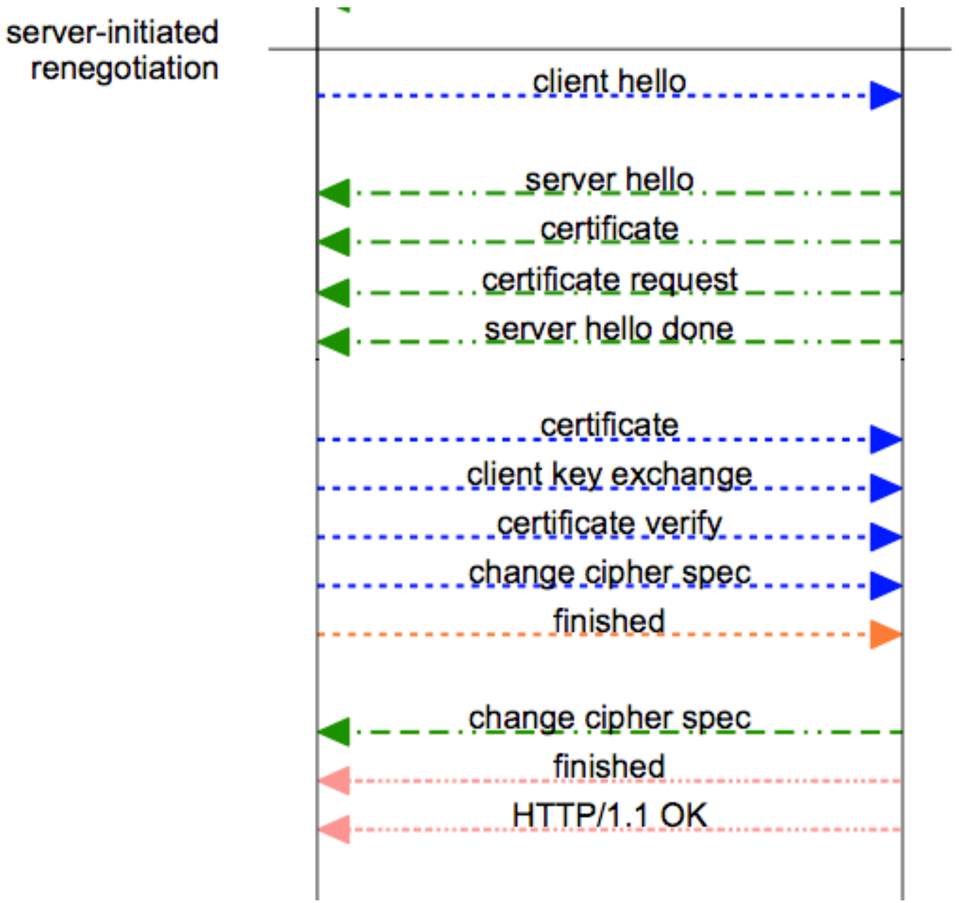
Here is an example of such a bplist:

```
D:   True
E:   'pair'
P:   <variable length binary data> (iMessage payload, deflate compressed)
U:   <128bit binary data> (iMessage UID)
c:   100
i:   <32bit integer> (messageId, same as in PUSH header)
sP:  mailto:tim_c@icloud.com  (sender URI)
t:   <256bit binary data> (sender Push-Token)
tP:  mailto:mark_z@facebook.com (receiver URI)
ua:  [Mac OS X,10.8.5,12F37,MacBookPro10,2] (sender OS and hardware version)
v:   1
```



# Cipher spec


TLS handshake with client cert (typical)
C
S
client hello
server hello
certificate
server hello done
client key exchange
change cipher spec
finished
change cipher spec
finished
GET /secure HTTP/1.1
hello request


server-initiated
renegotiation
client hello
server hello
certificate
certificate request
server hello done
certificate
client key exchange
certificate verify
change cipher spec
finished
change cipher spec
finished
HTTP/1.1 OK

# DECT

- Digital Enhanced Cordless Telephone protocol
  - European standard, now in US
  - Interoperable devices
  - Connects base station (FT) to handsets (PT)
- Tools:
  - DECT Standard Cipher (DSC)
  - DECT Standard Authentication Algorithm (DSAA)

# DECT

- Step 1: Pairing
  - User enters a 4-digit PIN into handset and base
  - Base generates a 64-bit seed, combined with PIN to generate shared key (UAK)
  - Base and handset conduct challenge/response handshake


Total entropy of UAK:
77.288 bits (64-bit seed + PIN)
Much less if PRNG is bad!

# DECT

- Step 2: Authentication
  - Two commended one:


*AS = A11(UAK, RS)*

*k, SRES = A12(AS, RAND_F)*

*RS, RAND_F*

*SRES*

In common mode, only the handset is authenticated!


*AS = A11(UAK, RS)*
*k, XRES = A12(AS, RAND_F)*

*SRES == XRES?*

Lucks, Schuler, Tews, Weinmann, Wenzel: Attacks on the DECT authentication mechanisms

# DECT Attack

- Step 2: Authentication
  - Two protocols, recommended one:


*RS, RAND_F*

*SRES*

Switch to Encrypted Mode?

Nope.


*AS = A11(UAK, RS)*
*k, SRES = A12(AS, RAND_F)*

*AS = A11(UAK, RS)*
*k, XRES = A12(AS, RAND_F)*

*SRES == XRES?*

Lucks, Schuler, Tews, Weinmann, Wenzel: Attacks on the DECT authentication mechanisms

# DECT, other

- A11, A12 built from weak cipher
  - Authors show how this cipher can be inverted using some clever attacks
  - Leaves room for attacks
    even if protocol bug fixed
  - Eerily reminiscent of GSM...

•Weak protocols

•Weak homebrew ciphers

# Example: DTCP

- BluRay & HD-DVD Disks
  - Contains “protected” area that can’t be read using normal Drive protocol
  - Embeds secret “Binding Nonce”
- 

# DTCP Protocol

- Digital Transmission Content Protection
  - Runs between Drive and Host
  - Encrypts & Authenticates Communications


# DTCP

- One layer of protection for HD-DVD/BluRay
  - Encrypts/authenticates content traversing unprotected bus lines

Attacker



# DTCP AKE

- Authenticated Key Exchange
  - EC Diffie-Hellman Protocol
  - Each device has a certificate & secret key
  - Devices also have a certificate revocation list, to prevent communication with hacked devices

# DTCP AKE (v1.4)

$r_A, Cert_A$

$r_B, Cert_B$

$g^a, Text_A, Sign_{sk_A}(r_A, g^a, Text_A)$

$g^b, Text_B, Sign_{sk_B}(r_B, g^b, Text_B)$

Compute shared key

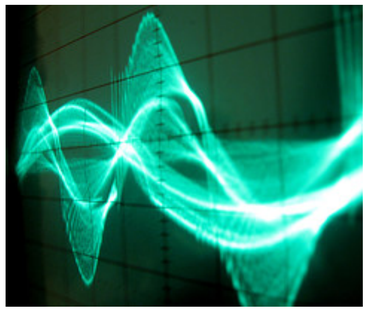
Verify Shared Key (Localization)

Talk!

# Side Channels

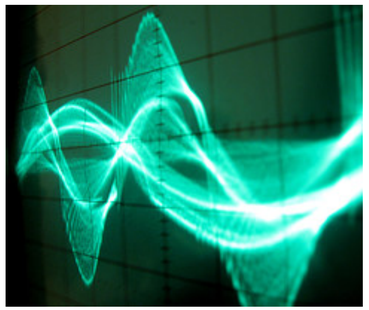
# Side Channels

```
func compareMac(macReceived []byte, macExpected []byte) {
    if (macExpected.Equal(macReceived) == false {
      return false
    } else {
      return true
    }
}
```



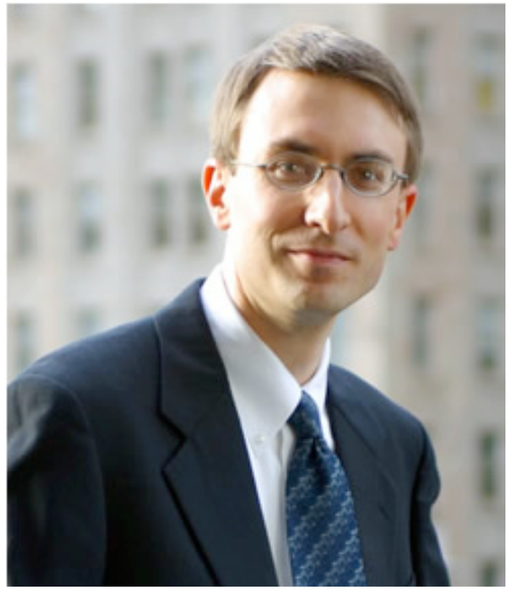
# Side Channels

- Some history:
  - 1943: Bell engineer detects power spikes from encrypting teletype
  - 1960s: US monitors EM emissions from foreign cipher equipment
  - 1980s: USSR places eavesdropping device inside IBM Selectric typewriters



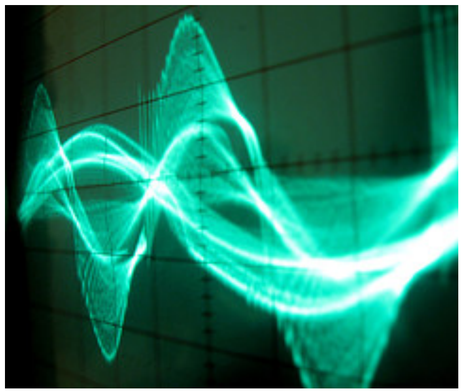
# Side Channels

- Some history:
  - 1990s: Paul Kocher demonstrates timing attacks, power analysis attacks against RSA, Elgamal



# Common Examples

- Timing
- Power Consumption
- RF Emissions
- Audio

# RSA Cryptosystem

# RSA Cryptosystem

# Modular Exponentiation

$$m = c^d \ mod \ N$$

$$d = 10101010011101010110010010010 01$$

```
modpow(c, d, N) {

    result = 1;

    for (i = 1 to |d|) {
        if (the ith bit of d is 1) {
            result = (result * c) mod N;
        }

        c = c² mod N;
    }

    return result;
}
```

# Modular Exponentiation

$$m = c^d \bmod N$$

$$d = 10101010011101010110010010010001$$

```
modpow(c, d, N) {

    result = 1;

    for (i = 1 to |d|) {
        if (the ith bit of d is 1) {
            result = (result * c) mod N;
        }

        c = c² mod N;
    }

    return result;
}
```

# Modular Exponentiation

$$m = c^d \bmod N$$

$$d = 10101010011101010110010010010 01$$

```
modpow(c, d, N) {

    result = 1;

    for (i = 1 to |d|) {
        if (the ith bit of d is 1) {
            result = (result * c) mod N;
        }

        c = c² mod N;
    }

    return result;
}
```

# Modular Exponentiation

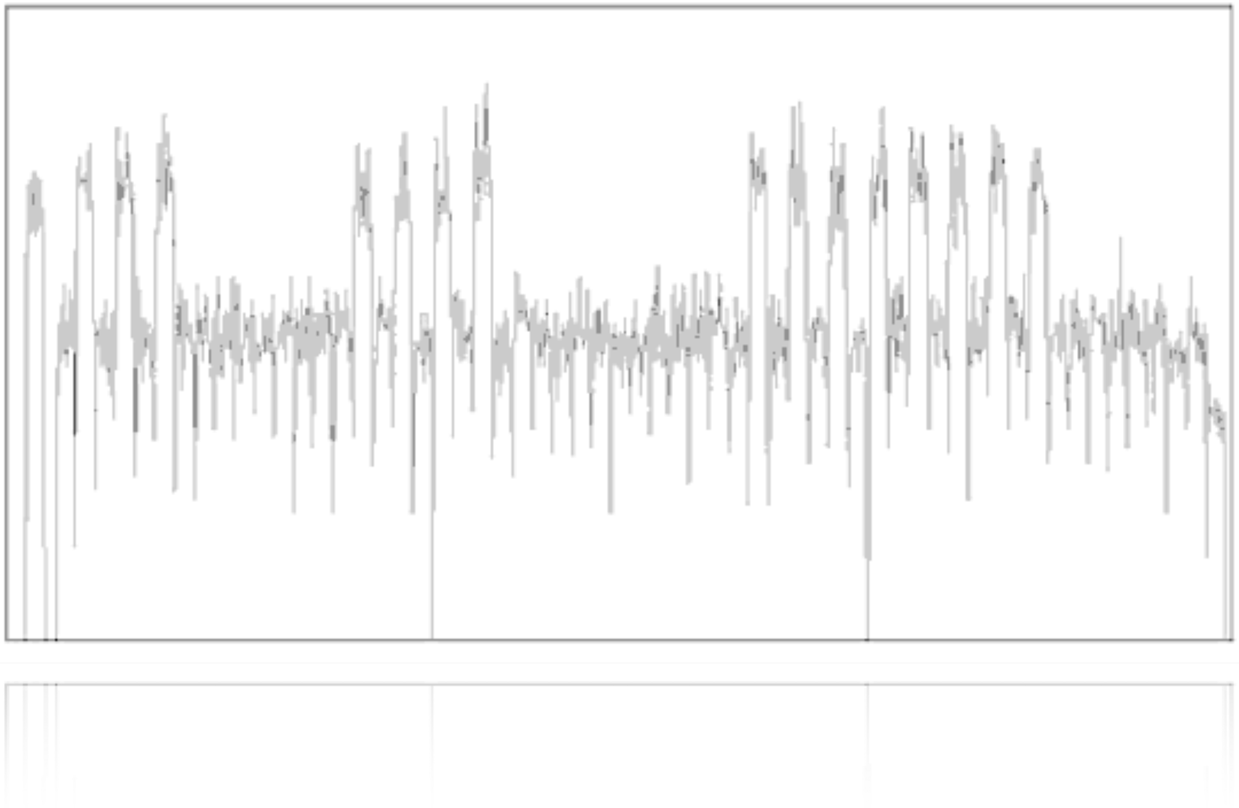
$$m = c^d \bmod N$$

$$d = 10101010011101010110010010010001$$

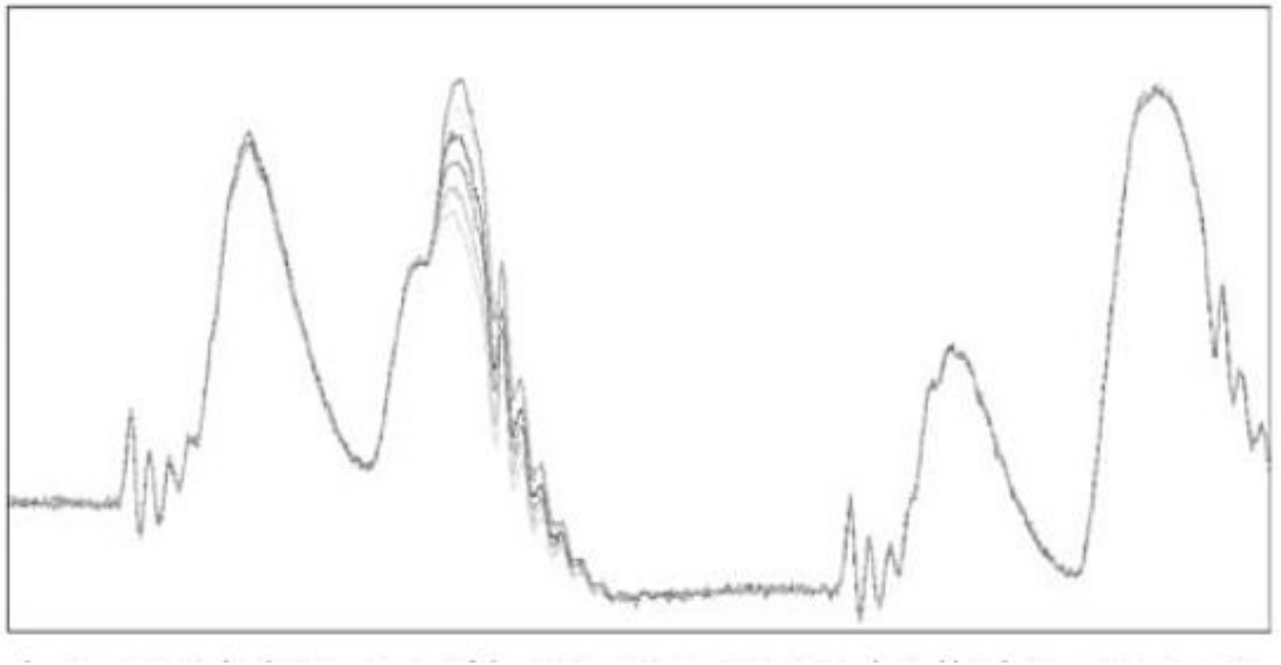
```
modpow(c, d, N) {

    result = 1;

    for (i = 1 to |d|) {
        if (the ith bit of d is 1) {
            result = (result * c) mod N;
        }

        c = c² mod N;
    }

    return result;
}
```

Expensive Operation

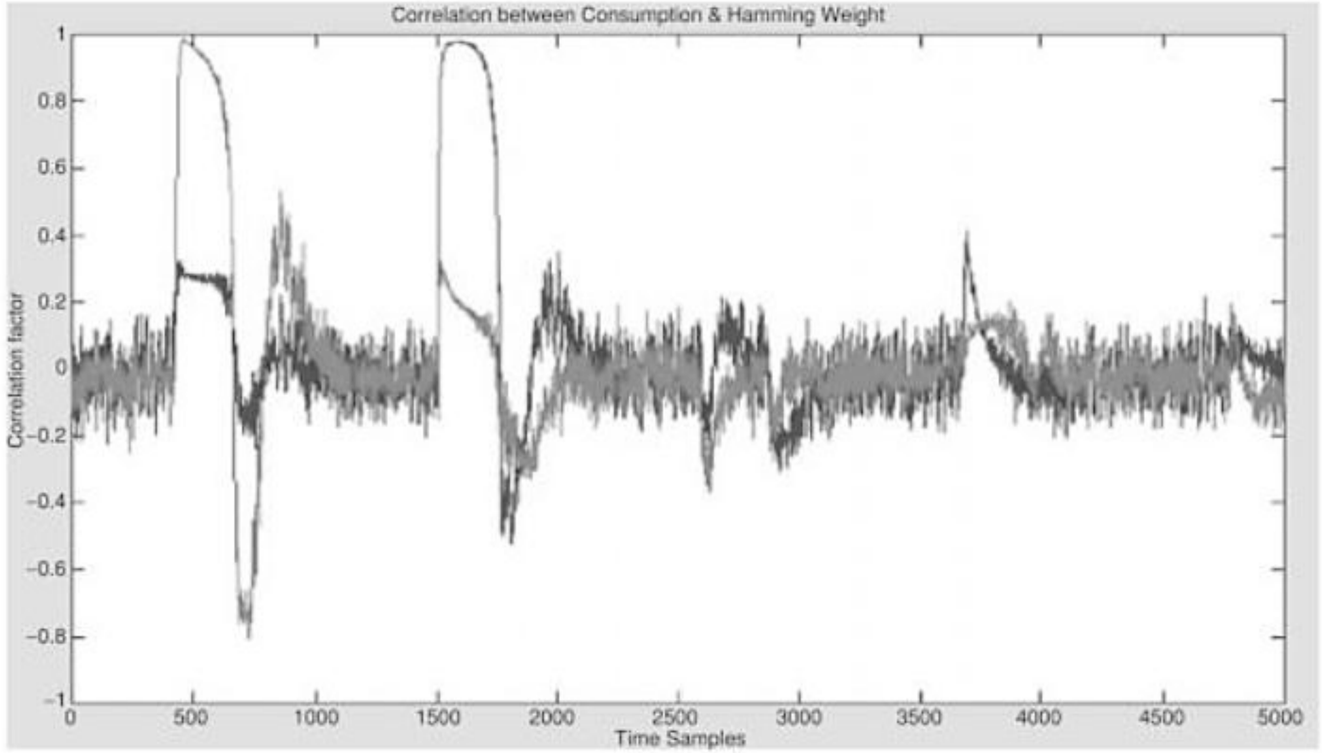
# Simple Power Analysis



# Differential Power Analysis



# Differential Power Analysis



# Modular Exponentiation

$$m = c^d \ mod \ N$$

$$d = 10101010011101010110010010010001$$

```
modpow(c, d, N) {

    result = 1;

    for (i = 1 to |d|) {
        if (the ith bit of d is 1) {
            result = (result * c) mod N;
        }

        c = c² mod N;
    }

    return result;
}
```

Expensive Operation

# Modular Exponentiation

$$m = c^d \bmod N$$

$$d = 10101010011101010110010010010001$$

```
modpow(c, d, N) {

    result = 1;

    for (i = 1 to |d|) {
        if (the ith bit of d is 1) {
            result = (result * c) mod N;
        }

        c = c² mod N;
    }

    return result;
}
```

Expensive Operation

# Modular Exponentiation

$$m = c^d \bmod N$$

$$d = 10101010011101010110010010010001$$

```
modpow(c, d, N) {

    result = 1;

    for (i = 1 to |d|) {
        if (the ith bit of d is 1) {
            result = (result * c) mod N;
        }

        c = c² mod N;
    }

    return result;
}
```

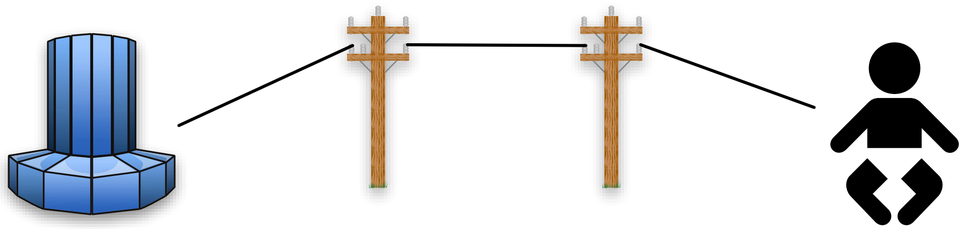
Expensive Operation

# Kocher’s Timing Attack

- Assume that for some values of (a * b),
  multiplication takes unusually long
- Given the first b bits, compute intermediate value
  “result” up to that point
- If the next bit of d = 1, then calc is (result * c)
- If this is expected to be slow,
  but response is fast then the next bit of d != 1

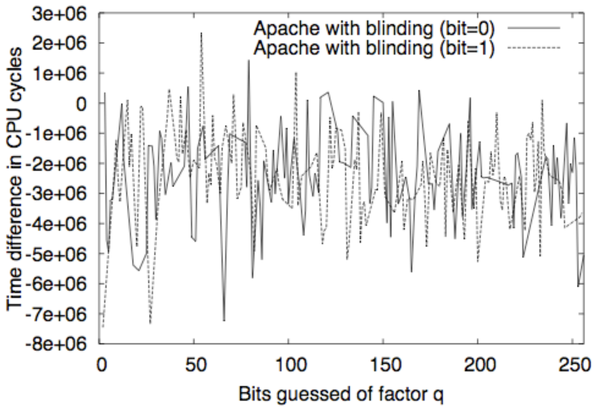
# Remote Timing Attacks

- Boneh & Brumley
  - Remote attack on RSA-CRT as implemented in OpenSSL
  - Optimization, uses knowledge of *p,q*



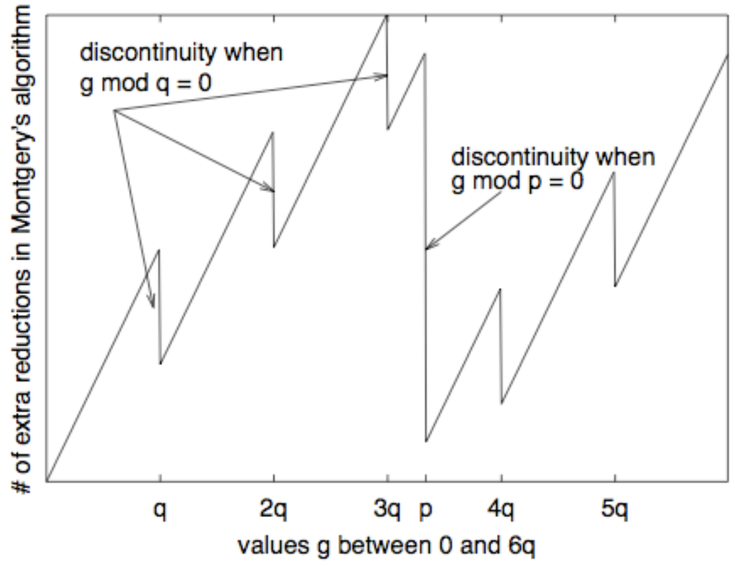
# Solutions

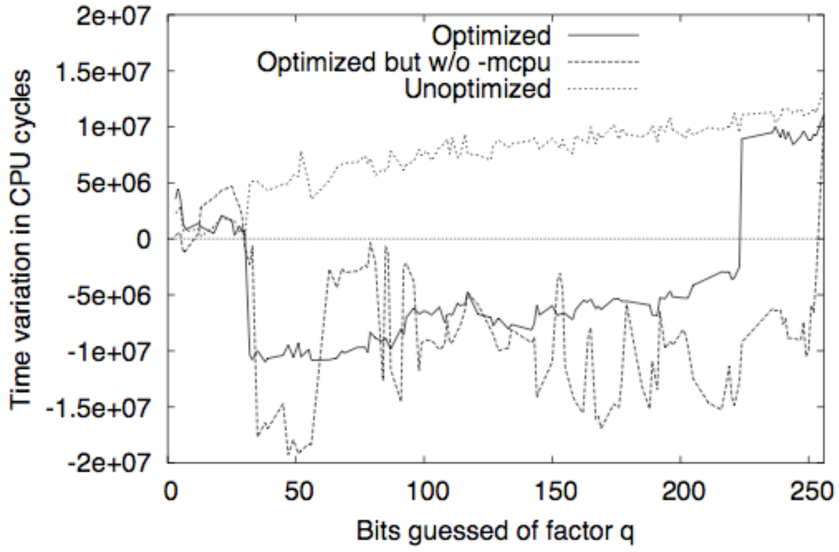
- Quantization:
  - All operations take the same time
  - Hard to do without sacrificing performance
- Blinding:
  - Prevents attacker from selecting ciphertext (that is processed with the secret key)



# Remote Timing Attacks

- Boneh & Brumley
  - By repeating the timing measurements, they were able to extract a secret key after several million samples



# Windowed Exponentiation

- Handles the input in larger “chunks”
  - Fixed or variable sized
  - Can speed up by pre-computing some values

•e.g., c^2, c^3, ..., c^16

# Windowed Exponentiation

$c^d$

0100

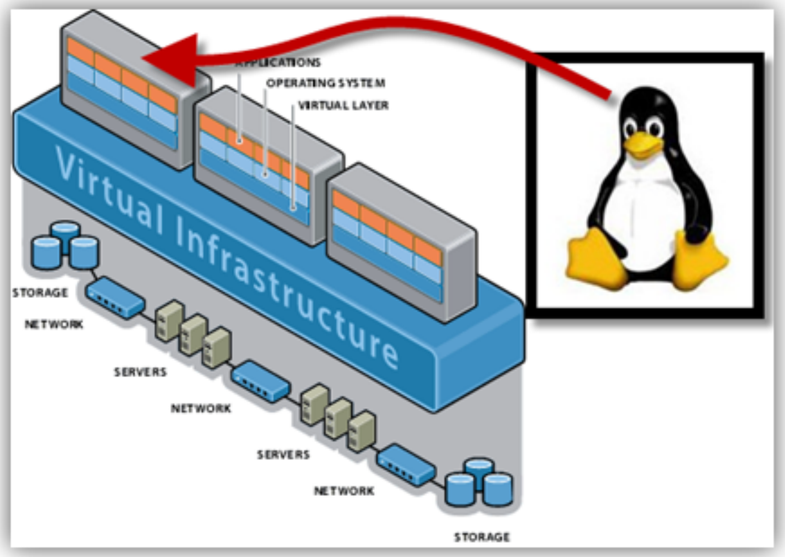
# Windowed Exponentiation

$c^d$

0010

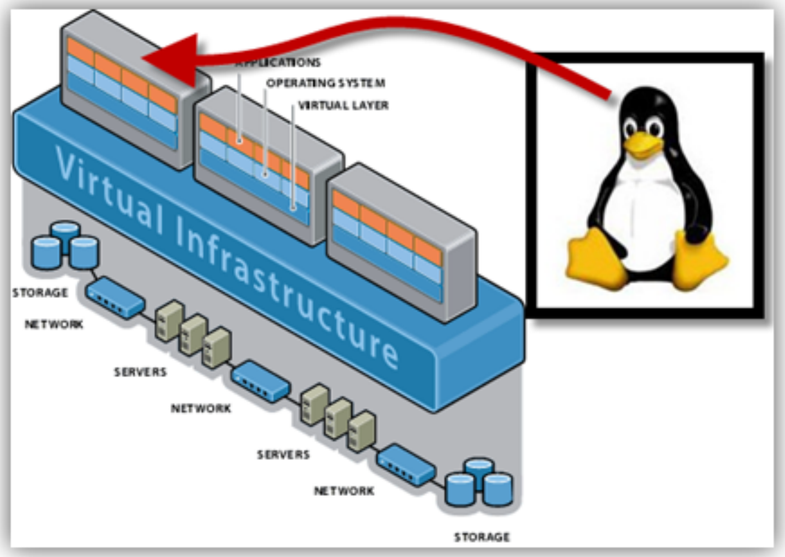
# Cache Timing Attacks

- Observation
  - When we use pre-computed tables, this implies a memory access
  - This takes time
  - Unless the data is cached!



# AES implementation

- Observation
  - When we use pre-computed tables, this implies a memory access
  - This takes time
  - Unless the data is cached!

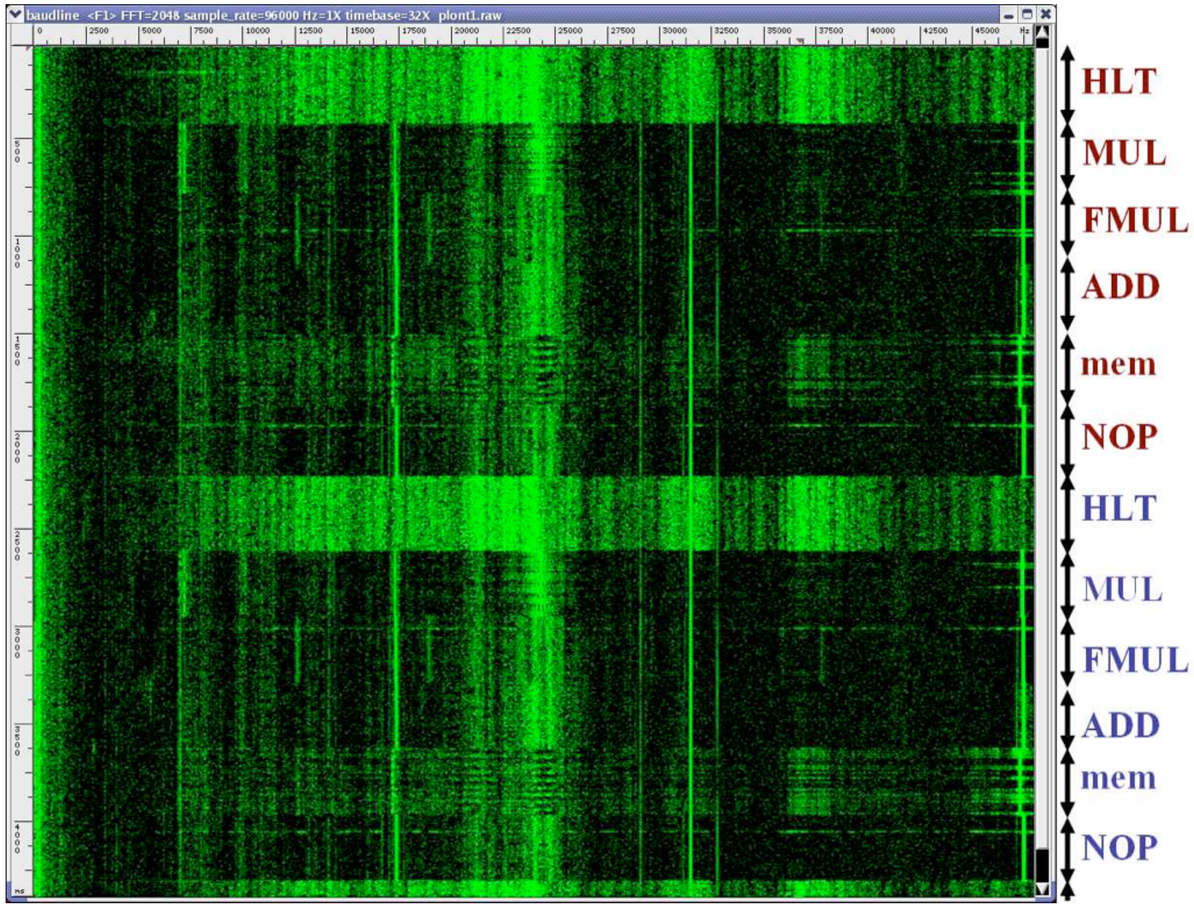


# Hypervisor attacks

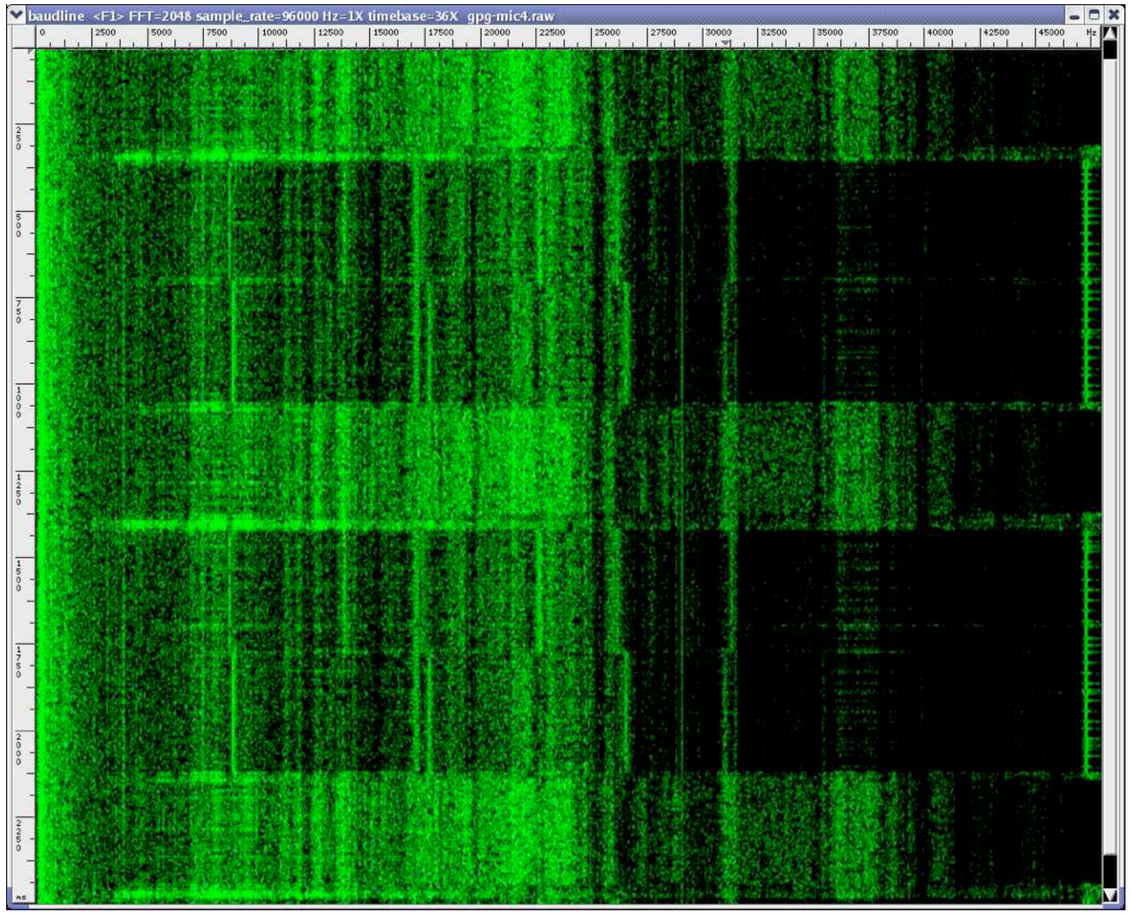
- Observation
  - This applies to code too!

```
SquareMult(x, e, N):
  let e_n, ..., e_1 be the bits of e
  y ← 1
  for i = n down to 1 {
    y ← Square(y)                (S)
    y ← ModReduce(y, N)          (R)
    if e_i = 1 then {
      y ← Mult(y, x)             (M)
      y ← ModReduce(y, N)        (R)
    }
  }
  return y
```

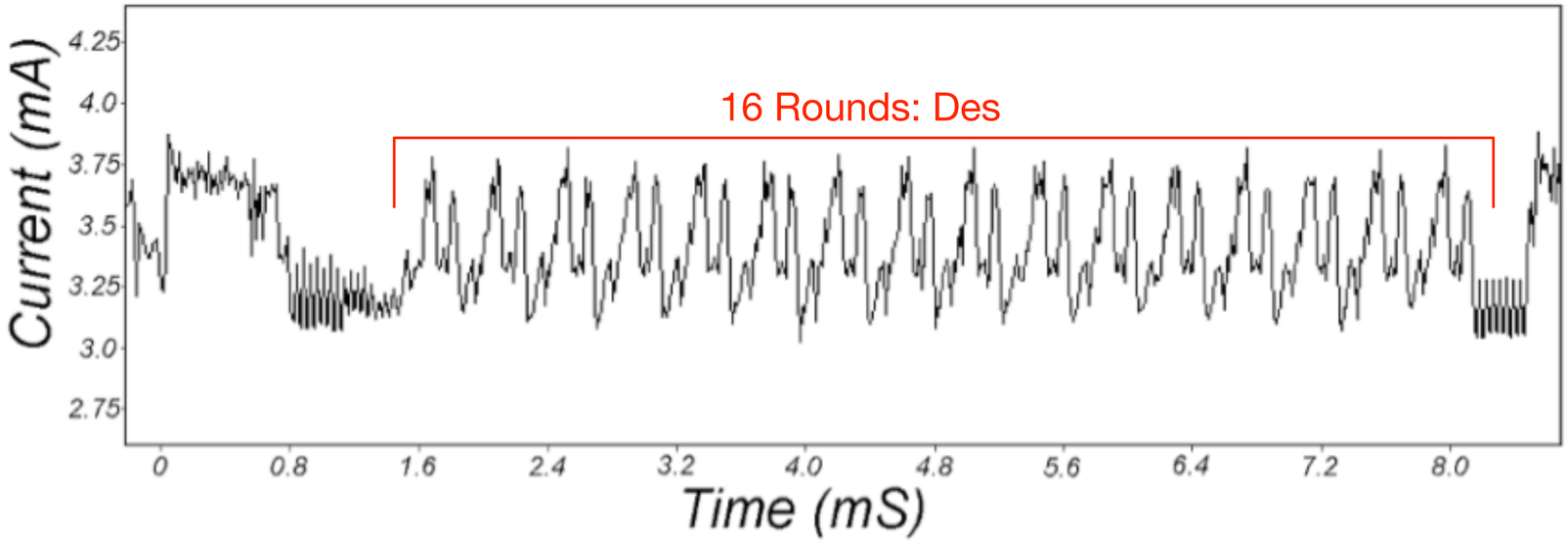
# Acoustic Side Channels



# Acoustic Side-Channels



# Reverse Engineering



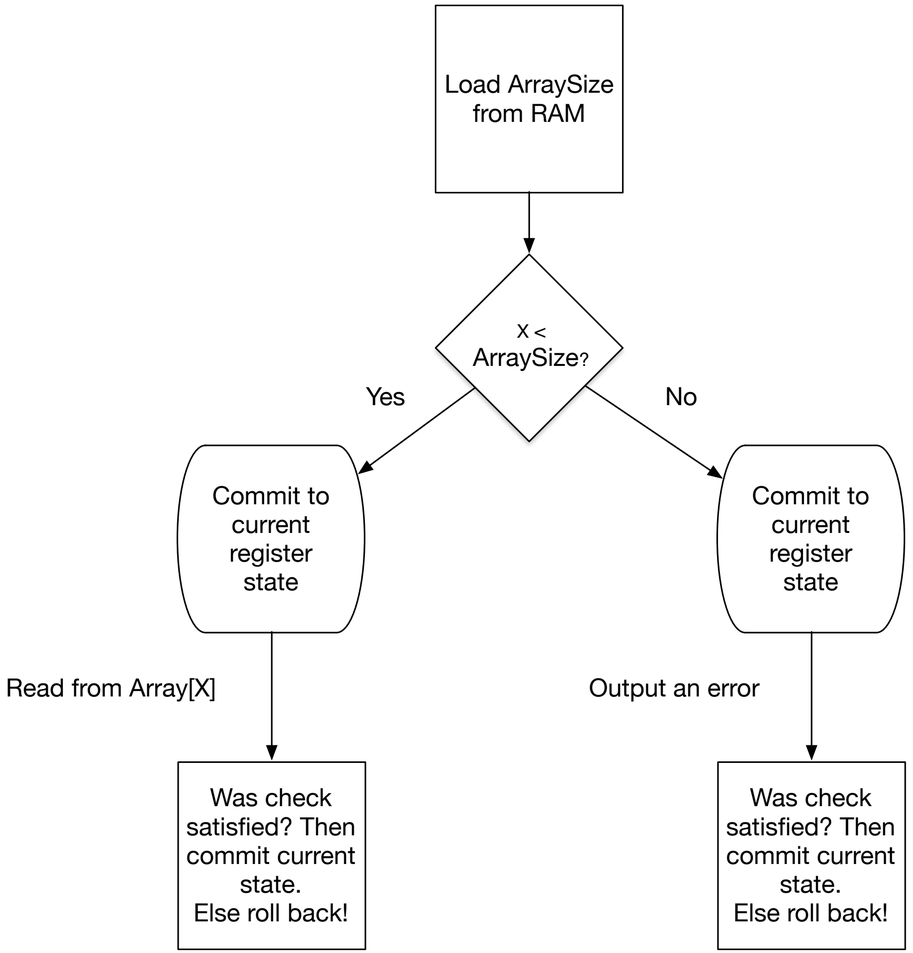
# Speculative Execution

- Programs often branch, based on values drawn from RAM
  - RAM accesses are slow
  - Modern processors could wait, but this would waste cycles
  - So instead, they “guess” the right value and perform speculative execution

# Speculative Execution

```
If (X < ArraySize) {
  y = array2[array[X] * 4096];
} else {
  // output an error
}
```

# Speculative Execution

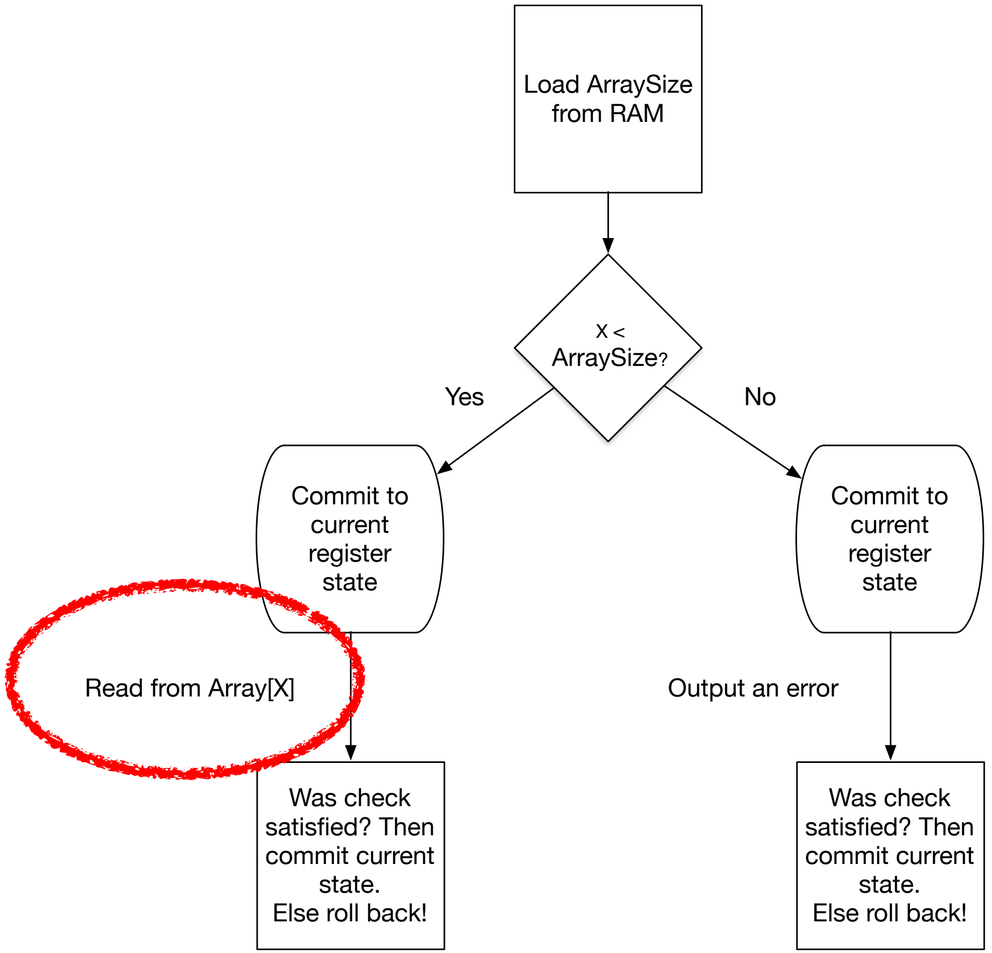


# Speculative Execution

```
If (X < ArraySize) {
  y = array2[array[X] * 4096];
} else {
  // output an error
}
```

* X is attacker controlled
* ArraySize is in RAM
* array[X] is in *cache*
* Processor must speculate on (X < ArraySize) without knowing that value.

# Speculative Execution



# Spectre

- In theory, rollback “eliminates”
  the results of a speculative execution
  that was not supposed to run
- In practice, however, processors are complicated…
- Calculation involving invalid X may affect cache state
  - Specifically, cache lines associated with array2[X*4096] will be affected by branch

# Spectre

- 1. Train branch predictor on many valid
  2. “Evict” ArraySize and array2 from cache
  3. Pass in an attacker controlled X
     that is well past the array, where
     array[X] is in cache
  4. Examine which cache lines have been
     affected

```
If (X < ArraySize) {
  y = array2[array[X] * 4096];
} else {
  // output an error
}
```

# Who chooses the branch?

- This is done by a “branch predictor”
- The predictor has memory, and remembers which branch it chose in a program it has run before
- It can be “trained” by an attacker to prefer one branch, regardless of the data it gets

# Spectre (variant 2)

- Speculative execution can do more than "if/then" branches
- For example, if you jump to an address, and the address isn't in cache, the predictor will try to "guess" which code should run
- It does this based on past experience
- This experience may have been derived from the execution of other processes

# Spectre (variant 2)

- The actions of one program/process influence the branch predictor's behavior in another
- If Process A (attacker controlled) repeatedly indirect-branches to memory address X,
  then Process B (victim) encounters a branch with X, the predictor will prefer that branch.
- So attacker can "train" the predictor in Process A to affect Process B.

# Spectre (variant 2)

- Identify a gadget G in the victim program
  - This is a chunk of code that e.g., accesses some data referred to by a specific register, uses this to make a second read
- Attacker process “trains” the predictor so that it will jump to the address of the Gadget in the attacker process
  - (This means the predictor may do the same thing in the victim process as well.)
- Attacker sends attacker-controlled to victim program
-

# Intel SGX