Practical Cryptographic Systems

Side Channel Attacks

Housekeeping:

- Quiz next Weds
  - Side channels
  - Section 4.3.2 of Boneh/Shoup (side channels)
  - Remote Timing Attacks are Practical

News

News

Another Update on the Situation at Indiana University

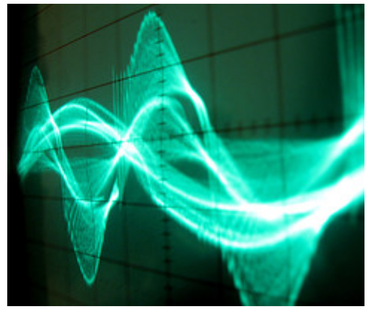
By Josh Marshall | March 31, 2025 10:37 a.m.

Send comments and tips to talk at talkingpointsmemo dot com. To share confidential information by secure channels contact me on Signal at joshtpm dot 99 or via encrypted mail at joshtpm (at) protonmail dot com.

I wanted to provide a quick update on the case of Professor Xiaofeng Wang at Indiana University. For overview details, see the posts below. The latest is the IU chapter of a faculty organization (the American Association of University Professors) has sent a letter to the university challenging Professor Wang's termination. You can see that letter here. The letter

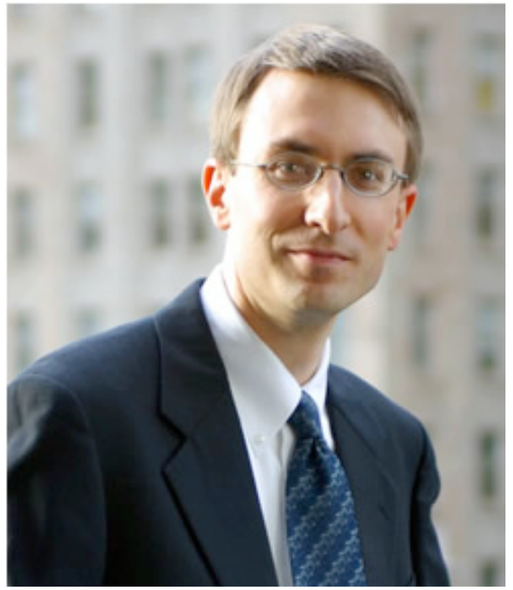
Side Channels

- 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



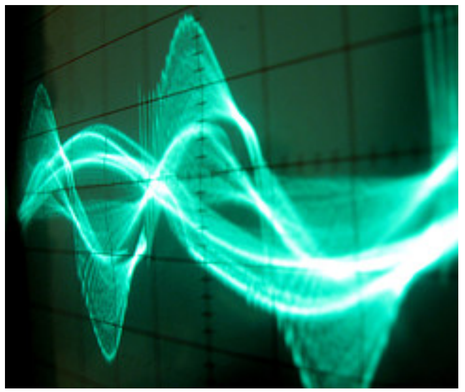
Side Channels

- 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

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

$$d = 10101010011101010110010010010001$$

```
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

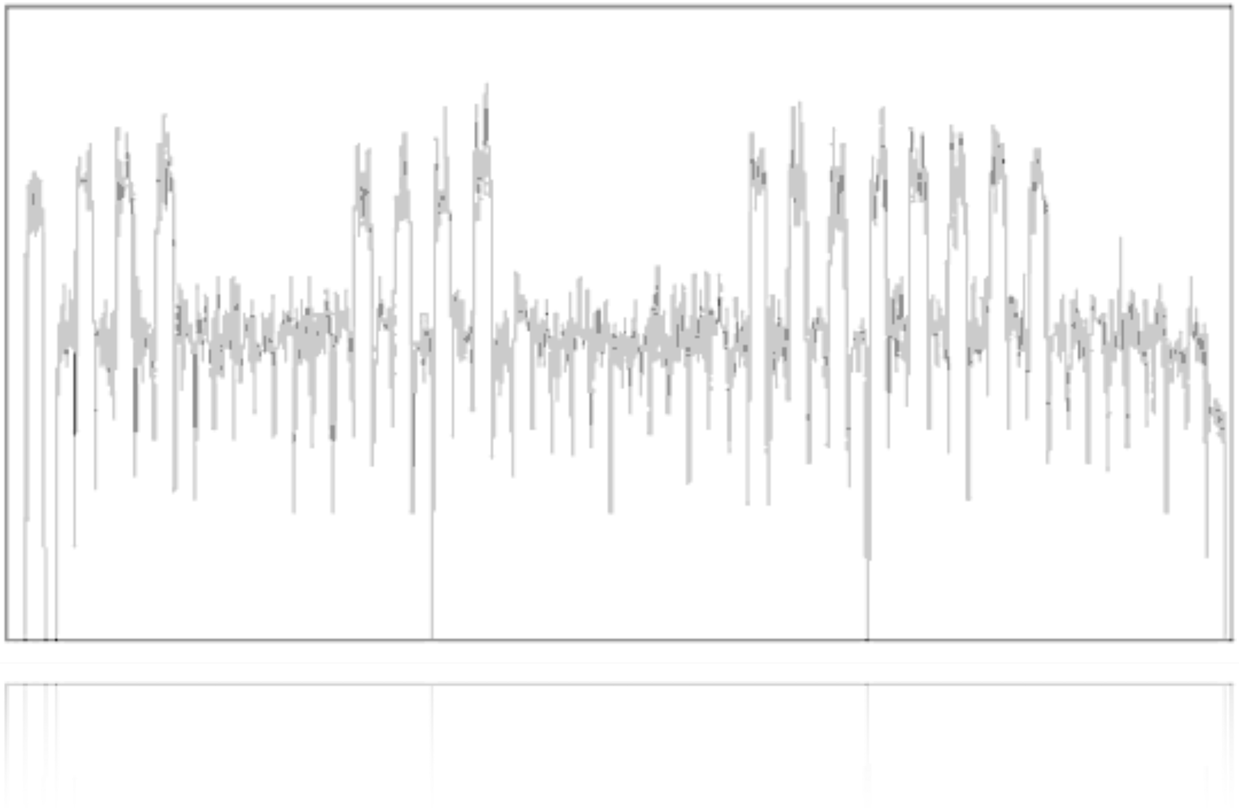
$$m = c^d \bmod N$$

$$d = 10101010011101010110010010010 01$$

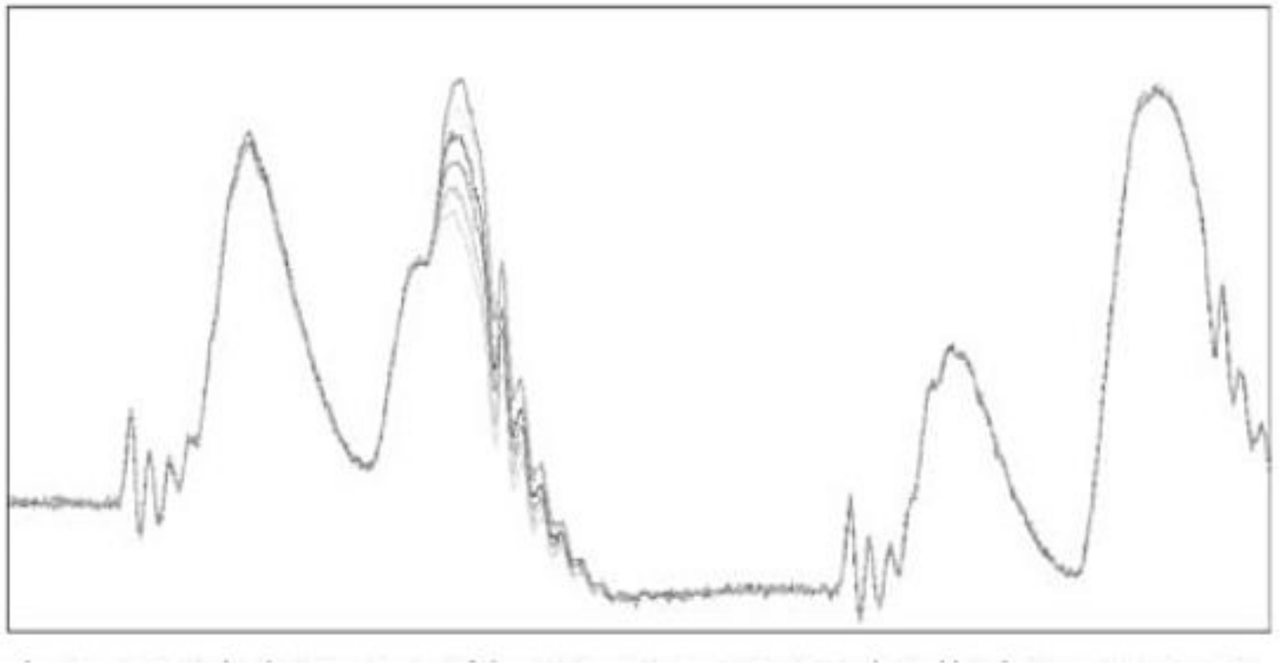
```
modpow(c, d, N) {

    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;
}
```

Expensive Operation

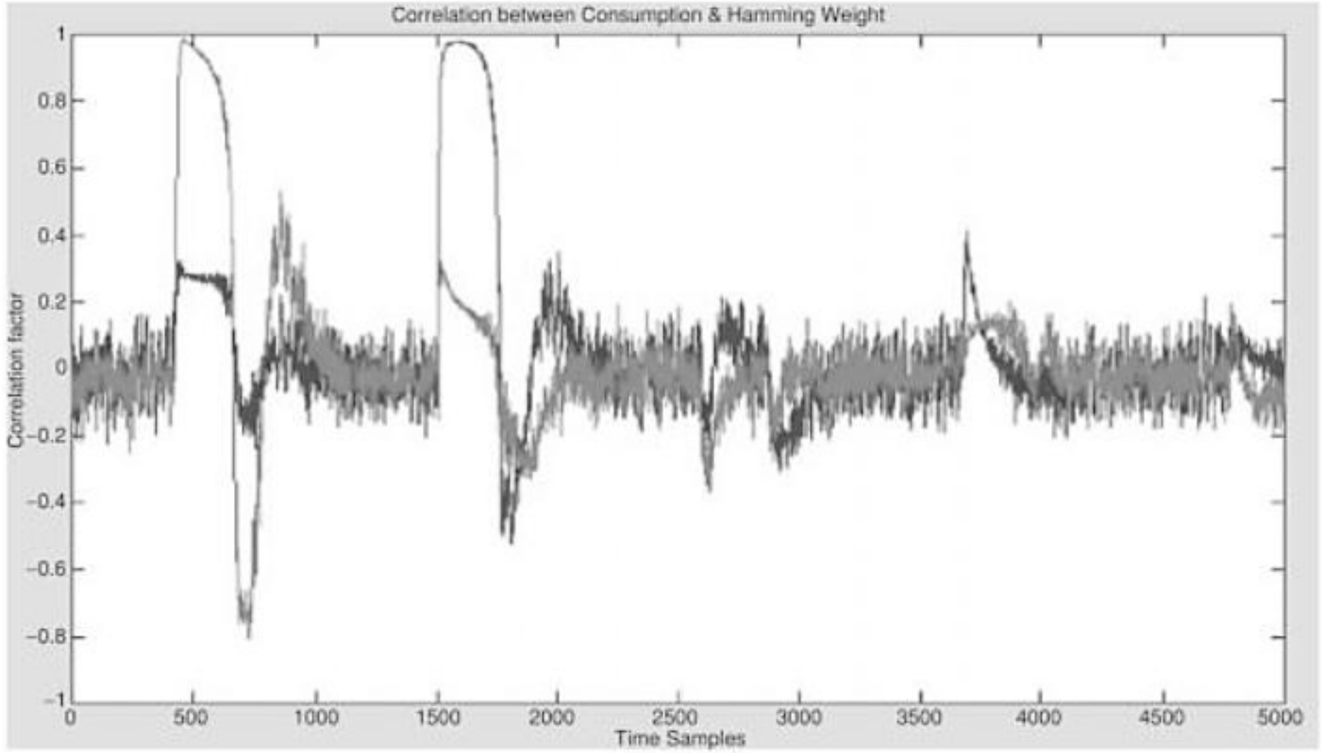
Simple Power Analysis



Differential Power Analysis



Differential Power Analysis



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;
}
```

Expensive Operation

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;
}
```

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;
}
```

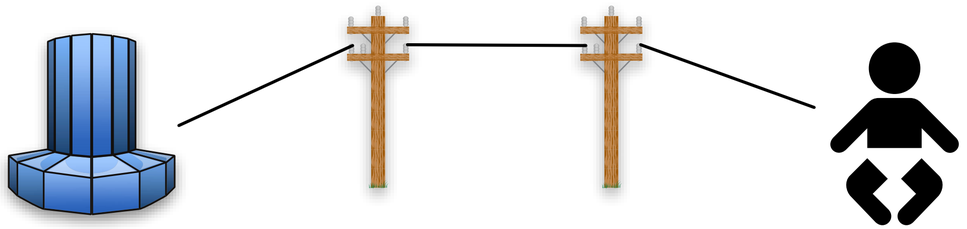
Expensive Operation

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

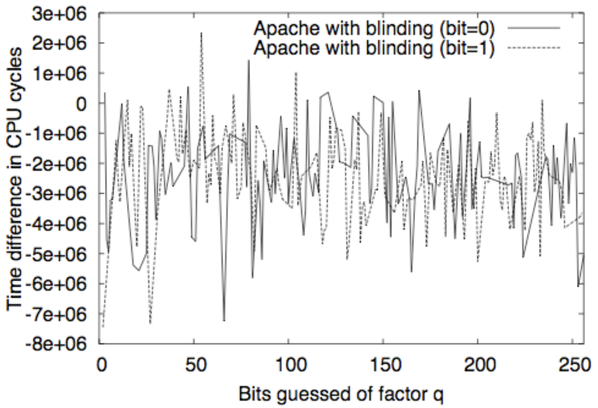
Remote Timing Attacks

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



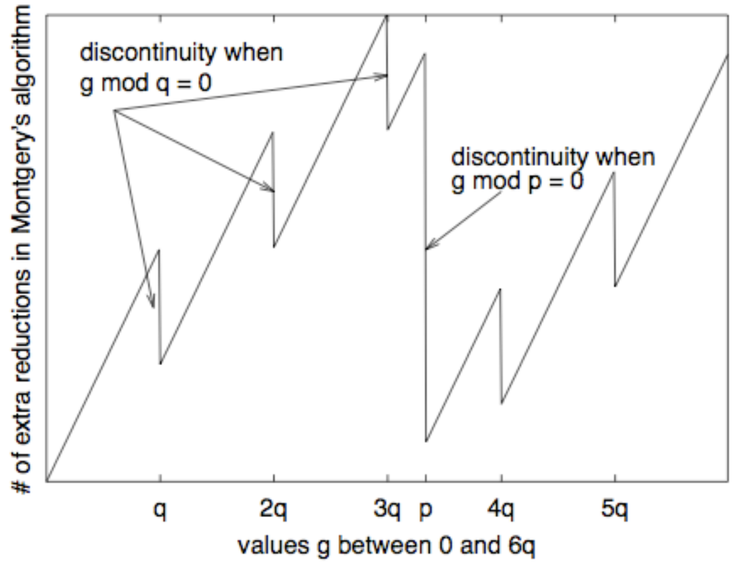
Solutions

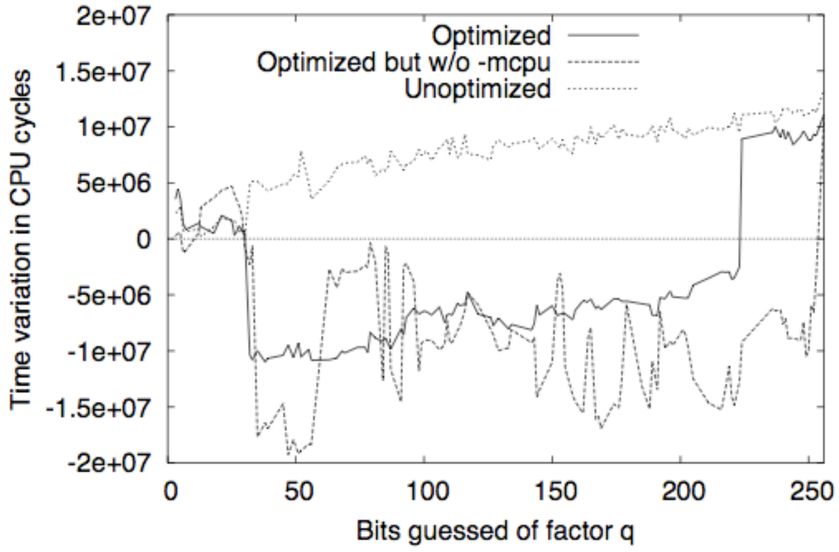
- Quantization:
  - All operations take the same time
  - Hard to do without sacrificing performance
- Blinding:
  - Prevents attacker from selecting ciphertext (that is processed with the secret key)



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

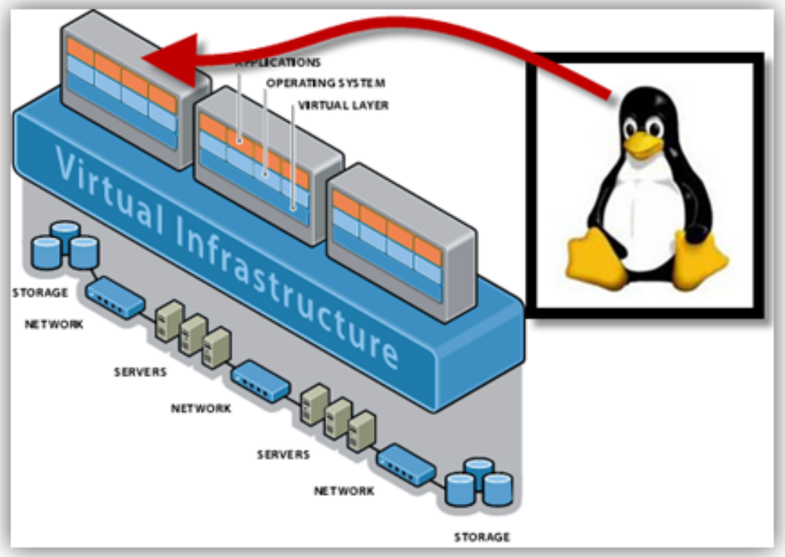
Windowed Exponentiation

c^d

0010

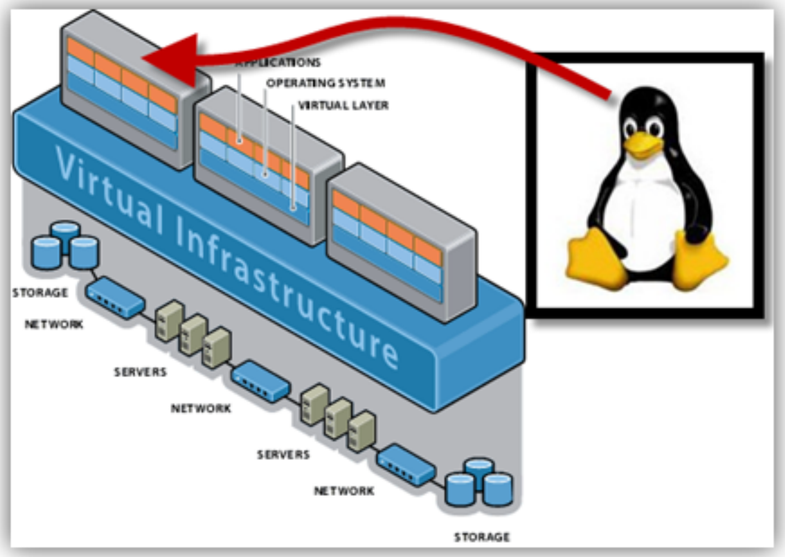
Cache Timing Attacks

- 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!

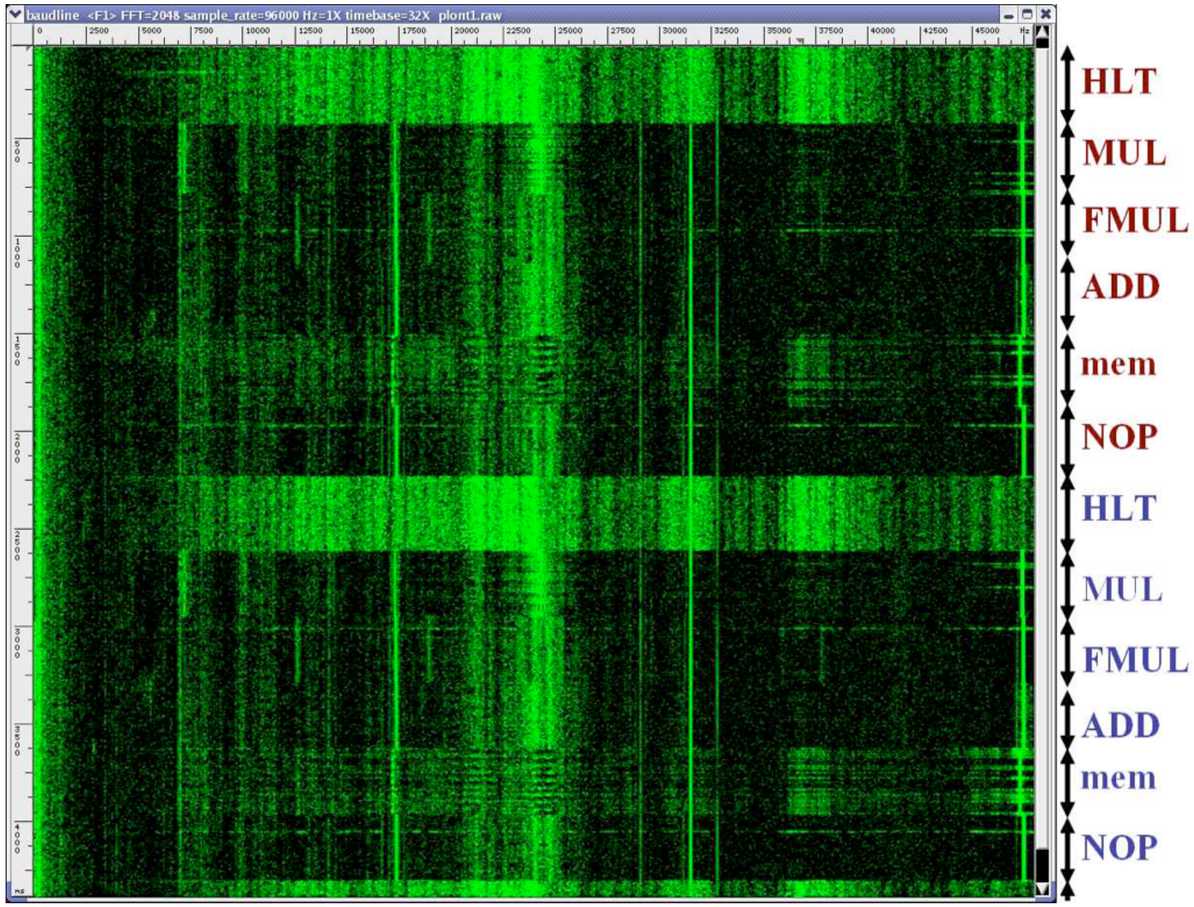


Hypervisor attacks

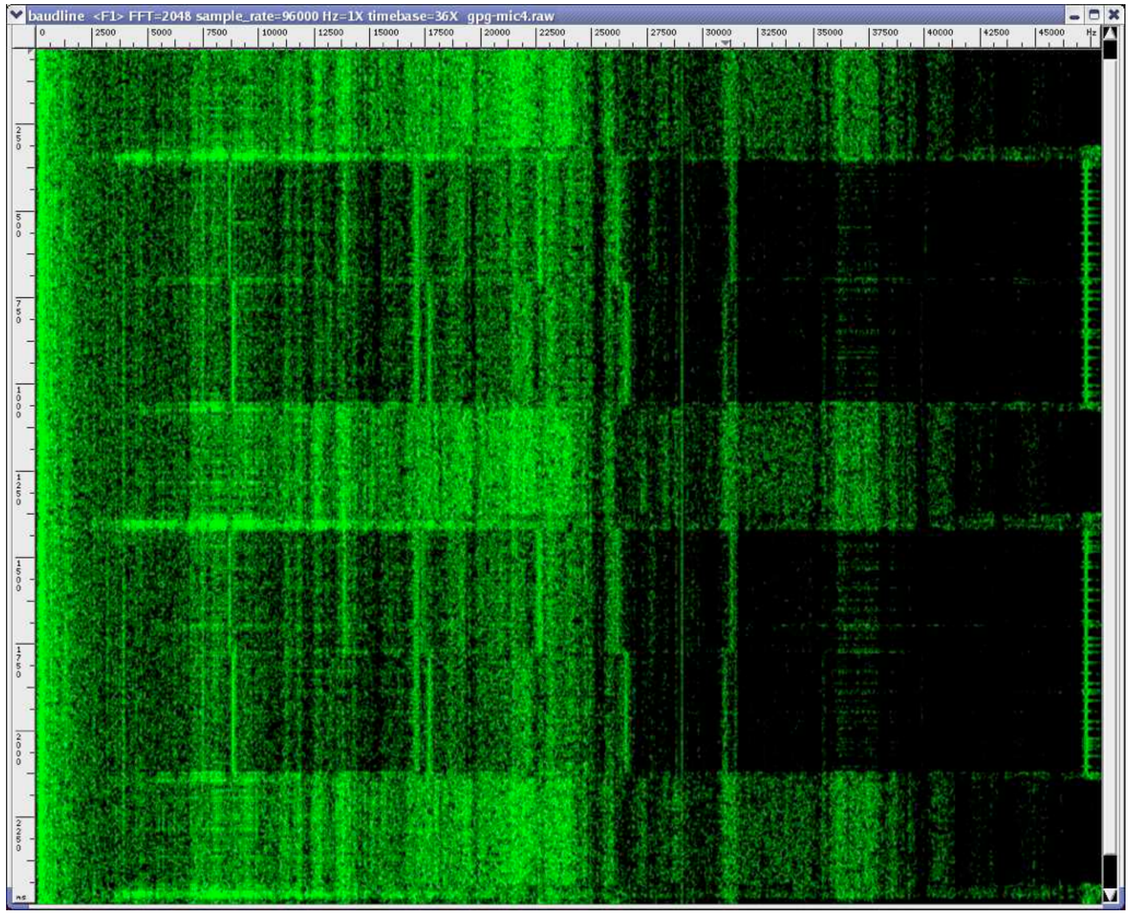
- Observation
  - This applies to code too!

```
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
```

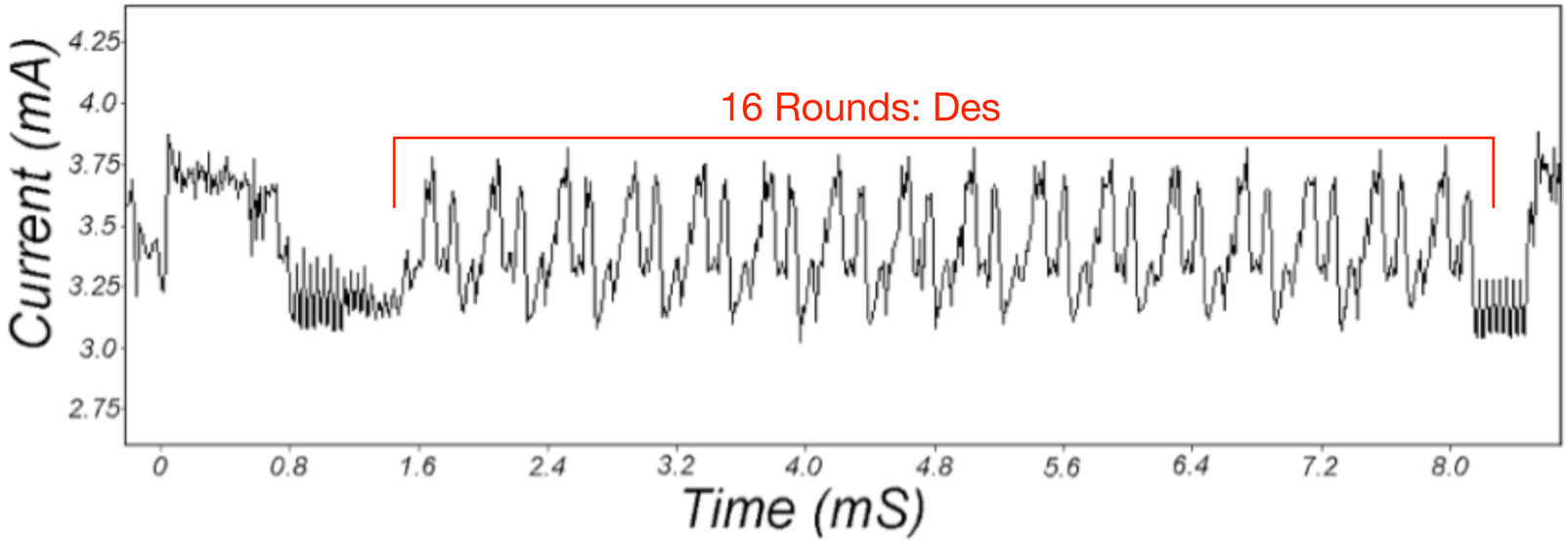
Acoustic Side Channels



Acoustic Side-Channels



Reverse Engineering



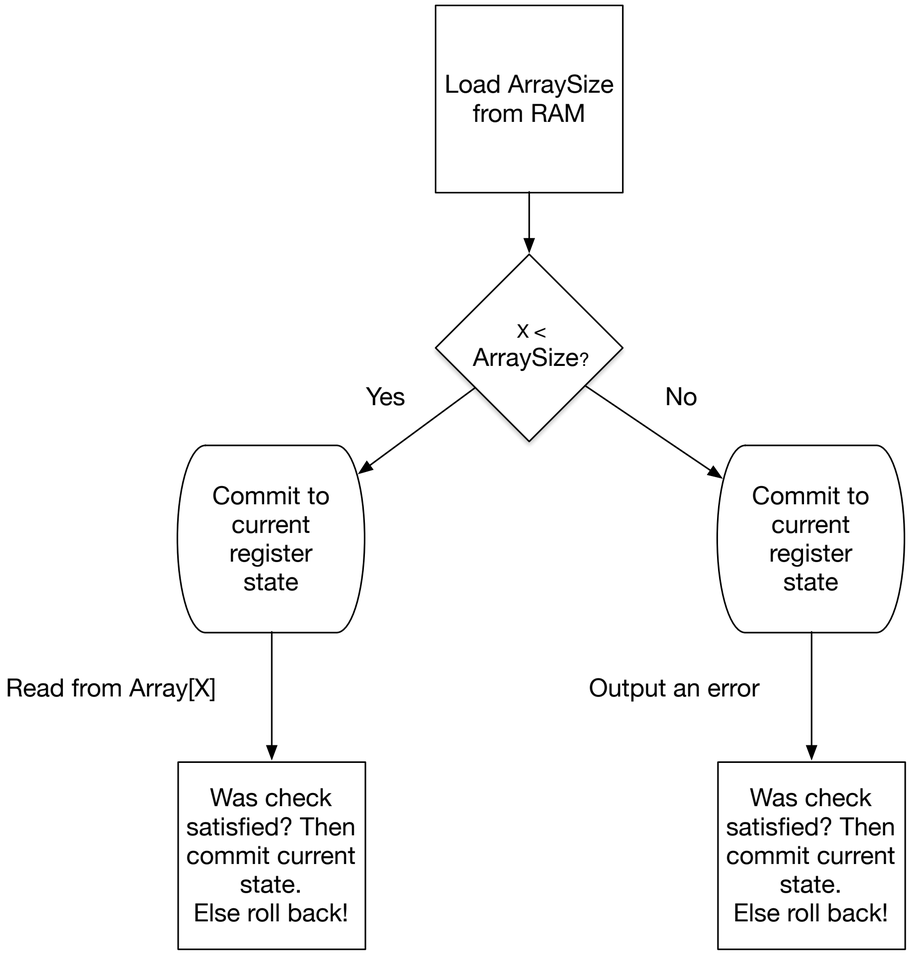
Speculative Execution

- 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

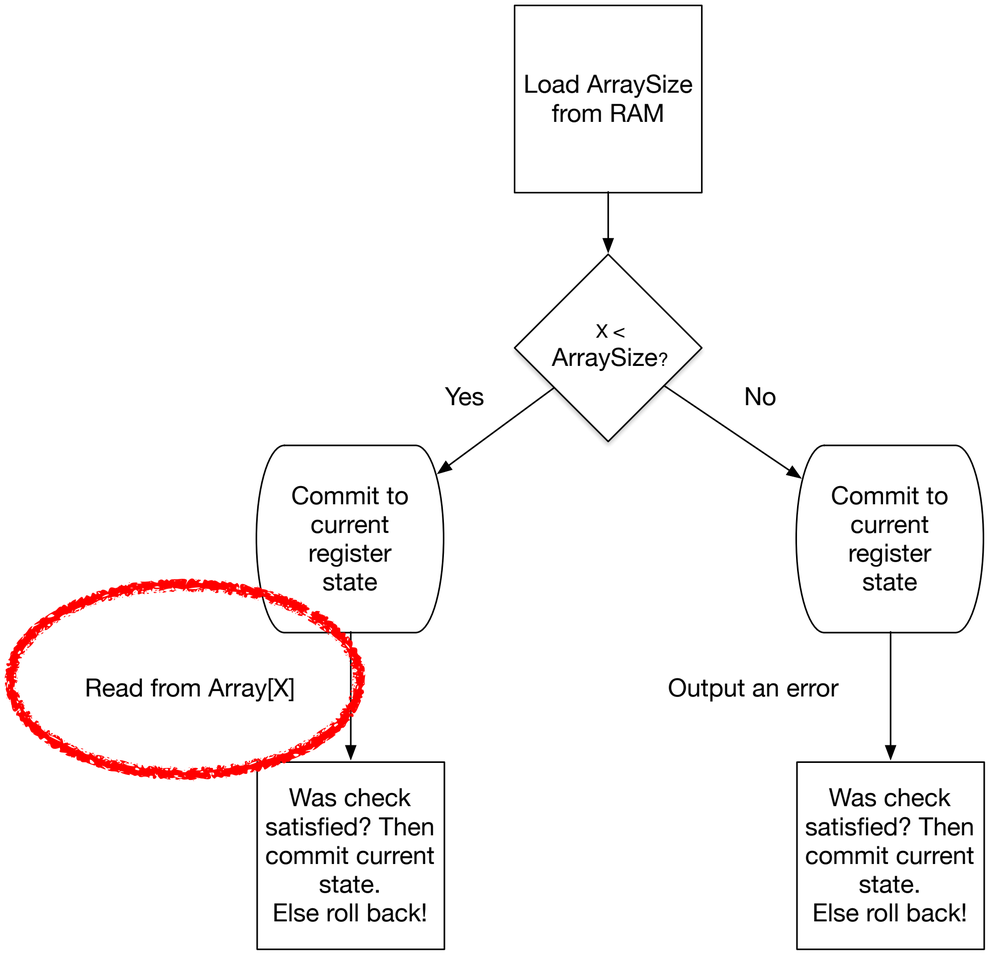


Speculative Execution

```
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