

Side-channels and constant-time analysis

Thomas Jensen

SOS

Master Science Informatique

Univ. Rennes 1

Side channels

Programs can leak information in many ways:

- Direct and indirect information flows
 - ▶ deducing secrets from observing intermediate and final values of the computation.
- Side channels
 - ▶ execution time,
 - ▶ energy consumption, sound emission
 - ▶ cache behaviour, branch prediction

An essential part of an **attacker model** is to describe what is **observable** for an attacker.

Timing channels

If an attacker can observe execution time of a program then

if $h > 0$ then (while $l \neq 0$ do $l := l-1$) else $l := 0$;

leaks the secret, even though non-interferent.*

A concrete problem in crypto algorithms such as RSA.

```
s := 1;
i := 0;
while (i < w) {
  if (k[i])
    r := (s*x) mod n
  else
    r := s;
  s := r*r;
  i := i+1
}      (The result is now in r)
```

Computes
 $x^k \bmod n$

Figure 11: An implementation of the modular exponentiation algorithm that leaks through timing.

J. Agat: Transforming out timing leaks, POPL 2000

* assuming $l > 0$

Timing analysis - for real

Seen in October 2019:



This page describes our discovery of a group of side-channel vulnerabilities in implementations of **ECDSA/EdDSA** in programmable smart cards and cryptographic software libraries. Our attack allows for practical recovery of the long-term private key. We have found implementations which leak the bit-length of the scalar during scalar multiplication on an elliptic curve. This leakage might seem minuscule as the bit-length presents a very small amount of information present in the scalar. However, in the case of **ECDSA/EdDSA** signature generation, the leaked bit-length of the random nonce is enough for full recovery of the private key used after observing a few hundreds to a few thousands of signatures on known messages, due

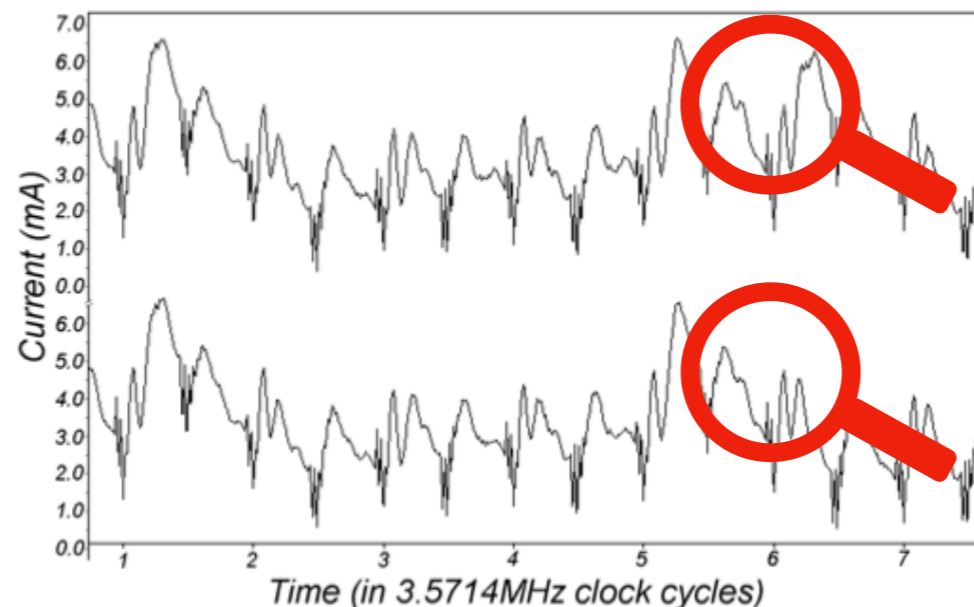
- **What is required to perform the attack?**

The attacker needs to be able to measure the duration of hundreds to thousands of signing operations of known messages. The less noise in the measurement is present, the less signatures the attacker needs. The computation of the private key is then a matter of seconds or minutes.

Power analysis

Deduce values of cryptographic keys

- ▶ stored on a smart card
- ▶ by measuring the power consumption.



From: P. Kocher et al: Differential Power analysis, CRYPTO'99

At cycle 6 there is a jump if i -th bit in key is set.

But: need physical access to the card to measure current.

Remote timing attacks

Do we need physical access to the processor? **No!**

Timing attacks have been done:

- between processes on the same machine,
- between virtual machines on same processor,
- *between different machines on the same network.

Example*: The OpenSSL implementation of RSA uses a sophisticated **modulo reduction** (due to Montgomery) and two different **multiplication** operators.

This leads to timing differences when computing $g^d \bmod q$.

*Brumley and Boneh: Timing attacks are Practical, 12th Usenix Security Symposium, 2003

Remote timing attacks

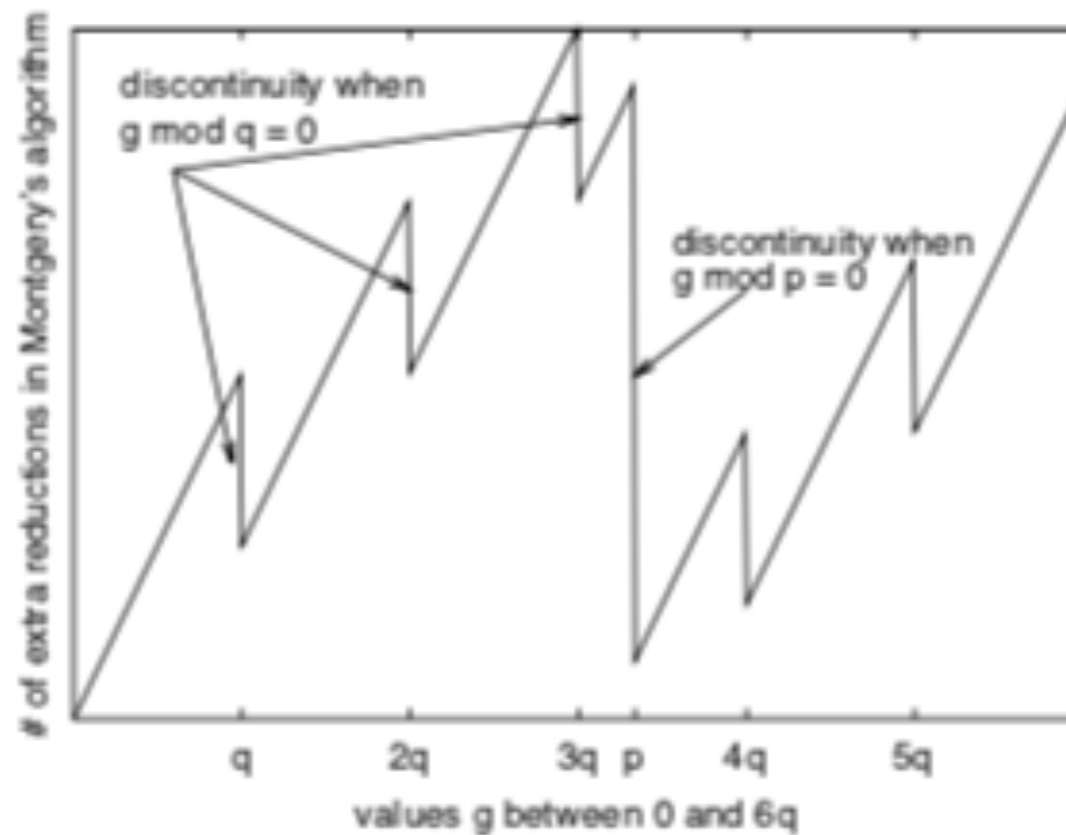


Figure 1: Number of extra reductions in a Montgomery reduction as a function (equation 1) of the input g .

Cache behaviour attacks

Another important side channel:

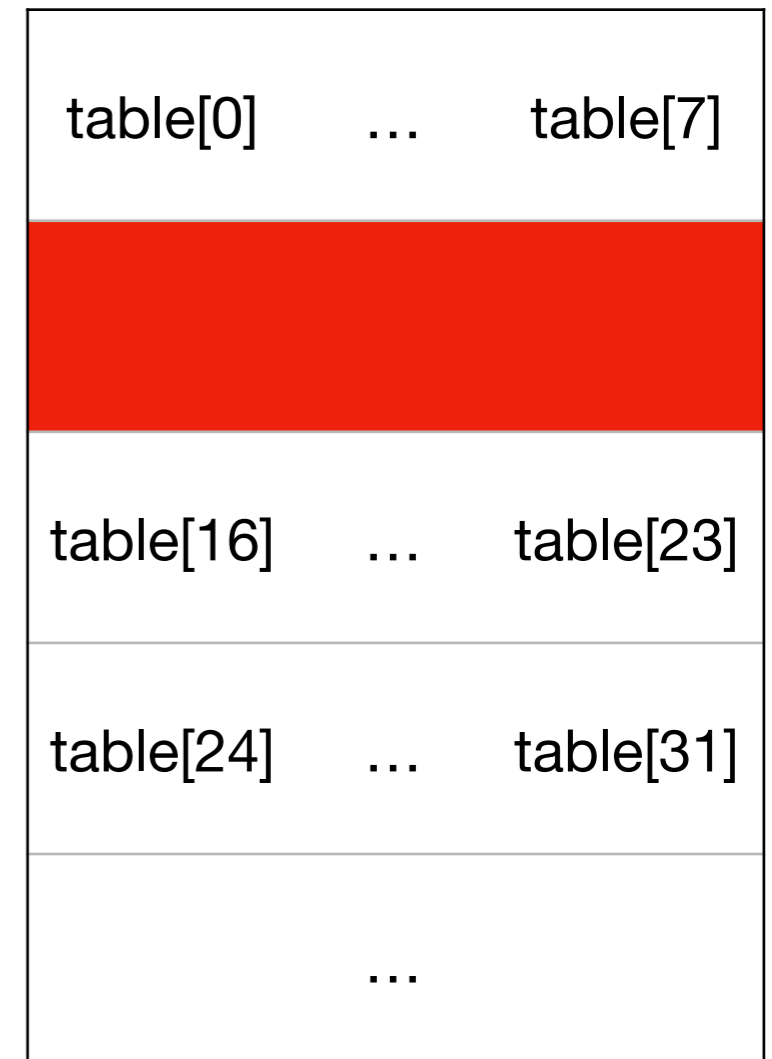
the memory cache.

Optimised crypto implementations use the secret key to index tables ("S-boxes").

`table[key[0]]`

The `table` is kept in cache, but certain lines can be evicted by attacking software.

Leads to differences in loading time, and hence to information about the key.



Memory cache with 2nd row evicted

Rule: avoid memory accesses depending on secrets.

Counter-measures

Guaranteeing constant-time

1. Program transformation for C-T.
2. Controlling compiler optimisations.
3. Languages for constant-time programming:
 - HACL - a variant of F^* for C-T programming.
 - C and Fact.
 - Intermediate and assembly languages.
4. Verifying constant-time by program analysis.

Program transformation for C-T.

Eliminate timing leaks by transforming the program into an equivalent program with constant execution time.

One source of problem: **branchings involving secrets.**

Remove problem

- by inserting ghost assignments,
- by removing assignments from branches.

Transforming out timing leaks

Back to the example with exponentiation.

For now, assume that attacker only observes **number** of execution steps - not cache behaviour.

Remove difference in execution time of branches by **inserting ghost ("dummy") assignments** with no effect on the end result.

```
s := 1;
i := 0;
while (i < w) {
  if (k[i])
    r := (s*x) mod n;
    skipAsn r s
  else {
    skipAsn r ((s*x) mod n);
    r := s
  };
  s := r*r;
  i := i+1
}
```

Figure 12: The output of our transformation: a secure implementation of the modular exponentiation algorithm.

J. Agat: Transforming out timing leaks, POPL 2000

"Branchless" assignment

Another technique to remove a dependency on a secret:

branchless assignments.

Example: Replace

`if h > 0 then x := exp`

by*

`x := h * exp + (1 - h) * x`

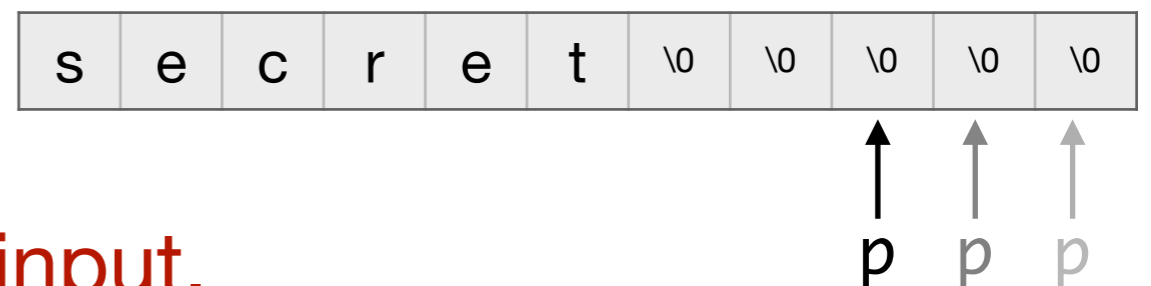
*here we assume our secret $h \in \{0,1\}$

A timing leak from the real world

Example from early mbdTLS: **unsecure** impl. of function that returns index of last non-zero element of secret input.

- Loop starts from end of buffer `input` and exits as soon as first non-zero element is found.

```
static int get_zeros_padding (unsigned char *input,
                             size_t input_len, size_t *data_len) {
    unsigned char *p = input + input_len - 1;
    ...
    while (*p == 0x00 && p > input)
        - - p;
    *data_len = (*p == 0x00) ? 0 : p - input + 1;
    ...
}
```

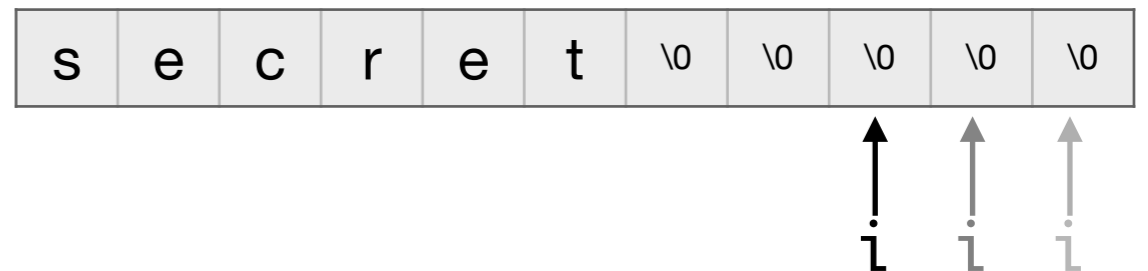


- This leaks the secret size of input.

Exercise

Find a loop body (complete the ...) that computes length of data and eliminates the timing leak, by iterating over the whole buffer:

```
i = input_len;    /* length of secret input[]
done = 0
while (i > 0) {
    prev_done = done;
    done |= ...
    data_len = ...
    i := i - 1;
};
return data_len;
```



Solution

To eliminate the timing leak, iterate over whole buffer with loop body:

```
i = input_len;    /* length of secret input[]
done = 0
while (i > 0) {
    prev_done = done;
    done |= (input[i-1] != 0);
    *data_len |= i * (done != prev_done)
    i := i - 1;
};
return data_len;
```

- Here, `done` changes from 0 to 1 when we reach the end of the string (first non-null character).
- This is the only `i` for which `done != prev_done`

Compiling C-T

Even if a source program is C-T , its compiled version may no longer be so!

Optimising compilers may:

- introduce tests on secrets,
- use instructions whose execution time depends on arguments
 - (on some processors, multiplication is a variable-time instruction),
- remove security-relevant code that does not affect functional behaviour
 - eg remove memory operations that look like dead code.

Languages for C-T

Languages for constant-time programming: controlling dependencies on secrets. Often specifically targeted for writing crypto code:

- HACL* - a crypto library written in functional language F* and compiled to C.
 - introduce an abstract type of "secret" integers `uint32_s` that the compiler generates constant-time instructions for.

```
val eq_mask: x:uint32_s → y:uint32_s → Tot (z:uint32_s {  
  if reveal x = reveal y then reveal z = 0xfffffffful  
  else reveal z = 0x0ul})
```

- use the type system to prevent dependencies on `uint32_s`
- QHASM portable assembly language (<http://cr.yp.to/qhasm.html>)
- Fact - a DSL for writing cryptographic algorithms in idiomatic, high-level C.

FaCT

FaCT: a domain-specific language for writing constant-time programs.

FaCT :

- a subset of C,
- a program transformation to C-T based on information flow analysis.

Cauligi et al: FaCT: A DSL for Timing-Sensitive Computation, PLDI'19, ACM Press

Early termination

Consider comparing two secret arrays x and y.

We would like to write this:

```
for (i from 0 to n)
  if (x[i] != y[i])
    return -1;
return 0;
```

But to be constant-time, programmers write this:

```
for (i=0;i<n;i++)
  d |= x[i] ^ y[i];
return (1 & ((d-1)>>8)) - 1;
```

Memory access

Copy a to b if secret swap is true

In FaCT:

```
if (swap != 0) {  
  for(i from 0 to 5) {  
    secret tmp = a[i];  
    a[i] = b[i];  
    b[i] = tmp;  
  }  
}
```

instead of the "classical" solution:

```
for (i=0;i<5;++i) {  
  x = swap & (a[i] ^ b[i]);  
  a[i] ^= x;  
  b[i] ^= x;  
}
```

FaCT language

Statements:

$S ::= S;S \mid x=e \mid x=f(e) \mid e:=e \mid \text{if}(e) \{S\} \text{ else } \{S\}$
 $\mid \text{for}(x \text{ from } e \text{ to } e)\{S\} \mid \text{return } e$

Expressions:

$e ::= \text{true} \mid \text{false} \mid n \mid x \mid e \oplus e \mid e[e] \mid \text{len } e \mid$
...
 $\mid \text{ctselect}(e, e, e)$

Constant-time select $\text{ctselect}(b, e1, e2)$ returns the value of second or third argument, depending on the value of b .

The compiler will guarantee that `ctselect` is compiled to constant-time code.

FaCT type system

The FaCT type system works over types (int, bool, arrays) with security levels (public, secret)

Programmers can annotate variables to indicate secret and public data.

The type system will reject programs which

- are not information flow secure or
- cannot be transformed to constant-time.

Typing rules for expressions - a selection

- `ctselect` is typable for all security levels.
- the security level of result is upper bound of levels of arguments

$$\begin{array}{c} \text{T-CT-SEL} \\ \Gamma \vdash e_1 : \text{BOOL}_\ell \\ \beta \text{ is numeric or BOOL} \quad \Gamma \vdash e_2 : \beta \quad \Gamma \vdash e_3 : \beta \\ \hline \Gamma \vdash \text{ctselect}(e_1, e_2, e_3) : \beta \sqcup \ell \end{array}$$

- array access is safe if index is public.
- the index must be in bound

$$\begin{array}{c} \text{T-ARR-GET} \\ \Gamma \vdash e_1 : \text{ARR}^{\text{SZ}}[\beta] \quad \Gamma \vdash e_2 : \text{UINT}_{\text{PUB}}^{\text{S}} \quad \Gamma \Rightarrow e_2 < \text{len } e_1 \\ \hline \Gamma \vdash e_1[e_2] : \beta \end{array}$$

Public safety

Public safety

Problem: the C-T transformation may introduce safety problems:

```
assume(secret_index <= len buf);  
if (i < secret_index)  
    buf[i] = 0;...
```

should not be transformed into

```
cond = (i < secret_index);  
buf[i] = ctselect(cond, 0, buf[i]);
```

because it introduces a potential buffer overflow

Public safety: the safety of a program must not depend on secret data.

Extend the type system with **path conditions** to enforce public safety

$$\frac{\text{T-ASSUME} \quad \Gamma \vdash e : \text{BOOL}_\ell \quad \Gamma' = \Gamma \wedge e}{\omega, pc, \beta_r \vdash \text{assume}(e) : \Gamma, rc \rightarrow \Gamma', rc}$$

Typing of statements

return
context

$$\omega, pc, \beta_r \vdash S : \Gamma, rc \rightarrow \Gamma', rc'$$

Classical if rule:

T-IF

$$\frac{\begin{array}{c} \Gamma \vdash e : \text{BOOL}_\ell \\ \omega, pc \sqcup \ell, \beta_r \vdash S_1 : \Gamma \wedge e, rc \rightarrow \Gamma_1, rc_1 \\ \omega, pc \sqcup \ell, \beta_r \vdash S_2 : \Gamma \wedge \neg e, rc \rightarrow \Gamma_2, rc_2 \end{array}}{\omega, pc, \beta_r \vdash \text{if } (e) \{ S_1 \} \text{ else } \{ S_2 \} : \Gamma, rc \rightarrow \Gamma, rc_1 \sqcup rc_2}$$

Only iterate over public bounds:

T-FOR

$$\frac{\begin{array}{c} \Gamma \vdash e_1 : \text{UINT}_{\text{PUB}} \quad \Gamma \vdash e_2 : \text{UINT}_{\text{PUB}} \\ \Gamma' = \Gamma, x : \text{UINT}_{\text{PUB}} \wedge e_1 \leq x < e_2 \\ rc \sqsubseteq rc' \quad \omega, pc, \beta_r \vdash S : \Gamma', rc' \rightarrow \Gamma'', rc' \end{array}}{\omega, pc, \beta_r \vdash \text{for } (x \text{ from } e_1 \text{ to } e_2) \{ S \} : \Gamma, rc \rightarrow \Gamma, rc'}$$

Return deferral

Early returns that depend on secrets may leak information:

```
if (sec) { return 1; }  
// long-running computation ...
```

FaCT will transform this into `ifs` that depend on secret info.

```
secret rval = 0;  
secret bool notRet = true;  
if (sec) { rval = 1; notRet = false; }  
if (notRet) {  
// long-running computation ...  
}  
return rval;
```

Doesn't solve the problem - but now we only have to deal with `if`.

Branch removal

Turn secret `if` into straight-line code:

```
if      (sec1) {a[1]=3;}  
else if (sec2) {a[2] = 4;}
```

becomes

```
a[1] = ctselect( sec1 , 3, a[1]);  
a[2] = ctselect(~sec1 & sec2, 4, a[2]);
```

Transformation rules for branch removal

Transformation rules of the form

$$p \vdash S \rightarrow S'$$

where p is a control predicate (= under what condition is S executed).

TR-BR-ASSIGN

$$\frac{p \neq \text{true}}{p \vdash e_1 := e_2 \rightarrow e_1 := \text{ctselect}(p, e_2, e_1)}$$

TR-BR-IF

$$\frac{\Gamma \vdash e : \text{BOOL}_{\text{SEC}} \quad \text{FRESH } m_t, m_f \quad \begin{array}{l} (p \& m_t) \vdash S_1 \rightarrow S'_1 \\ (p \& m_f) \vdash S_2 \rightarrow S'_2 \end{array}}{p \vdash \text{if}(e) \{ S_1 \} \text{else} \{ S_2 \} \rightarrow \begin{array}{l} \{ \text{BOOL}_{\text{SEC}} m_t = e; \\ \text{BOOL}_{\text{SEC}} m_f = \neg m_t; \\ S'_1; S'_2 \} \end{array}}$$

Security proof

Define leakage (ie what the attacker can observe) as a trace of **events** e^* :

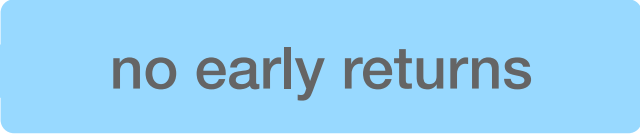

- the branches taken,
- the memory accessed.

Big-step leakage semantics $P : (\rho, \sigma) \xrightarrow{e^*} (\rho', \sigma')$

Definition: A program is C-T if leakage does not depend on secret input.

Theorem: Transformation $P \rightarrow P'$ produces a C-T program.

Proof:

- Prove $\vdash P$ and $\vdash P \rightarrow_{rd} P'$ then $\vdash_{rd} P'$  no early returns
- Prove $\vdash_{rd} P$ and $P \rightarrow_{ct} P'$ then $\vdash_{ct} P'$  no branch on secrets

Limits of FaCT

The FaCT type system will reject some programs that can be transformed to C-T.

FaCT rejects

```
if (sec) (x = l1 ; y = t[x]) else skip
```

because branch removal would produce the unsecured

```
x = ctselect(sec, l1, x);  
y = ctselect(sec, t[x], y)
```

But a secure C-T transformation exists:

```
xt = l1 ; yt = t[xt];  
xf = x ; yf = y;  
x = ctselect(sec, xt, xf);  
y = ctselect(sec, yt, yf)
```

Exercise

Consider the following program :

```
if (sec) x = l1 else x = l2; y = t[x];
```

FaCT will reject it (memory access with secret index!)

Can you find a C-T equivalent?

(Hint: increase the scope of the `if`)

Summary

Side channels are manifold - and arise regularly in software.
Particularly critical in optimised cryptographic primitives.

Timing differences can be observed

- on processors
- across processors
- and even across networks.

Different measures to eliminate certain side channels

- program to avoid timing leaks
 - no test on secrets and
 - no access memory with secrets.
- transform and verify constant-time of implementations.