Applied Cryptology

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
⟨csdsp@bristol.ac.uk⟩

April 24, 2024

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
   - the associated notes page may be pre-populated with extra, written explaination of material covered in lecture(s), plus
   - anything with a "grey'ed out" header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

Notes:

COMS30048 lecture: week #16

- **Agenda**: explore **implementation attacks** via
  1. an "in theory", i.e., concept-oriented perspective,
     1.1 explanation,
     1.2 justification,
     1.3 formalisation.

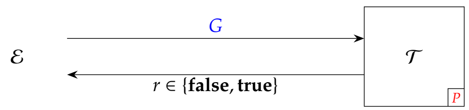
     and
  2. an "in practice", i.e., example-oriented perspective,
     2.1 attacks,
     2.2 countermeasures.
- Caveat!

$$\sim 2 \text{ hours} \Rightarrow \text{introductory, and (very) selective (versus definitive) coverage.}$$

Notes:

Part 1.1: in theory (1)
Explanation

- Scenario:
  - given the following interaction between an **attacker** $\mathcal{E}$ and a **target** $\mathcal{T}$



  - and noting that
    - the password P has $|P|$ characters in it,
    - each character in G and P is assumed to be from a known alphabet

$$A = \{\text{'a'}, \text{'b'}, \ldots, \text{'z'}\}$$

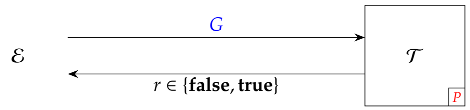
    such that $|A| = 26$,
  - how can $\mathcal{E}$ mount a successful attack, i.e., input a guess G matching P?

Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **brute-force attack** (i.e., try every G).

Attack (P = "pencil")

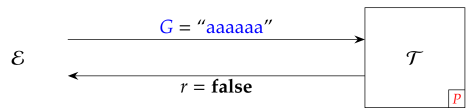


Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **brute-force attack** (i.e., try every G).

Attack (P = "pencil")

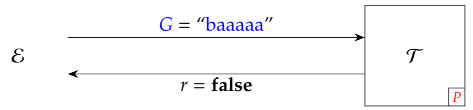


Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **brute-force attack** (i.e., try every G).

Attack (P = "pencil")

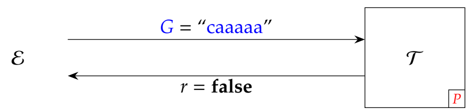


Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **brute-force attack** (i.e., try every G).

Attack (P = "pencil")

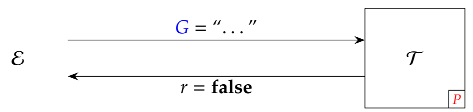


Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **brute-force attack** (i.e., try every G).

Attack (P = "pencil")

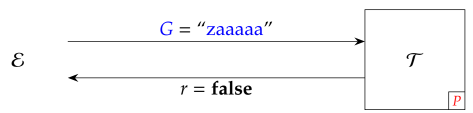


Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **brute-force attack** (i.e., try every G).

Attack (P = "pencil")

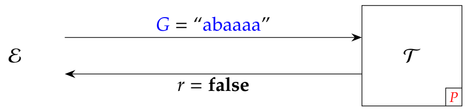


Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **brute-force attack** (i.e., try every G).

Attack (P = "pencil")

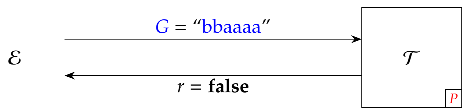


Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **brute-force attack** (i.e., try every G).

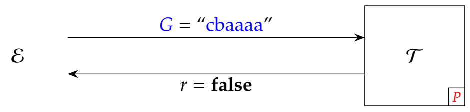
Attack (P = "pencil")



Notes:

- Idea: **brute-force attack** (i.e., try every G).

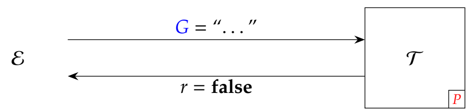
Attack (P = "pencil")



Notes:

- Idea: **brute-force attack** (i.e., try every G).

Attack (P = "pencil")

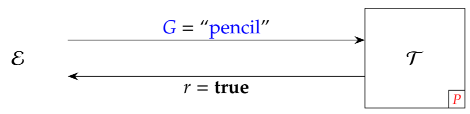


Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **brute-force attack** (i.e., try every G).

Attack (P = "pencil")



$\therefore$ if we play by the rules then

+ve: we always guess a $G = P$

−ve: we need quite a lot of guesses, e.g., for a 6-character lower-case password we'd make

$$26^6 = 308915776$$

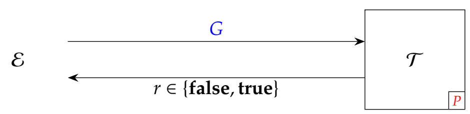
in the worst-case

Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **dictionary attack** (i.e., try common G).

Attack (P = "pencil", $G \in D$ = {"password", "admin", "bristolcity", . . . , "pencil"})

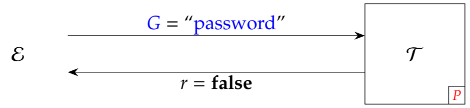


Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **dictionary attack** (i.e., try common G).

Attack (P = "pencil", $G \in D$ = {"password", "admin", "bristolcity", . . . , "pencil"})

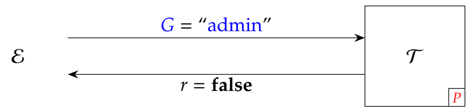


Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **dictionary attack** (i.e., try common G).

Attack (P = "pencil", $G \in D$ = {"password", "admin", "bristolcity", . . . , "pencil"})



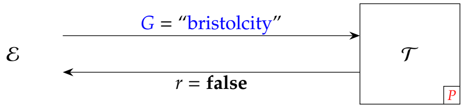
Notes:

Part 1.1: in theory (2)

Explanation

- Idea: **dictionary attack** (i.e., try common G).

Attack (P = "pencil", $G \in D$ = {"password", "admin", "bristolcity", ..., "pencil"})



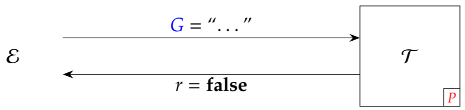
Notes:

Part 1.1: in theory (2)

Explanation

- Idea: **dictionary attack** (i.e., try common G).

Attack (P = "pencil", $G \in D$ = {"password", "admin", "bristolcity", ..., "pencil"})

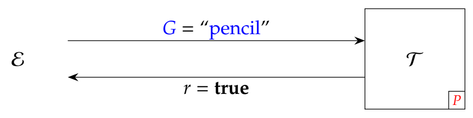


Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **dictionary attack** (i.e., try common G).

Attack (P = "pencil", $G \in D$ = {"password", "admin", "bristolcity", . . . , "pencil"})



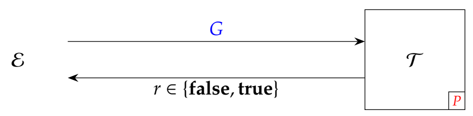
∴ if we play by the rules then
−ve: if $P \notin D$, we won't guess a $G = P$
+ve: we need fewer guesses, i.e., $|D|$ in the worst-case

Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **side-channel attack**.

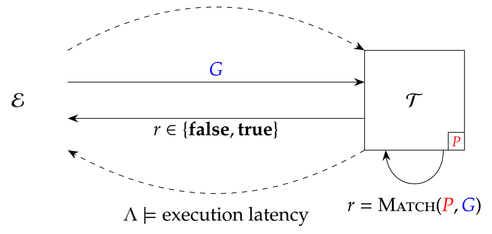
Attack (P = "pencil")



Notes:

- **Idea**: **side-channel attack.**

Attack (P = "pencil")



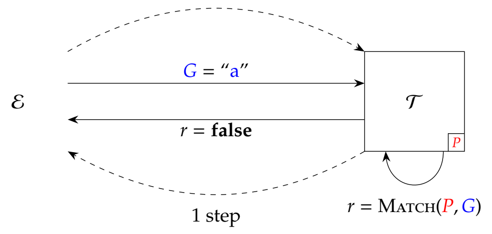

```
algorithm Match(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```

Notes:

Part 1.1: in theory (2)
Explanation

- **Idea**: **side-channel attack.**

Attack (P = "pencil")



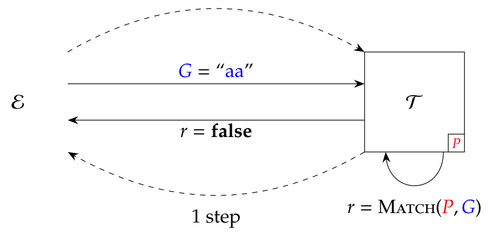

```
algorithm Match(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```

Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **side-channel attack**.

Attack (P = "pencil")



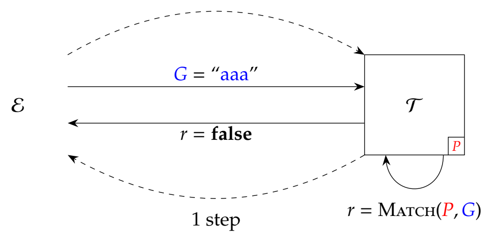

```
algorithm MATCH(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```

Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **side-channel attack**.

Attack (P = "pencil")




```
algorithm MATCH(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```

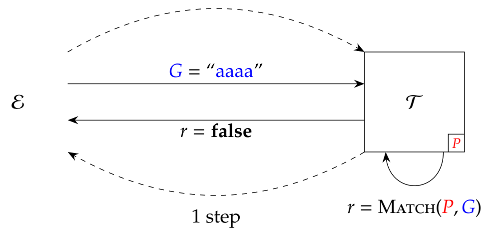
Notes:

Part 1.1: in theory (2)

Explanation

- Idea: **side-channel attack.**

Attack (P = "pencil")




```
algorithm MATCH(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```

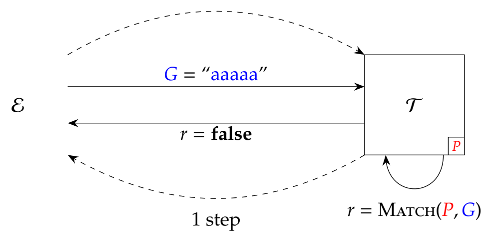
Notes:

Part 1.1: in theory (2)

Explanation

- Idea: **side-channel attack.**

Attack (P = "pencil")



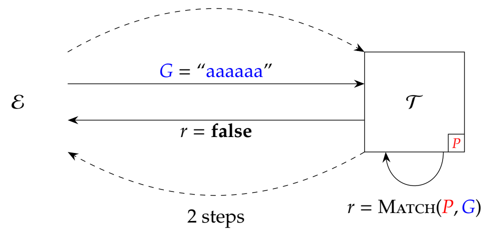

```
algorithm MATCH(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```

Notes:

Part 1.1: in theory (2)
Explanation

- **Idea**: **side-channel attack**.

Attack (P = "pencil")



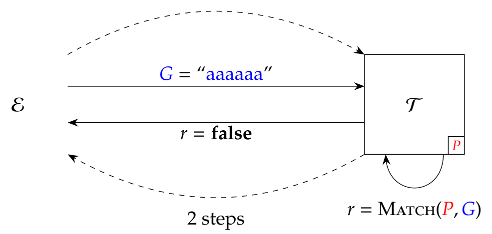

```
algorithm MATCH(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```


Notes:

Part 1.1: in theory (2)
Explanation

- **Idea**: **side-channel attack**.

Attack (P = "pencil")



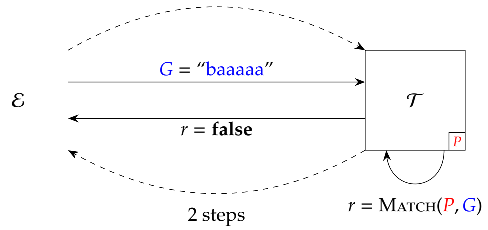

```
algorithm MATCH(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```


Notes:

Part 1.1: in theory (2)
Explanation

- **Idea**: **side-channel attack.**

Attack (P = "pencil")



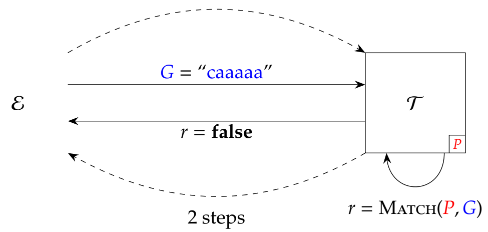

```
algorithm MATCH(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```


Notes:

Part 1.1: in theory (2)
Explanation

- **Idea**: **side-channel attack.**

Attack (P = "pencil")



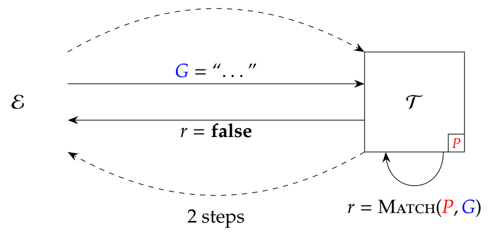

```
algorithm MATCH(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```


Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **side-channel attack.**

Attack (P = "pencil")



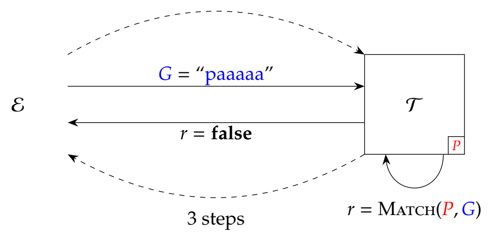

```
algorithm Match(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```

Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **side-channel attack.**

Attack (P = "pencil")




```
algorithm Match(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```

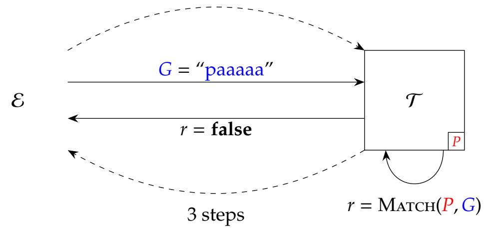
Notes:

Part 1.1: in theory (2)

Explanation

- Idea: **side-channel attack**.

Attack (P = "pencil")




```
algorithm MATCH(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```

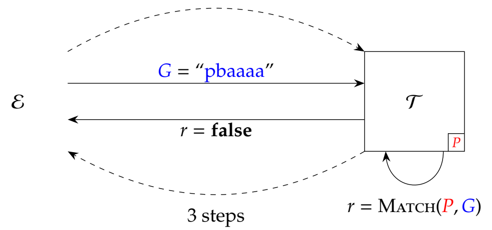
Notes:

Part 1.1: in theory (2)

Explanation

- Idea: **side-channel attack**.

Attack (P = "pencil")

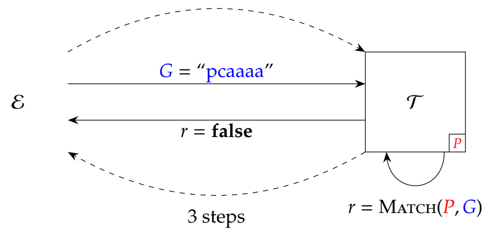



```
algorithm MATCH(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```

Notes:

- Idea: **side-channel attack**.

Attack (P = "pencil")

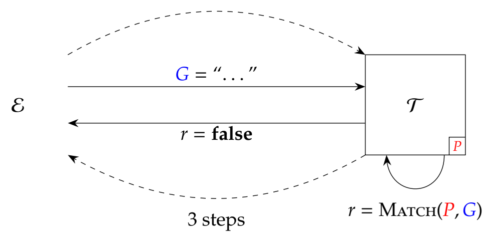



```
algorithm MATCH(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```

Notes:

- Idea: **side-channel attack**.

Attack (P = "pencil")



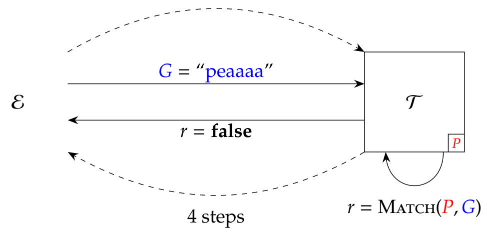

```
algorithm MATCH(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```

Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **side-channel attack.**

Attack (P = "pencil")



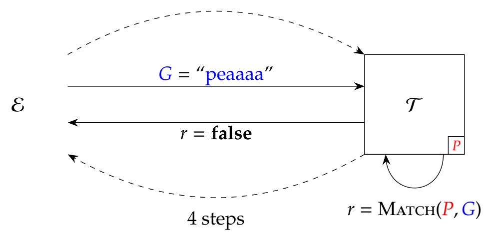

```
algorithm Match(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```

Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **side-channel attack.**

Attack (P = "pencil")




```
algorithm Match(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```

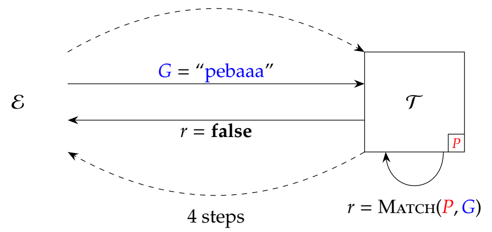
Notes:

Part 1.1: in theory (2)

Explanation

- Idea: **side-channel attack**.

Attack (P = "pencil")




```
algorithm Match(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```

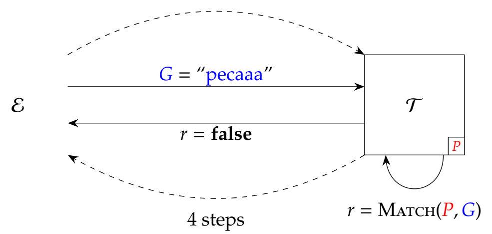
Notes:

Part 1.1: in theory (2)

Explanation

- Idea: **side-channel attack**.

Attack (P = "pencil")



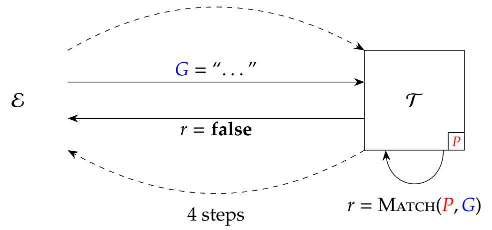

```
algorithm Match(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```


Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **side-channel attack.**

Attack (P = "pencil")



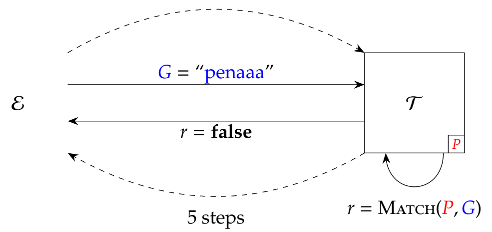

```
algorithm Match(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```


Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **side-channel attack.**

Attack (P = "pencil")



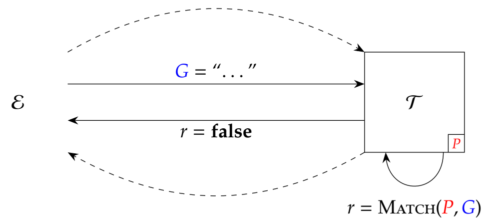

```
algorithm Match(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```


Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **side-channel attack.**

Attack (P = "pencil")



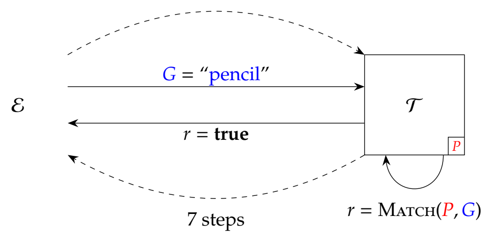

```
algorithm Match(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```

Notes:

Part 1.1: in theory (2)
Explanation

- Idea: **side-channel attack.**

Attack (P = "pencil")




```
algorithm Match(P, G) begin
  if |P| ≠ |G| then
    return false
  end
  for i from 0 upto |G| − 1 do
    if P_i ≠ G_i then
      return false
    end
  end
  return true
end
```

∴ if we *bend* the rules a little then

+ve: we always guess a $G = P$

+ve: we don't need too many guesses, e.g., for a 6-character lower-case password we'd make

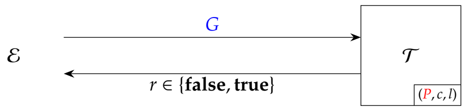
$$26 \cdot 6 = 156$$

in the worst-case (plus the few extra to recover $|P|$)

Notes:

Part 1.1: in theory (3)
Explanation

- Scenario:
  - given the following interaction between an **attacker** $\mathcal{E}$ and a **target** $\mathcal{T}$



  - and noting that
    - the Personal Identification Number (PIN) P has $|P| = 4$ digits in it,
    - each digit in G and P is assumed to be from a known alphabet

$$A = \{0, 1, \ldots, 9\}$$

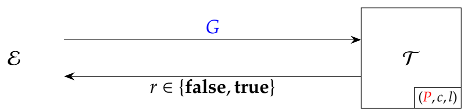
      such that $|A| = 10$,
    - the counter c is incremented after each (successive) incorrect guess; when c exceeds a limit $l = 3$, the target becomes "locked",
  - how can $\mathcal{E}$ mount a successful attack, i.e., input a guess G matching P?

Notes:

Part 1.1: in theory (4)
Explanation

- Idea:

Attack (P = 1234)

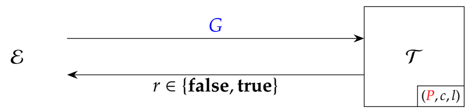


Notes:

Part 1.1: in theory (4)
Explanation

- Idea:

Attack ($P = 1234$)



$\therefore$ similar attacks as before apply, namely

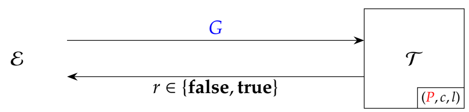
1. **brute-force attack**:
   - +ve: $10^4 = 10000$ possible PINs is not many
   - −ve: the counter limits how viable this approach is

Notes:

Part 1.1: in theory (4)
Explanation

- Idea:

Attack ($P = 1234$)



$\therefore$ similar attacks as before apply, namely

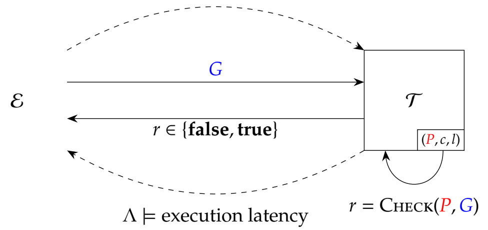
2. **dictionary attack**:
   - +ve: reasoning re. common passwords still applies to PINs (e.g., a birthday)
   - −ve: the counter limits how viable this approach is

Notes:

Part 1.1: in theory (4)
Explanation

- **Idea**:

Attack ($P = 1234$)




```
algorithm CHECK(P, G) begin
  if c ≥ l then
    return false
  end
  if P ≠ G then
    c ← c + 1
    return false
  end
  c ← 0
  return true
end
```

∴ similar attacks as before apply, namely

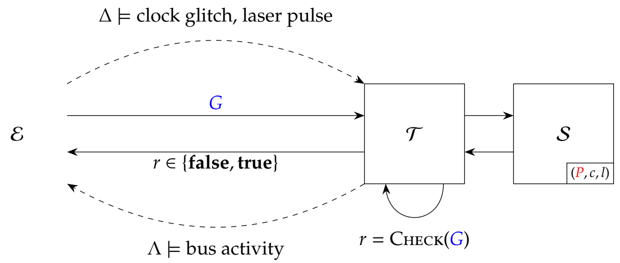
3. **side-channel attack**:
   - +ve: we can still measure execution time of Check
   - −ve: comparison of P and G no longer has data-dependent execution time

Notes:

Part 1.1: in theory (4)
Explanation

- **Idea**:

Attack ($P = 1234$)




```
algorithm CHECK(G) begin
  if LOAD(c) ≥ LOAD(l) then
    return false
  end
  if LOAD(P) ≠ G then
    STORE(c, LOAD(c) + 1)
    return false
  end
  STORE(c, 0)
  return true
end
```

but consider some more implementation detail:

1. we might consider *different* indirect inputs *and* outputs,
2. use of an external, non-volatile storage (e.g., SIM card) implies that for $x \leftarrow y$ we have

$$\left.\begin{array}{lcl} x \text{ on LHS} & \rightsquigarrow & \text{store operation} \\ y \text{ on RHS} & \rightsquigarrow & \text{load operation} \end{array}\right\} \rightsquigarrow \textsc{Store}(x, \textsc{Load}(y))$$

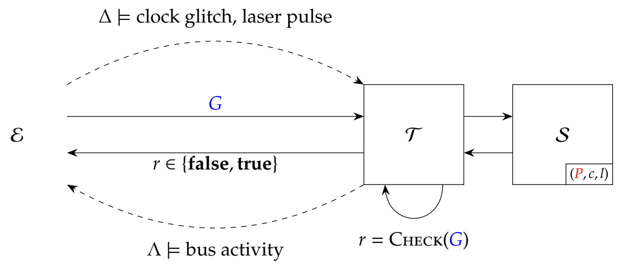
Notes:

Part 1.1: in theory (4)

Explanation

- Idea: **fault injection attack**.

Attack ($P = 1234$)




```
1  algorithm Check(G) begin
2    if Load(c) ≥ Load(l) then
3      return false
4    end
5    if Load(P) ≠ G then
6      Store(c, Load(c) + 1)
7      return false
8    end
9    Store(c, 0)
10   return true
11 end
```

∴ we could consider

1. disrupting *state*, e.g.
   - corrupt (or randomise) content stored by $\mathcal{S}$,
   - if l is an n-bit integer, all $2^n - l$ values of a random l' mean more guesses.

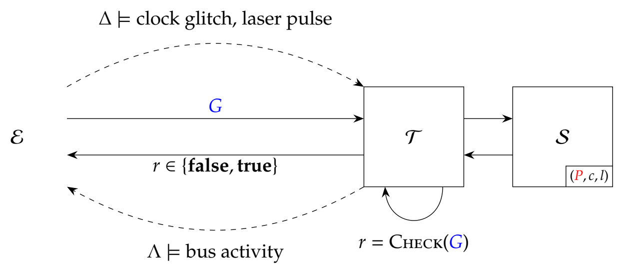
Notes:

Part 1.1: in theory (4)

Explanation

- Idea: **fault injection attack**.

Attack ($P = 1234$)




```
1  algorithm Check(G) begin
2    if Load(c) ≥ Load(l) then
3      return false
4    end
5    if Load(P) ≠ G then
6      Store(c, Load(c) + 1)
7      return false
8    end
9    Store(c, 0)
10   return true
11 end
```


∴ we could consider

2. disrupting *execution*, e.g.
   - control the power supply and probe the command bus,
   - when a command of the form $\text{Store}(x, y)$ is detected, we know it relates to either

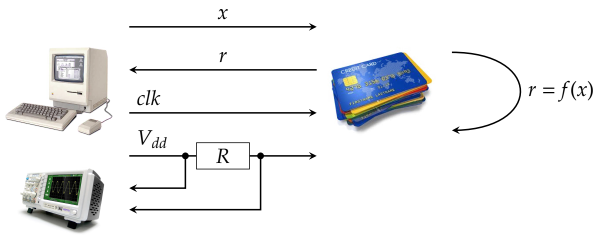
Line #6	:	we know $P \neq G$	$\rightsquigarrow$	disconnect the power, and prevent update to c
Line #9	:	we know $P = G$	$\rightsquigarrow$	do nothing

Notes:

Part 1.2: in theory (1)

Justification: Λ = power consumption

- Example: consider a scenario



whereby

- Ohm's Law tells us that, i.e., $V = IR$, so
- we can acquire a power consumption trace

$$\Lambda = \langle \Lambda_0, \Lambda_1, \ldots, \Lambda_{l-1} \rangle$$

i.e., an l-element sequence of instantaneous samples during execution of f.

Notes:

Part 1.2: in theory (1)

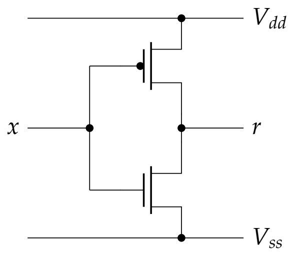
Justification: Λ = power consumption

- Claim: Λ may be
  - *computation*-dependent, i.e., depends on definition and implementation of f, and/or
  - *data*-dependent, i.e., depends on x.

Notes:

Part 1.2: in theory (1)
Justification: Λ = power consumption

- **Why?**
  - From a hardware perspective



power consumption will stem from

1. **static consumption**, and
2. **dynamic consumption**.

  - Therefore, different switching behaviour $\Rightarrow$ different power consumption, i.e.,

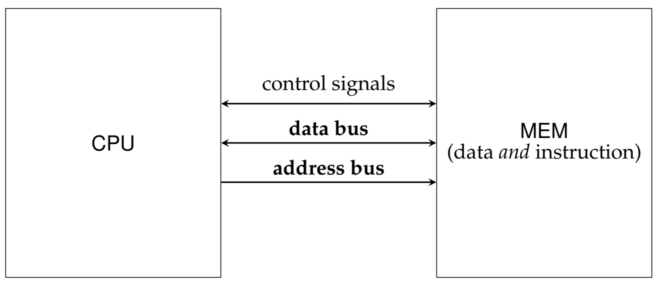
$$\begin{array}{lclcl}
\text{if } x = 0, \text{ setting } x \leftarrow 0 & \Rightarrow & \text{static only} & \Rightarrow & \text{low(er) power consumption} \\
\text{if } x = 0, \text{ setting } x \leftarrow 1 & \Rightarrow & \text{static plus dynamic} & \Rightarrow & \text{high(er) power consumption} \\
\text{if } x = 1, \text{ setting } x \leftarrow 0 & \Rightarrow & \text{static plus dynamic} & \Rightarrow & \text{high(er) power consumption} \\
\text{if } x = 1, \text{ setting } x \leftarrow 1 & \Rightarrow & \text{static only} & \Rightarrow & \text{low(er) power consumption}
\end{array}$$

which is data-dependent, and not *necessarily* in a symmetric manner.

Notes:

Part 1.2: in theory (1)
Justification: Λ = power consumption

- **Why?**
  - From a software perspective



power consumption will stem from

1. computation,
2. communication (i.e., use of buses), and
3. storage (e.g., registers, memory),
4. ...

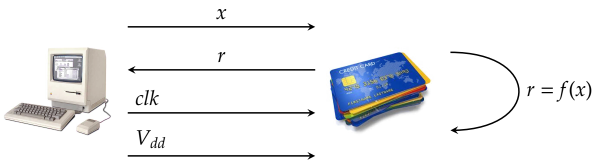
all of which are data-dependent.

Notes:

Part 1.2: in theory (2)

Justification: Λ = execution latency

- **Example**: consider a scenario



whereby

- we measure

$$\begin{array}{rcl} \Lambda_x &=& \text{time when } x \text{ is transmitted} \\ \Lambda_r &=& \text{time when } r \text{ is received} \end{array}$$

  so that
- $\Lambda = \Lambda_r - \Lambda_x$ approximates the execution latency of f.

Notes:

Part 1.2: in theory (2)

Justification: Λ = execution latency

- **Claim**: Λ may be
  - *computation*-dependent, i.e., depends on definition and implementation of f, and/or
  - *data*-dependent, i.e., depends on x.

Notes:

Part 1.2: in theory (2)

Justification: Λ = execution latency

- **Why**? for example, in each of

1.
```
       ...
       if GPR[x] = 0 then PC ← done
       stmt
done : ...
```
a. $\mathsf{GPR}[x] = 0$ so `stmt` is not executed
b. $\mathsf{GPR}[x] = 1$ so `stmt` is executed

2.
```
...
GPR[r] ← MEM[GPR[x]]
...
```
a. $\mathsf{MEM}[\mathsf{GPR}[x]]$ is resident in cache
b. $\mathsf{MEM}[\mathsf{GPR}[x]]$ is not resident in cache

3.
```
...
GPR[r] ← GPR[x] × GPR[y]
...
```
a. $\mathsf{GPR}[x]$ has small magnitude
b. $\mathsf{GPR}[x]$ has large magnitude

it *could* be the case that

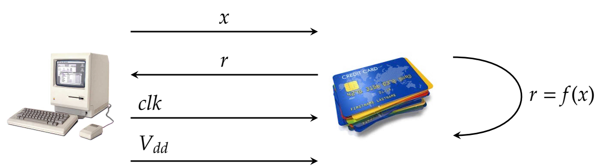
a. $\rightsquigarrow$ low(er) execution latency
b. $\rightsquigarrow$ high(er) execution latency

Notes:

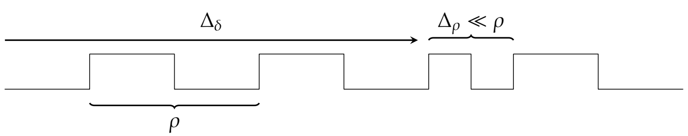
Part 1.2: in theory (3)

Justification: Δ = clock glitch

- **Example**: consider a scenario



whereby a controlled "glitch", i.e.,



such that

- ρ is the clock period,
- Δ_ρ is the period of the glitch,
- Δ_δ is the offset of the glitch.

can be caused in the clock signal *clk*.

Notes:

Part 1.2: in theory (3)

Justification: Δ = clock glitch

- Claim: given

$$\begin{array}{rl} & \cdots \\ & \textbf{if } \mathsf{GPR}[x] = 0 \textbf{ then } \mathsf{PC} \leftarrow \mathit{done} \\ & \texttt{stmt} \\ \mathit{done}: & \cdots \end{array}$$

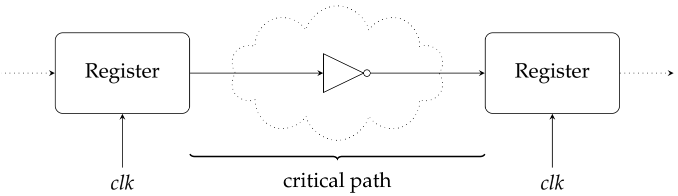
  Δ might allow one to skip the branch instruction, i.e., always execute `stmt`.

Notes:

Part 1.2: in theory (3)

Justification: Δ = clock glitch

- Why?
  - recall that



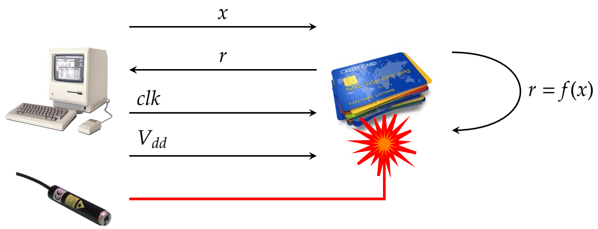
    where, if ρ is close to the critical path, the glitch is likely shorter,
  - therefore, it is plausible such a glitch can prevent complete execution of an instruction, e.g.,
    - $\mathsf{GPR}[x] = 0$ is not computed in time,
    - PC is not updated in time,
    - ...

    meaning that instruction is skipped.

Notes:

Part 1.2: in theory (4)
Justification: Δ = laser pulse

- Example: consider a scenario

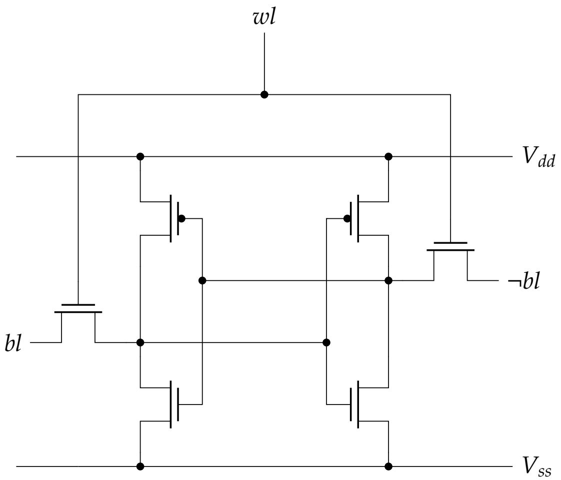


whereby a focused laser pulse can be aimed at the target device.

Notes:

Part 1.2: in theory (4)
Justification: Δ = laser pulse

- Claim: Δ might allow one to toggle the state of



i.e., an SRAM-based memory cell (within some larger device).

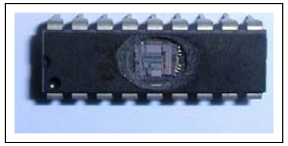
Notes:

Part 1.2: in theory (4)

Justification: Δ = laser pulse

- **Why**?
  - after decapsulation

$\rightsquigarrow$



  at least the top layer of the device is exposed,
  - the laser pulse can ionise regions of semi-conductor material,
  - doing so can be used to activate a transistor,
  - if the bottom-left transistor can be activated (for some short period), this will toggle Q.

https://www.cl.cam.ac.uk/~sps32/ches02-optofault.pdf

Notes:

Part 1.3: in theory (1)

Formalisation: attacks

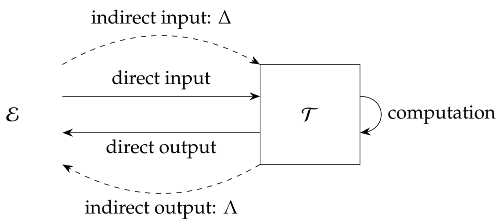
Definition

A **cryptanalytic attack** focuses on exploiting a vulnerability in the abstract, on-paper specification of a target. In contrast, an **implementation attack** focuses on exploiting a vulnerability in the concrete, in-practice implementation of a target by 1) actively influencing and/or 2) passively observing behaviour by it.

Notes:

Definition

Within the following scenario



$\mathcal{E}$ is said to

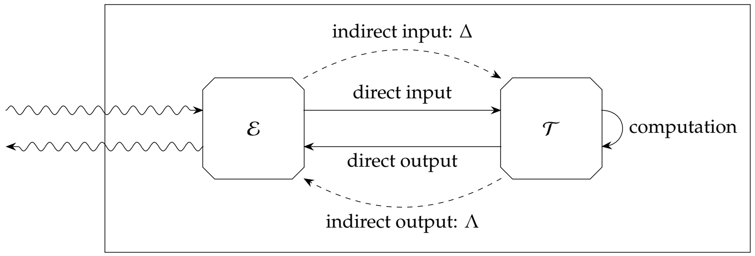
observe $\mathcal{T}$ via Λ $\rightsquigarrow$ **side-channel attack**
influence $\mathcal{T}$ via Δ $\rightsquigarrow$ **fault induction attack**

Notes:

Part 1.3: in theory (1)
Formalisation: attacks

Definition

Within the following scenario



$\mathcal{E}$ is said to

observe $\mathcal{T}$ via Λ $\rightsquigarrow$ **side-channel attack**
influence $\mathcal{T}$ via Δ $\rightsquigarrow$ **fault induction attack**

Notes:

Definition

$\mathcal{E}$ wants to realise some sort of **attack goal**, e.g.,

1.	recovery of state	from	the target
2.	manipulation of state	in	the target
3.	manipulation of behaviour	by	the target

measured relative to both efficacy *and* efficiency.

Notes:

Part 1.3: in theory (2)
Formalisation: attacks

Definition

$\mathcal{E}$ employs an **attack strategy**, which might be (generically) characterised as, e.g.,

1. profiled versus non-profiled
2. adaptive versus non-adaptive
3. differential versus non-differential

which also captures features of standard cryptanalysis, including known plaintext, chosen plaintext, etc.

Definition

$\mathcal{E}$ operates an **attack process**: *typically* this involves

1.	an offline pre-interaction phase	:	characterise, calibrate, pre-compute, etc.
2.	an online interaction phase	:	use input to acquire output
3.	an offline post-interaction phase	:	use input and output to realise goal

Notes:

Part 1.3: in theory (2)
Formalisation: attacks

Definition

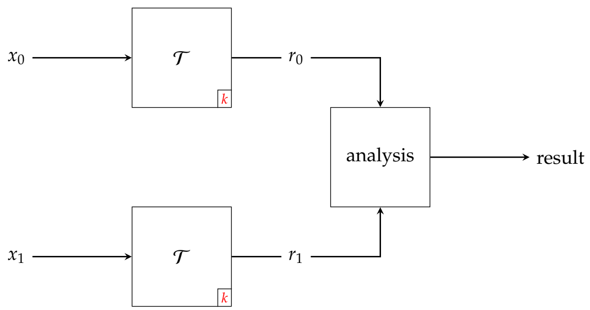
$\mathcal{E}$ employs an **attack mechanism**, which can be (generically) characterised as, e.g.,

1. software versus hardware
2. generic versus specific
3. local versus remote
4. contact-based versus contact-less
5. invasive versus non-invasive
6. destructive versus non-destructive
7. synchronous versus non-synchronous
8. deterministic versus non-deterministic

Notes:

Part 1.3: in theory (3)
Formalisation: attacks

- Note that:
  - a differential cryptanalytic attack [5]

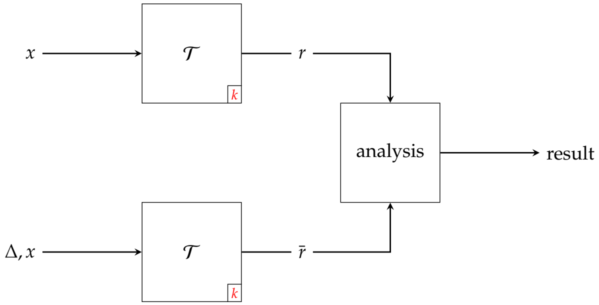


(roughly) analyses how an input difference affects the output difference.

Notes:

Part 1.3: in theory (3)
Formalisation: attacks

- Note that:
  - a differential fault induction attack

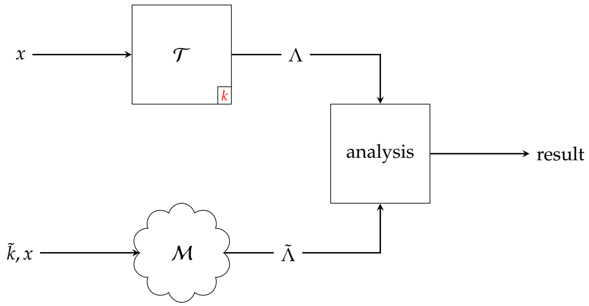


(typically) analyses how a fault affects the output difference.

Notes:

Part 1.3: in theory (3)
Formalisation: attacks

- Note that:
  - a differential side-channel attack



is (typically) such that
- $\mathcal{M}$ is a **model** (or simulation) of $\mathcal{T}$,
- $\tilde{k}$ is a **hypothesis** about (part of) k,
- $\tilde{\Lambda}$ is the **hypothetical leakage** (cf. the *actual* leakage Λ),

and so

$$\begin{array}{rclcll} \text{non-differential} & \Rightarrow & 1 \text{ interaction} & \simeq & \text{analysis within} & \text{single } \Lambda \\ \text{differential} & \Rightarrow & n \text{ interactions} & \simeq & \text{analysis between} & \text{many } \Lambda \end{array}$$

Notes:

Part 1.3: in theory (4)
Formalisation: attacks

Definition

The information leaked via some side-channel is modelled as $\mathcal{M}(\cdot) = \mathcal{M}_d(\cdot) + \mathcal{M}_n$, i.e., as the sum of 1) data-dependent **signal** (of interest) and 2) **noise** components.

Definition

Let V denote a set of values some (intermediate) variable can take, and L denote a set of leakage values.

- A **value**-based leakage model is such that $\mathcal{M}_d : V \rightarrow L$, meaning the leakage value depends on the current value of some variable.
- A **transition**-based leakage model is such that $\mathcal{M}_d : V \times V \rightarrow L$, meaning the leakage value depends on the previous and current value of some variable (i.e., the transition from the former to the latter).

Notes:

Part 1.3: in theory (4)
Formalisation: attacks

Definition

The information leaked via some side-channel is modelled as $\mathcal{M}(\cdot) = \mathcal{M}_d(\cdot) + \mathcal{M}_n$, i.e., as the sum of 1) data-dependent **signal** (of interest) and 2) **noise** components.

Definition

Let V denote a set of values some (intermediate) variable can take, and L denote a set of leakage values.

- A **value**-based leakage model is such that $\mathcal{M}_d : V \rightarrow L$, meaning the leakage value depends on the current value of some variable.
- A **transition**-based leakage model is such that $\mathcal{M}_d : V \times V \rightarrow L$, meaning the leakage value depends on the previous and current value of some variable (i.e., the transition from the former to the latter).

- Example:

  1. Hamming weight $\Rightarrow$ value-based leakage model
  2. Hamming distance $\Rightarrow$ transition-based leakage model

Notes:

Definition

A **fault model** is an abstraction of the fault injection mechanism, i.e., it separates fault *injection* from fault *exploitation*. it captures features such as

1.	timing	⇒	precise control, imprecise control, no control
2.	location	⇒	precise control, imprecise control, no control
3.	duration	⇒	transient, permanent, destructive
4.	plurality	⇒	single fault; multiple, i.e., n faults
5.	granularity	⇒	1 bit, n bits, variable
6.	effect	⇒	set-to-0/1, stuck-at-0/1, flip, randomise, variable
7.	implication	⇒	input data, computation on data, storage of data, execution of instructions

Notes:

Definition

$\mathcal{T}$ might employ a **countermeasure strategy**, which can be (generically) characterised as, e.g.,

1. implicit versus explicit
2. detection versus prevention

and typically forms a layered approach, i.e., a suite of countermeasures versus a single "silver-bullet" or panacea.

Notes:

Definition

$\mathcal{T}$ might design an *abstract* **countermeasure mechanism**, within (at least) the following *levels*

1. **protocol**,
2. **specification**,
3. **implementation**, i.e.,
   - software, and/or
   - hardware.

Definition

$\mathcal{T}$ might implement a *concrete* **countermeasure mechanism**, which can be (generically) characterised as, e.g.,

1. software versus hardware
2. generic versus specific
3. selective versus non-selective
4. proactive versus reactive

Notes:

Definition

Countermeasures against implementation attacks based on information leakage often fall into the following *classes*:

1. **hiding** $\simeq$ decrease SNR, or
2. **masking** $\simeq$ randomised redundant representation.

Notes:

Definition

Among a large design space of countermeasures, instances that focus on hiding (typically) fall into the following sub-classes:

1. increase noise, e.g., make Λ random:
   a. **spatial displacement**, i.e., *where* the operation is computed,
   b. **temporal displacement**, i.e., *when* the operation is computed, which can be further divided into
      - padding (or skewing), and
      - reordering (or shuffling),
   c. **diversified computation**, i.e., *how* the operation is computed,
   d. **obfuscated computation**, e.g., *whether* the operation computed is real or fake (or a dummy).
2. decrease signal, e.g., make Λ constant:
   a. **data-oblivious** (or "**constant-time**") computation of the operation.

Notes:

Definition

Among a large design space of countermeasures, instances that focus on masking (typically) fall into the following sub-classes:

1. **Boolean masking** (or **additive masking**):
$$x \mapsto \hat{x} = \langle \hat{x}[0], \hat{x}[1], \ldots, \hat{x}[d] \rangle$$
   such that
$$x = \hat{x}[0] \oplus \hat{x}[1] \oplus \cdots \oplus \hat{x}[d],$$
   and
2. **arithmetic masking** (or **multiplicative masking**):
$$x \mapsto \hat{x} = \langle \hat{x}[0], \hat{x}[1], \ldots, \hat{x}[d] \rangle$$
   such that
$$x = \hat{x}[0] + \hat{x}[1] + \cdots + \hat{x}[d] \pmod{2^w}.$$

Notes:

Part 1.3: in theory (10)

Formalisation: countermeasures

Definition

Countermeasures against implementation attacks based on fault injection often fall into the following *classes*:

1. **injection-oriented**, e.g.,
   - shielding,
   - sensing,
   - hiding,

   and
2. **exploitation-oriented**, e.g.,
   - duplication,
   - infection,
   - checksum.

Notes:

Part 1.3: in theory (11)

Formalisation: countermeasures

Definition

Among a large design space of countermeasures, instances that focus on exploitation are (typically) parameterised by

1. **type of duplication**, e.g.,
   - temporal duplication: n computations of $f(x)$ in 1 location,
   - spatial duplication: 1 computation of $f(x)$ in n locations,
2. **degree of duplication**,
3. **type of check**, e.g.,
   - direct check: $f(x) \stackrel{?}{=} f(x)$,
   - linearity check: $f(-x) \stackrel{?}{=} -f(x)$,
   - inversion check: $f^{-1}(f(x)) \stackrel{?}{=} x$,
4. **frequency of check**, and
5. **type of action**, e.g.,
   - preventative action: $f(x) \neq f(x) \rightsquigarrow \perp$,
   - infective action: $f(x) \neq f(x) \rightsquigarrow \$$,

and yield an outcome with an associated **detection probability**.

Notes:

Conclusions

- Take away points: implementation attacks
  1. are a potent threat, forming part of a complex attack landscape,
  2. extend well beyond cryptographic targets, posing a more general (cyber-)security challenge,
  3. present significant challenges, e.g., per
     - "attacks only get better" principle,
     - "no free lunch" principle,
     - need to consider multiple layers of abstraction,

     such that "raising the bar" is of use if not ideal,
  4. demand care re. evaluation and/or certification (e.g., FIPS 140-2 [9]) requirements.

Notes:

Additional Reading

- S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.
- P.C. Kocher et al. "Introduction to differential power analysis". In: *Journal of Cryptographic Engineering (JCEN)* 1.1 (2011), pp. 5–27.
- M. Joye and M. Tunstall, eds. *Fault Analysis in Cryptography*. Information Security and Cryptography. Springer, 2012.
- H. Bar-El et al. "The Sorcerer's Apprentice Guide to Fault Attacks". In: *Proceedings of the IEEE* 94.2 (2006), pp. 370–382.
- A. Barenghi et al. "Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures". In: *Proceedings of the IEEE* 100.11 (2012), pp. 3056–3076.
- D. Karaklajić, J.-M. Schmidt, and I. Verbauwhede. "Hardware Designer's Guide to Fault Attacks". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.12 (2013), pp. 2295–2306.
- B. Yuce, P. Schaumont, and M. Witteman. "Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation". In: *Journal of Hardware and Systems Security* 2.2 (2018), pp. 111–130.

Notes:

References

[1] M. Joye and M. Tunstall, eds. *Fault Analysis in Cryptography*. Information Security and Cryptography. Springer, 2012 (see p. 175).

[2] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007 (see p. 175).

[3] H. Bar-El et al. "The Sorcerer's Apprentice Guide to Fault Attacks". In: *Proceedings of the IEEE* 94.2 (2006), pp. 370–382 (see p. 175).

[4] A. Barenghi et al. "Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures". In: *Proceedings of the IEEE* 100.11 (2012), pp. 3056–3076 (see p. 175).

[5] E. Biham and A. Shamir. "Differential Cryptanalysis of DES-like Cryptosystems". In: *Advances in Cryptology (CRYPTO)*. LNCS 537. Springer-Verlag, 1990, pp. 2–21 (see p. 147).

[6] D. Karaklajić, J.-M. Schmidt, and I. Verbauwhede. "Hardware Designer's Guide to Fault Attacks". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.12 (2013), pp. 2295–2306 (see p. 175).

[7] P.C. Kocher et al. "Introduction to differential power analysis". In: *Journal of Cryptographic Engineering (JCEN)* 1.1 (2011), pp. 5–27 (see p. 175).

[8] B. Yuce, P. Schaumont, and M. Witteman. "Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation". In: *Journal of Hardware and Systems Security* 2.2 (2018), pp. 111–130 (see p. 175).

[9] *Security Requirements For Cryptographic Modules*. National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 140-2. 2001. URL: http://csrc.nist.gov (see p. 173).

Notes: