# Side-Channel Attacks and Non-Interference

**Aymeric Fromherz**
Inria Paris,

MPRI 2-30

# Outline

- Last week:
  - Safety and correctness bugs in cryptographic implementations
  - Introduction to the F* proof assistant
- Today:
  - Side-channel attacks
  - Establishing non-interference in implementations
- Exam will be on Feb 27

# Leaking Secrets

*secret s, key k*

```
m <- encrypt(k, s)
send m
```

Assumption: **k is secret**

Implementation:

```
print(k)
let m = encrypt(k, s) in
send(m)
```

# Indirectly Leaking Secrets

```
if k = 0xDEADBEEF then
  print(foo)
else
  print(bar)
let m = encrypt(k, s) in
send(m)
```

# Leaking Information through Observations

```
let verify_pwd(string msg, string pwd) =
  if msg.length <> pwd.length then return false
  for (k = 0; k < msg.length; k ++) {
    if msg[k] <> pwd[k] then return false
  }
  return true
```

Possible attack:

- Measure execution time
- *Observe* longer execution time when msg has the same length as pwd
- *Observe* longer execution time when msg and pwd match on the first k characters

# Side-Channel Attacks

- A ***side-channel attack*** exploits *physical observations* due to running a program to *infer information* about secrets
  - Execution time
  - Power consumption
  - Cache patterns
  - Keyboard sounds
  - …

- Can leak cryptographic keys, plaintexts, state information, …

# Timing Attacks [Kocher, CRYPTO' 96]

- First published side-channel attack on cryptography
- Focuses on modular exponentiation
- Able to find fixed Diffie-Hellman exponents, factor RSA keys, …
- Let's look at this on RSA

# Background on RSA [Rivest, Shamir, Adleman, 78]

- Public-key encryption algorithm (can also be used for signing)
- Relies on a public key *(N, e)*, and a private key *d*
- *N* is the product of two large prime numbers *p* and *q*
- *e* and *d* are related through $ed = 1 \bmod (p - 1)(q - 1)$
- Security relies on *p* and *q* being unknown to the attacker (i.e., factoring large numbers is hard)

# RSA Encryption

- Public key *(N, e),* private key *d,* plaintext *M*
- Encryption: Ciphertext is $M^e \ mod \ N$
- Decryption: We receive a ciphertext *C.* We return $C^d \ mod \ N$

- Correctness: For any plaintext M, decrypt(encrypt(M)) == M
  Mathematically: $(M^e)^d \ mod \ N = M \ mod \ N$
  Proof relies on Fermat's little theorem

- Can also be used for signing:
  - Send $\left(M, M^d \ mod \ N\right)$
  - Anybody can check that $(M^d)^e \ mod \ N = M \ mod \ N$

# Timing Attack on RSA

- Attacker goal: Guess private key *d*
- Attacker capabilities: Can query decryption for any ciphertext C

$C^d \ mod \ N$ implementation (assume d contains w bits):

x = 1

for k = 0 to w – 1 do

    if d[k] = 1 then x = xC mod N

    x = $x^2$ mod N

return x

# Timing Attack on RSA

x = 1

for k = 0 to w – 1 do

  if d[k] = 1 then x = xC mod N

  x = $x^2$ mod N

return x

Example: Take d = 10   (binary: 1010)

(Iteration 0): d[0] = 0

  x = $x^2$ mod N // = 1 mod N

(Iteration 1): d[1] = 1

  x = xC mod N // = C mod N

  x = $x^2$ mod N // = $C^2$ mod N

(Iteration 2): d[2] = 0

  x = $x^2$ mod N // = $C^4$ mod N

(Iteration 3): d[3] = 1

  x = xC mod N // = $C^5$ mod N

  x = $x^2$ mod N // = $C^{10}$ mod N

# Timing Attack on RSA

```
x = 1
for k = 0 to w – 1 do
    if d[k] = 1 then x = xC mod N
    x = x^2 mod N
return x
```

- Attacker goal: Guess d[0]
- Assumption: y mod N is slower for some values of y
    - Ex: When y >= N depending on mod impl

Attack:

- Call decrypt with two ciphertexts $C_1, C_2$, such that $C_1^2 < N$ <= $C_2^2$
- If execution times differ, then d[0] = 1, else d[0] = 0
- In practice, statistical analysis with a family of $C_1, C_2$ to account for noise, network delay, …

# Timing attack on RSA

```
for k = 0 to w – 1 do
    if d[k] = 1 then x = xC mod N
    x = x^2 mod N
return x
```

- Assume d[0], … d[k-1] are known
- Attacker goal: Guess d[k]
- Assumption: y mod N is slower when $N$ <= y

Attack:

- The attacker can compute the first k iterations for any ciphertext C
- Call decrypt with two ciphertexts $C_1, C_2$, such that $x_1^2 < N$ <= $x_2^2$ where $x_1, x_2$ are intermediate results after k iterations for $C_1, C_2$
- If execution times differ, then d[k] = 1, else d[k] = 0

# Timing Attack on RSA

- Recursively applying this methodology, we can guess all bits of *d*
- Original results:
  - 128-bit key could be broken with about 10,000 samples (4 bits/sec)
  - 512-bit key coud be broken in a few minutes with ~350,000 measurements
- Further attacks on optimized RSA implementations intended to circumvent timing attacks also shown effective

  *Remote Timing Attacks are Practical,* Brumley and Boneh, USENIX' 03

# Cache-based Side Channel Attacks

- Exploit timing differences due to accesses to memory caches
- Especially demonstrated on the AES block cipher

Bernstein, D. J. (2005). *Cache-timing attacks on AES.*

Osvik, D. A., Shamir, A., & Tromer, E. (2006). *Cache attacks and countermeasures: the case of AES.*

Bonneau, J., & Mironov, I. (2006). *Cache-collision timing attacks against AES.*

Tromer, E., Osvik, D. A., & Shamir, A. (2010). *Efficient cache attacks on AES, and countermeasures*

# Background on AES

- Block cipher: transforms a fixed-size plaintext (128 bits) into a ciphertext using a secret key *k*
  - Many encryption modes to support arbitrary-sized plaintexts (AES-GCM, AES-CTR, …)
- Initially, xor plaintext with key
- Followed by several rounds of encryption operating on a state of 16 bytes

$$\begin{matrix} p_0 & p_4 & p_8 & p_{12} \\ p_1 & p_5 & p_9 & p_{13} \\ p_2 & p_6 & p_{10} & p_{14} \\ p_3 & p_7 & p_{11} & p_{15} \end{matrix} \xrightarrow{\text{AES Rounds}} \begin{matrix} c_0 & c_4 & c_8 & c_{12} \\ c_1 & c_5 & c_9 & c_{13} \\ c_2 & c_6 & c_{10} & c_{14} \\ c_3 & c_7 & c_{11} & c_{15} \end{matrix}$$

# AES Round

Several Successive Transformations:

- Substitute bytes through affine transformation (SubBytes)
- Different shifts in each row (ShiftRows)
- Apply linear transformation to each column (MixColumns):
- Xor with (a derived sub)key (AddRoundKey): $c_i = p_i'' \oplus k_i$

$$\begin{array}{cccc} p_0 & p_4 & p_8 & p_{12} \\ p_1 & p_5 & p_9 & p_{13} \\ p_2 & p_6 & p_{10} & p_{14} \\ p_3 & p_7 & p_{11} & p_{15} \end{array}$$

$$\Downarrow$$

$$\begin{array}{cccc} p'_0 & p'_4 & p'_8 & p'_{12} \\ p'_1 & p'_5 & p'_9 & p'_{13} \\ p'_2 & p'_6 & p'_{10} & p'_{14} \\ p'_3 & p'_7 & p'_{11} & p'_{15} \end{array}$$

$$\Downarrow$$

$$\begin{array}{cccc} p'_0 & p'_4 & p'_8 & p'_{12} \\ p'_5 & p'_9 & p'_{13} & p'_1 \\ p'_{10} & p'_{14} & p'_2 & p'_6 \\ p'_{15} & p'_3 & p'_7 & p'_{11} \end{array}$$

$$\Downarrow$$

$$\begin{array}{cccc} p''_0 & p''_4 & p''_8 & p''_{12} \\ p''_1 & p''_5 & p''_9 & p''_{13} \\ p''_2 & p''_6 & p''_{10} & p''_{14} \\ p''_3 & p''_7 & p''_{11} & p''_{15} \end{array}$$

# Optimized AES Round

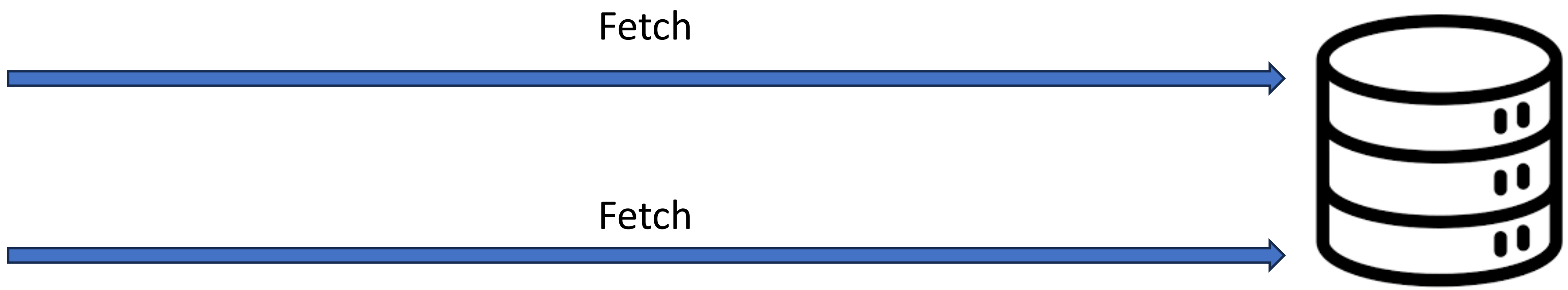
- The first three transformations (SubBytes, ShiftRows, MixColumns) only depend on the input state
- The result can be precomputed for all $p_i$ , and stored in tables $T_k$.

Optimized AES round:

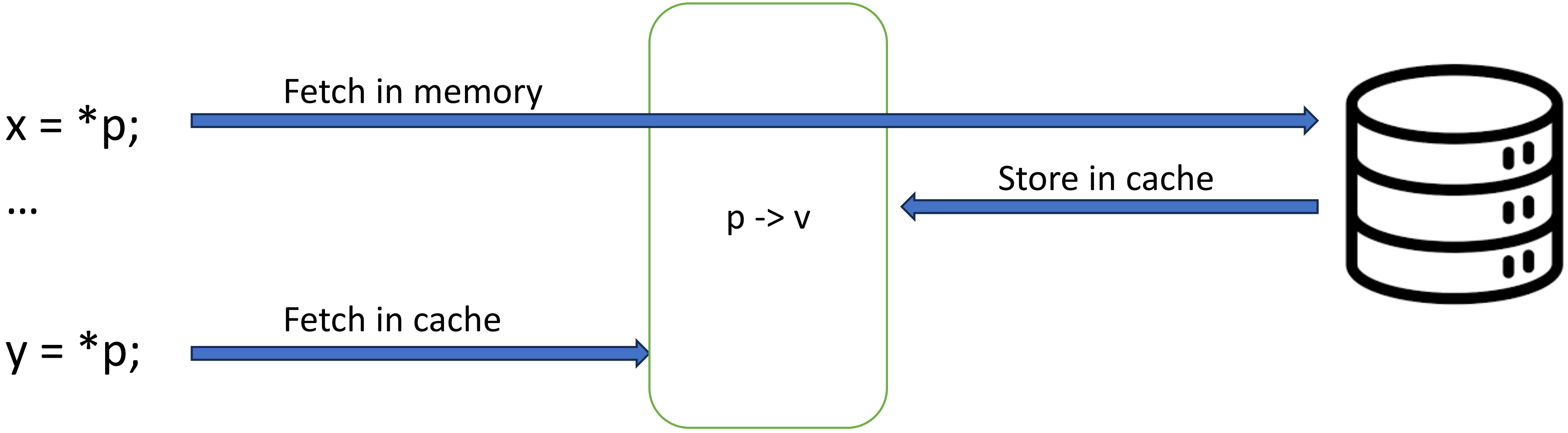
$$
\begin{matrix}
x_0 & x_4 & x_8 & x_{12} \\
x_1 & x_5 & x_9 & x_{13} \\
x_2 & x_6 & x_{10} & x_{14} \\
x_3 & x_7 & x_{11} & x_{15}
\end{matrix}
\quad \longrightarrow \quad
\begin{matrix}
T_0[x_0] \oplus \ T_1[x_5] \oplus \ T_2[x_{10}] \oplus \ T_3[x_{15}] \oplus \{k_0, k_1, k_2, k_3\} \\
T_0[x_4] \oplus \ T_1[x_9] \oplus \ T_2[x_{14}] \oplus \ T_3[x_3] \oplus \{k_4, k_5, k_6, k_7\} \\
T_0[x_8] \oplus \ T_1[x_{13}] \oplus \ T_2[x_2] \oplus \ T_3[x_7] \oplus \{k_8, k_9, k_{10}, k_{11}\} \\
T_0[x_{12}] \oplus \ T_1[x_1] \oplus \ T_2[x_6] \oplus \ T_3[x_{11}] \oplus \{k_{12}, k_{13}, k_{14}, k_{15}\}
\end{matrix}
$$

# Cache Model (Simplified)

```
x = *p;
…
y = *p;
```



# Cache Model (Simplified)




- Accesses to the cache are faster than to main memory
- Storage in the cache is smaller than memory
- When the cache is full, storing a new value removes older mappings

# AES First Round Cache Attack

- For the first round, the inputs $x_i$ are equal to $p_i \oplus k_i$
- We are accessing memory at address $T_k[x_i]$
- The attacker controls input *p*
- We access $T_0[x_0], T_0[x_4], T_0[x_8], T_0[x_{12}]$
- If (e.g.) $x_0 = x_4$, execution time is lower as $T_0[x_4]$ is stored in cache when accessing $T_0[x_0]$
- Trying different samples, we can find values of $p_0, p_4$, such that $x_0 = p_0 \oplus k_0 = x_4 = p_4 \oplus k_4$
- We can determine the value of $k_0 \oplus k_4$

# AES Cache-Based Attacks

- Similar attacks allow to infer more information about the key, leading to key retrieval
- Omitted details
  - Attacker needs to control the initial state of the cache
  - Cache does not allow to reason about lower bits of accessed addresses
  - Other computations can lead to timing differences
- There exists technical solutions for all of this

# Speculative Side-Channel Attacks: Spectre

```
if (0 <= x < a.length) {
  i = a[x];
  r = b[i];
}
```

- Assume that all values in *a* are in [0; b.length[
- Can this code lead to a buffer overflow?

- In theory, no, all accesses are in bound, but…

# CPU Branch Prediction

- CPU instruction pipeline: Fetch, Decode, Execute, Access Memory, Write results in registers
- Modern CPUs anticipate and start executing next instructions early
- When branching occur, CPUs “guess” which branch is most likely to start the instruction pipeline
- When wrong, rollback to earlier CPU state
- Problem: Rollback does not include the entire microarchitectural state, e.g., cache state

# Speculative Side-Channel Attacks: Spectre

```
if (0 <= x < a.length) {
  i = a[x];
  r = b[i];
}
```

- Run program with x = a.length + n
- CPU predicts that the if branch will be taken
- Pre-executes the two memory accesses
- When rolling back, the cache contains a mapping for *i*

- Attack:
  - Train branch predictor for if branch
  - Pick *n* such that a[a.length + n] contains a secret
  - Launch a cache side channel attack to infer i

# Physical Side-Channel Attacks

- Similar attacks exploit the power consumption or electromagnetic leakage.
- Ex: Power consumption of a given instruction is correlated to the number of bits set in its operands (Hamming weight model)
- Infer information about secrets manipulated by the program
- Require some access to the device

# Recent Physical Side-Channel Attacks

*Video-Based Cryptanalysis: Extracting Cryptographic Keys from Video Footage of a Device's Power LED*, Nassi et al., 2023

- **Core idea:**
  - Direct access to device is not needed, a video of its use might be enough
  - The power consumption of a device affects the brightness of its power LED
  - In some cases, this is sufficient to launch a remote power-based side-channel attack

- Today: Focus on *digital* side-channel attacks

# Non-Interference [Goguen-Meseguer, 82]

- Goal: We want to ensure that *secret data* does not impact *public observations* available to an attacker

- Information-flow property based on *secrecy labels*:
  - High (H) == Secret data
  - Low (L) == Public data

- High-level idea: There is no flow from high data to low data

# Non-Interference, Formally

For a given program *p*,

$\forall\ (s_1\ s_2 \colon state)$,

$s_{1|\,L} = s_{2|\,L} \Rightarrow$ // States agree on low values

$s_1 \rightarrow_p^* s_1' \Rightarrow$ // Executing *p* in $s_1$ yields $s_1'$

$s_2 \rightarrow_p^* s_2' \Rightarrow$ // Executing *p* in $s_2$ yields $s_2'$

$s'_{1|\,L} = s'_{2|\,L}$ // Results agree on low values

# Non-Interference Example

if x = 1 then y := 1 else y := 0

- If x : H, y : H: No low values, non-interference
- If x : L, y : L: Initial agreement on x, non-interference
- If x : L, y : H: Initial agreement on x, non-interference
- If x : H, y : L: Observing the result of y leaks information about x

- Goal: Statically ensure noninterference

# Non-Interference by Typing [Volpano et al., 96]

$$
\begin{array}{llll}
(\textit{expressions}) & e & ::= & x \mid l \mid n \mid e + e' \mid e - e' \mid e = e' \mid e < e' \\
(\textit{commands}) & c & ::= & e := e' \mid c; c' \mid \textbf{if } e \textbf{ then } c \textbf{ else } c' \mid \\
 & & & \textbf{while } e \textbf{ do } c \mid \textbf{letvar } x := e \textbf{ in } c
\end{array}
$$

$$
\begin{array}{llll}
(\textit{data types}) & \tau & ::= & s \\
(\textit{phrase types}) & \rho & ::= & \tau \mid \tau\ \textit{var} \mid \tau\ \textit{cmd}
\end{array}
$$

- Data types *s* are security labels (in our case, H and L)
- Each expression and command is annotated with a security label

# Typing Judgement

$$\lambda; \gamma \vdash p : \rho$$

- $\lambda$ is a memory store: It associates to each *location* its security label
- $\gamma$ is a variable environment: It maps variables to their type
- Under this context, this judgement gives program $p$ the type $\rho$

# Typing Rules

$$(\textsc{int}) \qquad \lambda; \gamma \vdash n : \tau$$

$$(\textsc{var}) \qquad \lambda; \gamma \vdash x : \tau\ var \qquad \text{if } \gamma(x) = \tau\ var$$

$$(\textsc{varloc}) \qquad \lambda; \gamma \vdash l : \tau\ var \qquad \text{if } \lambda(l) = \tau$$

$$(\textsc{arith}) \quad \frac{\begin{array}{l}\lambda; \gamma \vdash e : \tau, \\ \lambda; \gamma \vdash e' : \tau\end{array}}{\lambda; \gamma \vdash e + e' : \tau} \qquad (\textsc{assign}) \quad \frac{\begin{array}{l}\lambda; \gamma \vdash e : \tau\ var, \\ \lambda; \gamma \vdash e' : \tau\end{array}}{\lambda; \gamma \vdash e := e' : \tau\ cmd}$$

# Typing Rules

$$(\textsc{compose}) \quad \frac{\lambda; \gamma \vdash c : \tau\ cmd, \quad \lambda; \gamma \vdash c' : \tau\ cmd}{\lambda; \gamma \vdash c;\ c' : \tau\ cmd}$$

$$(\textsc{if}) \quad \frac{\lambda; \gamma \vdash e : \tau, \quad \lambda; \gamma \vdash c : \tau\ cmd, \quad \lambda; \gamma \vdash c' : \tau\ cmd}{\lambda; \gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c' : \tau\ cmd}$$

$$(\textsc{while}) \quad \frac{\lambda; \gamma \vdash e : \tau, \quad \lambda; \gamma \vdash c : \tau\ cmd}{\lambda; \gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ c : \tau\ cmd}$$

# Typing Example

if x = 1 then y := 1 else y := 0

Assume that x : H var, y : H var

Goal : Give this program the type *H cmd*

# Typing Example

Goal: x: H var, y: H var ⊢ if x = 1 then y := 1 else y := 0 : *H cmd*

$$
(\text{IF}) \quad \frac{\begin{array}{l}\lambda;\gamma \vdash e : \tau, \\ \lambda;\gamma \vdash c : \tau\ cmd, \\ \lambda;\gamma \vdash c' : \tau\ cmd\end{array}}{\lambda;\gamma \vdash \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c' : \tau\ cmd}
$$

Need to prove

- x: H var, y : H var ⊢ x = 1 : H
- x: H var, y : H var ⊢ y := 1 : H cmd
- x: H var, y : H var ⊢ y := 0 : H cmd

# Typing Example

Goal: x: H var, y: H var ⊢ x = 1 : H

$$(\text{ARITH}) \quad \frac{\lambda; \gamma \vdash e : \tau, \quad \lambda; \gamma \vdash e' : \tau}{\lambda; \gamma \vdash e + e' : \tau}$$

Need to prove

- x: H var, y : H var ⊢ 1 : H
- x: H var, y : H var ⊢ x : H

$$(\text{INT}) \quad \lambda; \gamma \vdash n : \tau$$

$$(\text{VAR}) \quad \lambda; \gamma \vdash x : \tau\ var \quad \text{if } \gamma(x) = \tau\ var$$

$$(\text{R-VAL}) \quad \frac{\lambda; \gamma \vdash e : \tau\ var}{\lambda; \gamma \vdash e : \tau}$$

# Typing Example

Goal: x: H var, y : H var ⊢ y := 1 : *H cmd*

$$(\text{ASSIGN}) \quad \frac{\lambda; \gamma \vdash e : \tau\ var, \quad \lambda; \gamma \vdash e' : \tau}{\lambda; \gamma \vdash e := e' : \tau\ cmd}$$

Need to prove

- x: H var, y : H var ⊢ y : H var $\quad (\text{VAR}) \quad \lambda; \gamma \vdash x : \tau\ var \quad \text{if } \gamma(x) = \tau\ var$
- x: H var, y : H var ⊢ 1 : H $\quad (\text{INT}) \quad \lambda; \gamma \vdash n : \tau$

# Label Subtyping

- The type system is sufficient when *x* and *y* have the same label
- What about x : L var, y : H var ?

$$(\text{IF}) \qquad \frac{\begin{array}{l}\lambda;\gamma \vdash e : \tau, \\ \lambda;\gamma \vdash c : \tau\ cmd, \\ \lambda;\gamma \vdash c' : \tau\ cmd\end{array}}{\lambda;\gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c' : \tau\ cmd}$$

- The If rule requires the condition and the commands to have the same label!

# Label Subtyping

$$\text{(BASE)} \quad \frac{\tau \leq \tau'}{\vdash \tau \subseteq \tau'} \qquad\qquad \text{(SUBTYPE)} \quad \frac{\lambda; \gamma \vdash p : \rho, \quad \vdash \rho \subseteq \rho'}{\lambda; \gamma \vdash p : \rho'}$$

- We consider that label L is “lower” than label ”H”
- Models that a public value can always be hidden as secret

- Given x = 0 : L, this allows us to derive x = 0 : H

# Label Subtyping

$$(\text{CMD}^-) \qquad \frac{\vdash \tau \subseteq \tau'}{\vdash \tau' \ cmd \subseteq \tau \ cmd}$$

- Different variance compared to expression rule
- Intuitively: If a program is "secure" when operating on/accessing secret variables, then it is also when accessing less privileged data
- Alternative proof: y := 1 : H cmd => y := 1 : L cmd

# Exercises

- For the following programs, either give a typing derivation showing non-interference, or explain why the program does not typecheck

- $x$: L var, $y$: H var $\vdash$ while ($x < 10$) do ($x := x + 1$; $y := y + 1$)

- $x$: H var, $y$: L var $\vdash$ while ($x < 10$) do
  if $y = 2$ then $x := x + 1$ else $x := x + 2$

# Back to Digital Side-Channels

- The typing approach so far avoids indirect leaks, e.g., by observing public values
- However, it allows typechecking if key = … then x = …, which leaks the key by observing the timing of the attack

- Need to extend formalism beyond leaking values!

# Instrumenting Semantics

- Previously: $s_1 \rightarrow_p^* s_1'$
- We record traces containing all branching and memory accesses

(Trace) $l$ ::= $\varepsilon$ | Branch (b) . $l$ | Access(n) . $l$

$$s_1 \rightarrow_p^* s_1', l_1$$

When executing *if b then p else p' ,* we record Branch(b)

When executing a[n], we record Access(n)

# Non-Intereference with Observations

For a given program *p*,

$$\forall\ (s_1\ s_2 : state),$$

$$s_{1|\,L} = s_{2|\,L} \Rightarrow s_1 \rightarrow_p^* s_1', l_1 \Rightarrow s_2 \rightarrow_p^* s_2', l_2 \Rightarrow$$
$$s'_{1|\,L} = s'_{2|\,L} \land l_1 = l_2$$

Captures that the program executes the same program paths, and performs identical memory (and hence cache) accesses for the same attacker-controlled inputs

# The “Constant-Time” Programming Discipline

Cryptographic implementations must follow a “constant-time” programming discipline, which forbids

- Branching involving secrets
- Using instructions which execute in variable time with secrets (e.g., division)
- Accessing memory based on secret indices

# The “Constant-Time” Programming Discipline

- Is this enough?

*System-level Non-interference for Constant-time Cryptography,* Barthe et al., CCS’ 14 studies this formally

- Easy programming discipline to follow?

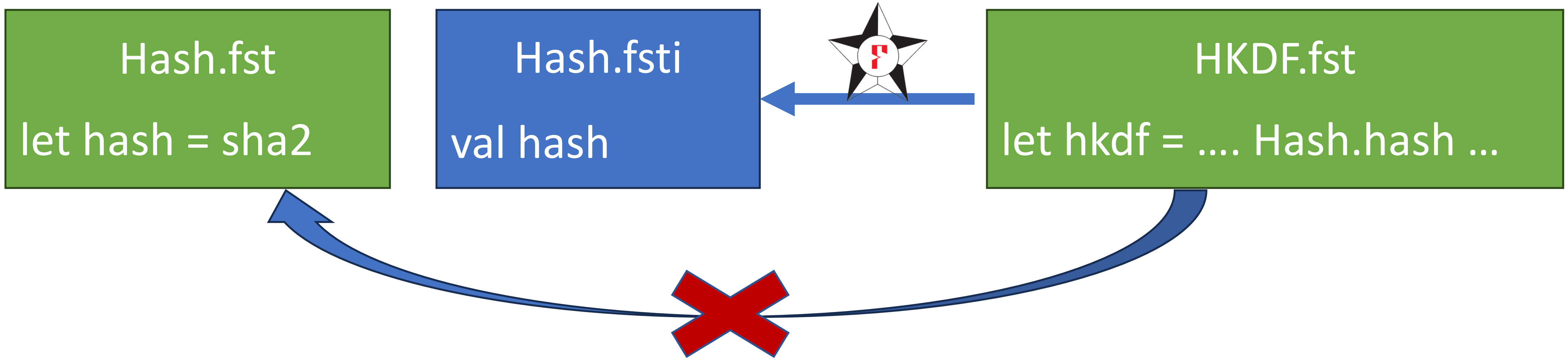
  Jan 2024: **KyberSlash: division timings depending on secrets in Kyber software**

  https://kyberslash.cr.yp.to/ , https://cryspen.com/post/ml-kem-implementation/

- We need tools to enforce this

# Non-Interference by Typing Abstraction

- Remember from last week:




- Client modules only have access to the interface
- Underlying implementation is hidden (true for other languages supporting abstraction)

# Non-Interference by Typing Abstraction

SUInt32.fsti

```
val suint32: Type          // Abstract type for secret uint32 integers

val (+) : suint32 -> suint32 -> suint32
val (*) : suint32 -> suint32 -> suint32
// Non-constant time operations are not exposed
// val (/) : suint32 -> suint32 -> suint32
```

# Implementing Abstract Secret Integers

SUInt32.fst

```
let suint32 = uint32     // Underlying definition is simply standard integers

let (+) n1 n2 = n1 + n2
let (*) n1 n2 = n1 * n2
```

- Abstract type for opaque “secret integers”
- Exposes arithmetic and bitwise constant-time operations, but not comparison, division
- After extraction, compiled to standard integer, no runtime cost

# Using Secret Integers

```
n1, n2 : suint32          // Secret integers

if n1 > n2 then …
```

No comparison defined for secret integers

```
val index (b: array uint8) (i: uint32) : …

let x = b.[n1] in …
```

Expected type uint32, got type suint32

- Can be seen as an extension of previous typing discipline

# Typing Limitations

- Only guarantees resistance against timing and cache-based side-channels (variants exist for speculative side-channels)
- Only provides guarantees within the semantics of the source language (C, OCaml, …)
- Compilers can reintroduce side-channels

# Compiler-Induced Side Channels

```
let login() =
    x = read_passwd()
    res = check_pwd(x)
    x = 0
    return res
```

Compile

```
let login() =
    x = read_passwd()
    res = check_pwd(x)
    return res
```

Unused assignment

Password can leak after execution!

# Crypto Compiler-Induced Side Channels

Assume b is secret

if b then r := x else r := y

Rewrite into constant-time version

int mask = create_mask(b);

r := (x & mask) | (y & ~mask);

```
[@@ Comment "Returns 2^64 - 1 if a = b, otherwise returns 0.
static inline uint64_t FStar_UInt64_eq_mask(uint64_t a, uint64_t b)
{
  uint64_t x = a ^ b;
  uint64_t minus_x = ~x + (uint64_t)1U;
  uint64_t x_or_minus_x = x | minus_x;
  uint64_t xnx = x_or_minus_x >> (uint32_t)63U;
  return xnx - (uint64_t)1U;
}
```

LLVM COMPILER INFRASTRUCTURE : Did you mean

if b then r := x else r := y

# Avoiding Compiler-Induced Side-Channels

- Several solutions:
  - Use a constant-time preserving compiler

    e.g., *Formal verification of a constant-time preserving C compiler,* Barthe et al., POPL' 20
  - Analyze binary code after compilation

    *Verifying constant time implementations*, Almeida et al., USENIX' 16

    *BINSEC/REL: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level,* Daniel et al., S&P' 20

    …

# Non-Interference by Taint Analysis

- Taint analysis: a static analysis for non-interference
- **Core idea:** Mark some inputs as secret ("taint" them)
- Static analysis *propagates* the taint throughout the program
- If taint is propagated to attacker-observable components, raise error

# Taint Analysis Example

```
let f (x : int) =
    y := x;
    z := 0;
    w := z + y;
```

```
let f (x : int) =
    y := x;
    z := 0;
    w := z + y;
```

- Mark input x as secret
- Propagate taint through program

# Taint Analysis: Join Operator

```
let f (x : int, p: int) =
  z := p;
  if z > 0
    y := x;
  else
    y := 0;
  w := z + y;
```

```
let f (x : int, p: int) =
  z := p;
  if z > 0
    y := x;
  else
    y := 0;
  w := z + y;
```

- When joining two execution paths, we take the ”highest” value for each variable

# Taint Analysis: Raising Errors

```
let f (x : int) =
  c := x + 2;
  if c > 0
    y := 1;
  else
    y := 2;
```

```
let g (x : int, a: int[]) =
  y := a[x];
```

```
let f (x : int) =
  c := x + 2;
  if c > 0
    y := 1;
  else
    y := 2;
```

```
let g (x : int, a: int[]) =
  y := a[x];
```

# Taint Analysis: Erasing Taint

```
let f (x : int) =
  i := x + 2;
  c := xor(x, x);
  if c > 0
    y := 1;
  else
    y := 2;
```

```
let f (x : int) =
  i := x + 2;
  c := xor(x, x);
  if c > 0
    y := 1;
  else
    y := 2;
```

- While tainted in theory, the output of some operations does not depend on its inputs
- We can soundly erase the taint in these cases

# Taint Analysis: Memory Accesses

```
let f (x : int, y: int, a: int[]) =
  a[0] := x;
  c := a[y];
  if c > 0
    y := 1;
  else
    y := 2;
```

- Is this program constant-time?
- Depends on the values of y

# Taint Analysis: Memory Accesses

```
let f (x : int, y: int, a: int[]) =
  a[0] := x;
  if y > 0
    c := a[y];
  else
    c := 2;
   if c > 0 …
```

- Is this program constant-time?
- Yes, however tracking this requires tracking information about possible values of y
- We need a precise analysis to avoid false positives

# Taint Analysis: Memory Accesses

```
let f (x : int, p1: *int, p2: *int) =
  *p1 := x;
  y := *p2;
  if y > 0 ...
```

- Is this program constant-time?
- Depends on whether p1 and p2 alias
- We need aliasing information, either inferred (points-to analysis) or provided by programmer

# Taint Analysis: Summary

- Mark secret inputs as “tainted”
- Propagate taint throughout the program
- If the taint reaches an attacker observation (return value, branching, memory access), possible secret leak
- Main difficulty: Reasoning about memory, which requires specific analyses
- Can be done on a variety of languages, including assembly
- Applicable beyond constant-time reasoning, e.g., to track possible leaks of private user information

*Certification of Programs for Secure Information Flow*, Denning and Denning, CACM’ 77