

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

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 $M^e \bmod N$
- Decryption: We receive a ciphertext C . We return $C^d \bmod N$
- Correctness: For any plaintext M , $\text{decrypt}(\text{encrypt}(M)) == M$
Mathematically: $(M^e)^d \bmod N = M \bmod N$
Proof relies on Fermat's little theorem
- Can also be used for signing:
 - Send $(M, M^d \bmod N)$
 - Anybody can check that $(M^d)^e \bmod N = M \bmod N$

Timing Attack on RSA

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

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

$x = 1$

for $k = 0$ to $w - 1$ do

 if $d[k] = 1$ then $x = xC \bmod N$

$x = x^2 \bmod N$

return x

Timing Attack on RSA

$x = 1$

for $k = 0$ to $w - 1$ do

 if $d[k] = 1$ then $x = xC \bmod N$

$x = x^2 \bmod N$

return x

Example: Take $d = 10$ (binary: 1010)

(Iteration 0): $d[0] = 0$

$$x = x^2 \bmod N // = 1 \bmod N$$

(Iteration 1): $d[1] = 1$

$$x = xC \bmod N // = C \bmod N$$

$$x = x^2 \bmod N // = C^2 \bmod N$$

(Iteration 2): $d[2] = 0$

$$x = x^2 \bmod N // = C^4 \bmod N$$

(Iteration 3): $d[3] = 1$

$$x = xC \bmod N // = C^5 \bmod N$$

$$x = x^2 \bmod N // = C^{10} \bmod N$$



Timing Attack on RSA

$x = 1$

for $k = 0$ to $w - 1$ do

 if $d[k] = 1$ then $x = xC \bmod N$

$x = x^2 \bmod N$

return x

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

Attack:

- Call decrypt with two ciphertexts C_1, C_2 , such that $C_1^2 < N \leq C_2^2$
- 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 $x = xC \bmod N$

$x = x^2 \bmod N$

return x

- Assume $d[0], \dots, d[k-1]$ are known
- Attacker goal: Guess $d[k]$
- Assumption: $y \bmod N$ is slower when $N \leq y$

Attack:

- The attacker can compute the first k iterations for any ciphertext C
- Call decrypt with two ciphertexts C_1, C_2 , such that $x_1^2 < N \leq x_2^2$ 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 could 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*.

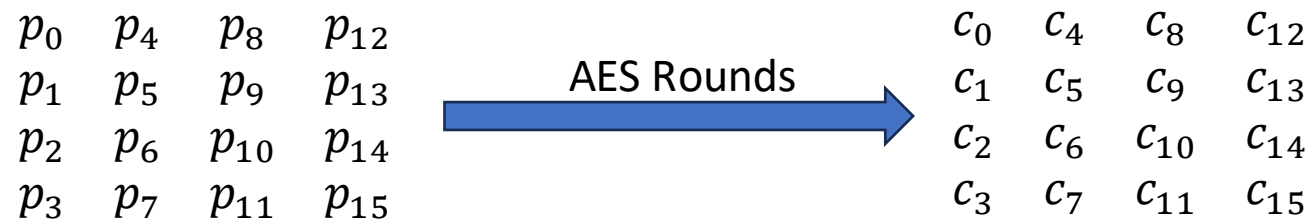
Osvik, D. A., Shamir, A., & Tromer, E. (2006). *Cache attacks and countermeasures: the case of AES*.

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

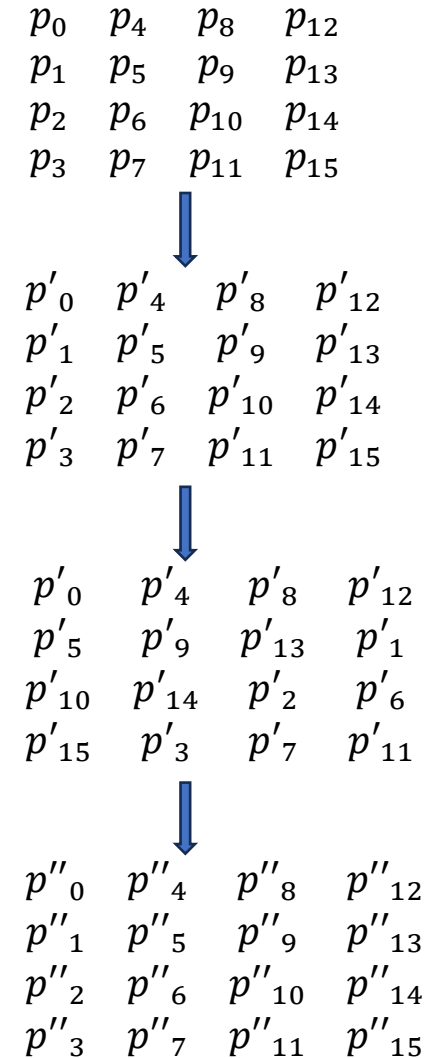
- Block cipher: transforms a fixed-size plaintext (128 bits) into a ciphertext using a secret key k
 - Many encryption modes to support arbitrary-sized plaintexts (AES-GCM, AES-CTR, ...)
- Initially, xor plaintext with key
- Followed by several rounds of encryption operating on a state of 16 bytes



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): $c_i = p_i'' \oplus k_i$



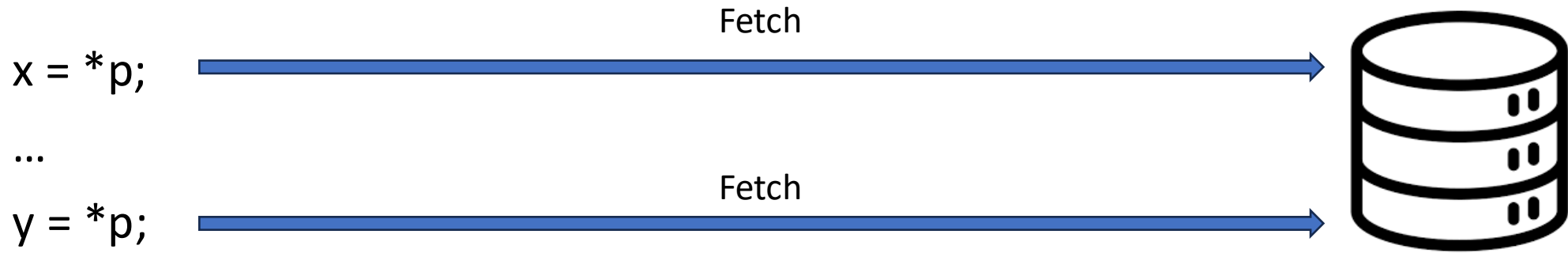
Optimized AES Round

- 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 .

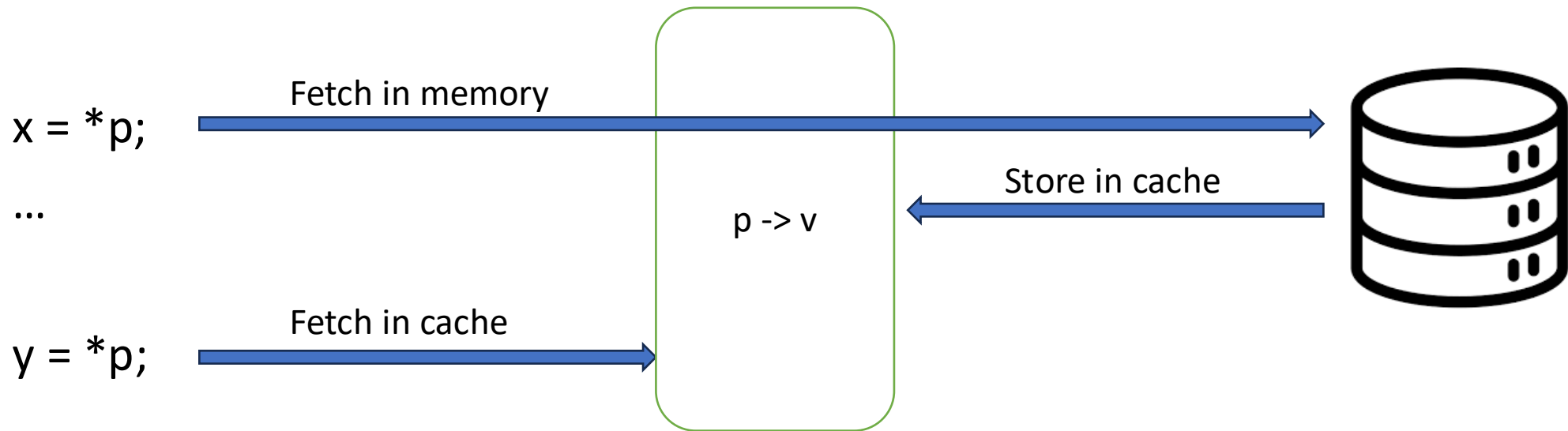
Optimized AES round:

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

Cache Model (Simplified)



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 ,

$\forall (s_1 \ s_2: \text{state}),$

$$s_{1|L} = s_{2|L} \Rightarrow$$

// 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]

(*expressions*) $e ::= x \mid l \mid n \mid e + e' \mid e - e' \mid e = e' \mid e < e'$
(*commands*) $c ::= e := e' \mid c; c' \mid \mathbf{if\ } e \mathbf{\ then\ } c \mathbf{\ else\ } c' \mid$
 $\mathbf{while\ } e \mathbf{\ do\ } c \mid \mathbf{letvar\ } x := e \mathbf{\ in\ } c$

(*data types*) $\tau ::= s$

(*phrase types*) $\rho ::= \tau \mid \tau \textit{ var} \mid \tau \textit{ cmd}$

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

Typing Judgement

$$\gamma \vdash p : \rho$$

- γ is a variable environment: It maps variables to their type
- Under this context, this judgement gives program p the type ρ

Typing Rules

(INT) $\gamma \vdash n : \tau$

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

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

Typing Rules

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

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

Typing Rules

$$\text{(COMPOSE)} \quad \frac{\gamma \vdash c : \tau \text{ cmd}, \quad \gamma \vdash c' : \tau \text{ cmd}}{\gamma \vdash c; c' : \tau \text{ cmd}}$$

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

$$\text{(WHILE)} \quad \frac{\gamma \vdash e : \tau, \quad \gamma \vdash c : \tau \text{ cmd}}{\gamma \vdash \mathbf{while} \ e \ \mathbf{do} \ c : \tau \text{ cmd}}$$

Typing Example

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

Assume that $x : H \text{ var}$, $y : H \text{ var}$

Goal : Give this program the type $H \text{ cmd}$

Typing Example

Goal: $x: H \text{ var}, y: H \text{ var} \vdash \text{if } x = 1 \text{ then } y := 1 \text{ else } y := 0 : H \text{ cmd}$

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

Need to prove

- $x: H \text{ var}, y : H \text{ var} \vdash x = 1 : H$
- $x: H \text{ var}, y : H \text{ var} \vdash y := 1 : H \text{ cmd}$
- $x: H \text{ var}, y : H \text{ var} \vdash y := 0 : H \text{ cmd}$

Typing Example

Goal: $x: H \text{ var}, y: H \text{ var} \vdash x = 1 : H$

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

Need to prove

- $x: H \text{ var}, y : H \text{ var} \vdash 1 : H$

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

- $x: H \text{ var}, y : H \text{ var} \vdash x : H$

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

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

Typing Example

Goal: $x: H \text{ var}, y: H \text{ var} \vdash y := 1 : H \text{ cmd}$

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

Need to prove

- $x: H \text{ var}, y: H \text{ var} \vdash y : H \text{ var}$

- $x: H \text{ var}, y: H \text{ var} \vdash 1 : H$ (INT) $\gamma \vdash n : \tau$



Label Subtyping

- The type system is sufficient when x and y have the same label
- What about $x : L \text{ var}$, $y : H \text{ var}$?

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

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

Label Subtyping

$$\text{(BASE)} \quad \frac{\tau \leq \tau'}{\vdash \tau \subseteq \tau'}$$

$$\text{(SUBTYPE)} \quad \frac{\gamma \vdash p : \rho, \quad \vdash \rho \subseteq \rho'}{\gamma \vdash p : \rho'}$$

- 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

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

- Different variance compared to expression rule
- Intuitively: If a program is “secure” in a context which might depend on secret data, then it is also in a less privileged context
- Alternative proof: $y := 1 : H \text{ cmd} \Rightarrow y := 1 : L \text{ cmd}$

Exercises

- For the following programs, either give a typing derivation showing non-interference, or explain why the program does not typecheck
- $x: L \text{ var}, y: H \text{ var} \vdash \text{while } (x < 10) \text{ do } (x := x + 1; y := y + 1)$
- $x: H \text{ var}, y: L \text{ var} \vdash \text{while } (x < 10) \text{ do}$
 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 $\text{key} = \dots$ then $x = \dots$, 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) $l ::= \varepsilon \mid \text{Branch}(b) . l \mid \text{Access}(n) . l$

$$s_1 \rightarrow_p^* s'_1, l_1$$

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

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

Non-Interference with Observations

For a given program p ,

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

$$s_{1|L} = s_{2|L} \Rightarrow s_1 \xrightarrow{*}_p s'_1, l_1 \Rightarrow s_2 \xrightarrow{*}_p s'_2, l_2 \Rightarrow \\ s'_{1|L} = s'_{2|L} \wedge l_1 = l_2$$

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**

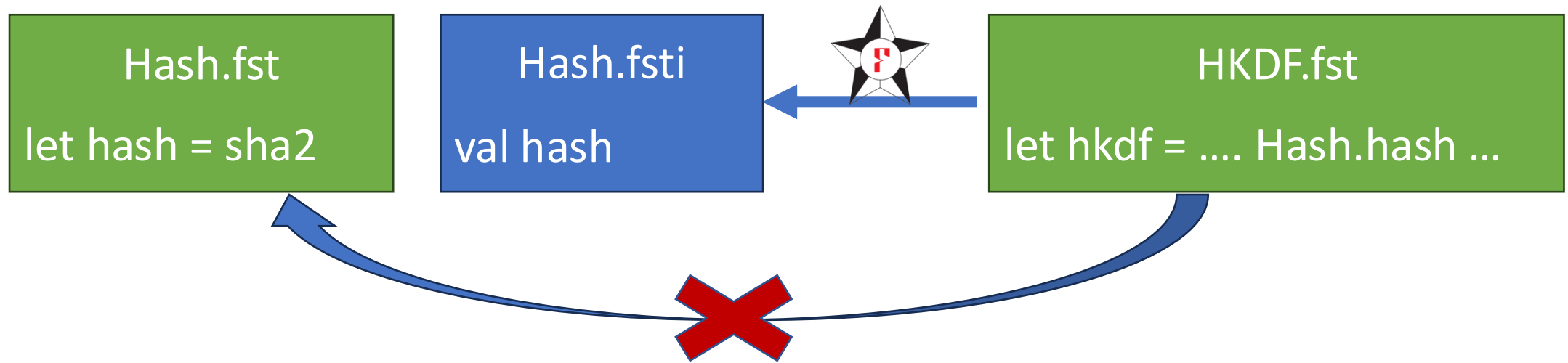
<https://kyberslash.cryp.to/> , <https://cryspen.com/post/ml-kem-implementation/>

KyberSlash: Exploiting secret-dependent division timings in Kyber implementations, Bernstein et al., CHES’ 25

- 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 uint32 = uint32    // Underlying definition is simply standard integers
```

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

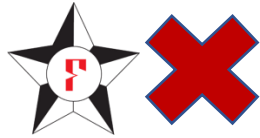
```
let (*) n1 n2 = n1 * n2
```

- 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 : uint32` *// Secret integers*

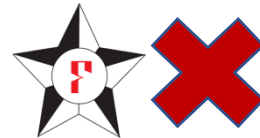
`if n1 > n2 then ...`



No comparison defined for secret integers

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

`let x = b.[n1] in ...`



Expected type uint32, got type uint32

- Can be seen as an extension of previous typing discipline

Speculative Execution



```
if i < 10 {  
    x = p[i];  
}  
w[x] = 0;
```

Spectre v1-read



```
if i < 5 {  
    s[i] = sec;  
}  
x = p[0];  
w[x] = 0;
```

Spectre v1-write

Protecting Against Speculative Execution

```
if i < 10 {  
    x = p[i];  
}  
w[x] = 0;
```



```
if i < 10 {  
    fence();  
    x = p[i];  
}  
w[x] = 0;
```

Need to insert a fence at each branch
Large overhead

Protecting Against Speculative Execution

```
if i < 10 {  
    x = p[i];  
}  
w[x] = 0;
```



```
if i < 10 {  
    x = p[i];  
    protect(x);  
}  
w[x] = 0;
```



```
if i < 10 {  
    x = p[i];  
}  
protect(x);  
w[x] = 0;
```

Protect Semantics

- We rely on a specific variable, **ms**
- y = protect(x, ms):** “conditional masking”
- -1 if **ms** = -1
 - no-op otherwise

Need to set ms when misspeculating: **set_ms(cond)**

- set_ms(cond) sets **ms** to -1 if cond is false
- no-op otherwise

Protecting Against Speculative Execution

```
if i < 10 {  
    x = p[i];  
}  
w[x] = 0;
```



```
if i < 10 {  
    set_ms(i < 10);  
    x = p[i];  
    x = protect(x, ms);  
}  
w[x] = 0;
```

How to ensure this protects against speculative attacks?

A Type-System for Speculative Constant-Time

[Shivakumar et al., 23]

- Type systems for constant-time had one security label, **L** or **H**
- Idea: Extend it with a pair of labels τ_n, τ_s which are either **L** or **H**
- τ_n : security label for “normal” executions
- τ_s : security label for speculative executions

Typing Rules

$$\begin{array}{c} \text{VAR} \\ \Gamma \vdash x : \Gamma(x) \end{array}$$

$$\begin{array}{c} \text{OP} \\ \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \\ \hline \Gamma \vdash op(e_1, e_2) : \tau_1 \cup \tau_2 \end{array}$$

- $\mathbf{L \cup H = H, \quad L \cup L = L, \quad H \cup H = H}$
- $(\tau_n, \tau_s) \cup (\tau_n', \tau_s') = (\tau_n \cup \tau_n', \tau_s \cup \tau_s')$

Typing Rules

CONST

$$\Gamma \vdash n : (L, L)$$

SUB

$$\frac{\Gamma \vdash e : \tau \quad \tau \leq \tau'}{\Gamma \vdash e : \tau'}$$

- **L ≤ H**

- $(\tau_n, \tau_s) \leq (\tau_n', \tau_s') \iff \tau_n \leq \tau_n' \wedge \tau_s \leq \tau_s'$

Typing Rules: Speculative Load

$$\frac{\text{LOAD} \quad \Gamma \vdash i : (L, L) \quad \Gamma(a) = (\tau_n, \tau_s)}{\Gamma \vdash x = a[i] : \Gamma[x \leftarrow (\tau_n, H)]}$$

Typing Rules: Protect

$y = \text{protect}(x, ms)$

Recall: Behaviour depends on **ms** !

Conceptually, “ y is protected against speculative attacks if **ms** accurately models the current state of misspeculation”

Need to keep track of the state of ms !

Typing Rules: Execution Modes

Idea: Keep track of the relationship between **ms** and misspeculation in a mode Σ

$\Sigma := \mid \mathbf{unk} \mid \mathbf{ms} \mid \mathbf{ms}_{|e}$

- **ms**: If misspeculation, then $ms = -1$
- **unk**: No information about the current state
- **ms**_{|e} : If misspeculation and e is true, then $ms = -1$

Typing Rules: Protect and Set-ms

PROTECT

$$\frac{\Gamma' = \Gamma[y \leftarrow (\Gamma_n(x), \Gamma_n(x))]}{\text{ms}, \Gamma \vdash y = \text{protect}(x, \text{ms}) : \text{ms}, \Gamma'}$$

SET-MS

$$\text{ms}|_e, \Gamma \vdash \text{ms} = \text{set_ms}(e) : \text{ms}, \Gamma$$

Typing Rules: Load

LOAD

$$\frac{\Gamma \vdash i : (L, L) \quad \Gamma(a) = (\tau_n, \tau_s)}{\Gamma \vdash x = a[i] : \Gamma[x \leftarrow (\tau_n, H)]}$$



LOAD

$$\frac{\Gamma \vdash i : (L, L) \quad \Gamma(a) = (\tau_n, \tau_s)}{\boxed{\Sigma,} \Gamma \vdash x = a[i] : \boxed{\Sigma,} \Gamma[x \leftarrow (\tau_n, H)]}$$

CONST-LOAD

$$\frac{n \text{ is constant}}{\Sigma, \Gamma \vdash x = a[n] : \Sigma, \Gamma[x \leftarrow \Gamma(a)]}$$

Typing Rules: Seq and Assign

ASSIGN

$$\frac{\Gamma \vdash e : \tau}{\Sigma, \Gamma \vdash x = e : \Sigma, \Gamma[x \leftarrow \tau]}$$

SEQ

$$\frac{\Sigma_0, \Gamma_0 \vdash c_1 : \Sigma_1, \Gamma_1 \quad \Sigma_1, \Gamma_1 \vdash c_2 : \Sigma_2, \Gamma_2}{\Sigma_0, \Gamma_0 \vdash c_1; c_2 : \Sigma_2, \Gamma_2}$$

Typing Rules: Branching

$$\frac{\text{IF} \quad \Gamma \vdash b : (L, L) \quad \Sigma|_b, \Gamma \vdash c_1 : \Sigma_1, \Gamma_1 \quad \Sigma|_{!b}, \Gamma \vdash c_2 : \Sigma_2, \Gamma_2}{\Sigma, \Gamma \vdash \text{if } b \text{ then } c_1 \text{ else } c_2, \Sigma_1 \cap \Sigma_2, \Gamma_1 \cup \Gamma_2}$$

- $\Sigma|_b = \mathbf{ms}|_b$ if $\Sigma = \mathbf{ms}$, otherwise **unk**
- $\Sigma_1 \cap \Sigma_2 = \Sigma_1$ if $\Sigma_1 = \Sigma_2$, otherwise **unk**

Branching Example

```
{ ms }
```

```
if i < 10 {
```

```
  { ms| i<10 }
```

```
  set_ms(i < 10);
```

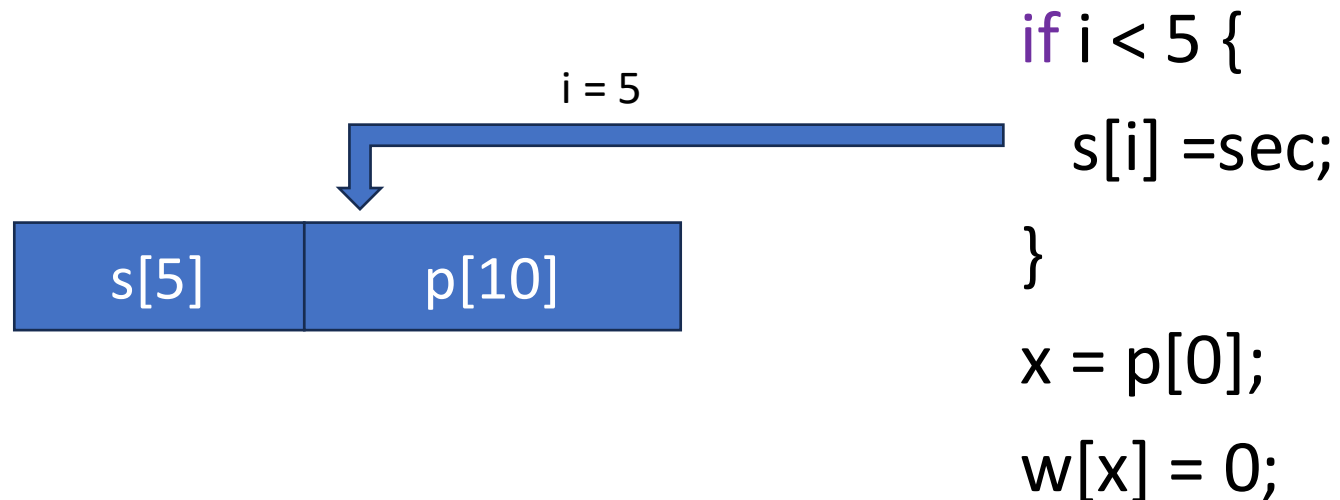
```
  { ms }
```

After set_ms, ms correctly models misspeculation

Can be safely used for speculative protection

Speculative Stores

- We can store a value with label τ in an array with label τ' if $\tau \leq \tau'$
- Implicit assumption: accesses are in bound
- Speculative executions break this assumption!



Typing Rules: Store

$$\frac{\text{STORE} \quad \Gamma \vdash i : (L, L) \quad \Gamma \vdash e : \tau \quad \tau \leq \Gamma(a) \quad \forall a' : \mathbf{A}, a' \neq a. \Gamma'[a'] = (\Gamma_n[a'], \tau_s \cup \Gamma_s[a'])}{\Sigma, \Gamma \vdash a[i] = e : \Sigma, \Gamma'}$$

Exercises

- Starting from **ms**, either provide a typing derivation or explain typing failures for the following programs. All variables but **s** and **sec** have type **L**, **L**

```
if i < 10 {  
  x = p[i];  
}  
w[x] = 0;
```

```
if i < 10 {  
  s[i] = 0;  
}  
x = p[0];  
w[x] = 0;
```

```
if i < 5 {  
  s[i] = sec;  
}  
x = p[0];  
w[x] = 0;
```

```
if i < 10 {  
  ms = set_ms(i < 10);  
  x = p[i];  
  x = protect(x, ms);  
}  
w[x] = 0;
```

```
if b {  
  ms = set_ms(b);  
  s[3] = sec;  
} else {  
  ms = set_ms(!b);  
}  
x = p[5];  
w[x] = 0;
```

```
b = i < 5;  
if b {  
  ms = set_ms(b);  
  s[i] = sec;  
} else {  
  ms = set_ms(!b);  
}  
x = p[0];  
x = protect(x, ms);  
w[x] = 0;
```

Typing Limitations

- Only guarantees resistance against timing, cache-based, and speculative (with extension) 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);
 $r := (x \& \text{mask}) \mid (y \& \sim \text{mask});$



: Did you mean

```
[[@ 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;
}
```

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

Avoiding Compiler-Induced Side-Channels

- Use a constant-time preserving compiler

Formal verification of a constant-time preserving C compiler, Barthe et al., POPL' 20

Preservation of Speculative Constant-Time by Compilation, Arranz Olmos et al., POPL' 25

- Impressive, but heavy effort needed
- How to reach performance of industrial compilers?
- How to scale to variety of backends and architectures?

Avoiding Compiler-Induced Side Channels

- 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

- How to determine which parts of memory/registers should be secret?
- How to precisely analyze binary code, and retrieve semantic structure?