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

\textit{secret s, key k}

\begin{align*}
m &\leftarrow \text{encrypt}(k, s) \\
\text{send } m
\end{align*}

Assumption: \textit{k is secret}

Implementation:

\begin{align*}
\text{print}(k) \\
\text{let } m = \text{encrypt}(k, s) \text{ in} \\
\text{send}(m)
\end{align*}
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 \(ed = 1 \mod (p - 1)(q - 1)\)
- 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 \((M, M^d \mod N)\)
  • 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):

\begin{align*}
x &= 1 \\
&\text{for } k = 0 \text{ to } w - 1 \text{ do} \\
&\quad \text{if } d[k] = 1 \text{ then } x = xC \mod N \\
&\quad x = x^2 \mod N \\
&\text{return } x
\end{align*}
Timing Attack on RSA

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

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

\[ x = x^2 \mod N \quad // = 1 \mod N \]

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

\[ x = x \cdot C \mod N \quad // = C \mod N \]

\[ x = x^2 \mod N \quad // = C^2 \mod N \]

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

\[ x = x^2 \mod N \quad // = C^4 \mod N \]

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

\[ x = x \cdot C \mod N \quad // = C^5 \mod N \]

\[ x = x^2 \mod N \quad // = C^{10} \mod N \]

\[ x = 1 \] for \( k = 0 \) to \( w - 1 \) do

\[ \text{if } d[k] = 1 \text{ then } x = x \cdot C \mod N \]

\[ x = x^2 \mod N \]

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

• 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 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.*
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

\[
\begin{align*}
    p_0 & \quad p_4 & \quad p_8 & \quad p_{12} \\
    p_1 & \quad p_5 & \quad p_9 & \quad p_{13} \\
    p_2 & \quad p_6 & \quad p_{10} & \quad p_{14} \\
    p_3 & \quad p_7 & \quad p_{11} & \quad p_{15}
\end{align*}
\]

\[
\begin{align*}
    c_0 & \quad c_4 & \quad c_8 & \quad c_{12} \\
    c_1 & \quad c_5 & \quad c_9 & \quad c_{13} \\
    c_2 & \quad c_6 & \quad c_{10} & \quad c_{14} \\
    c_3 & \quad c_7 & \quad c_{11} & \quad c_{15}
\end{align*}
\]

AES Rounds
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:

$x_0$ $x_4$ $x_8$ $x_{12}$
$x_1$ $x_5$ $x_9$ $x_{13}$
$x_2$ $x_6$ $x_{10}$ $x_{14}$
$x_3$ $x_7$ $x_{11}$ $x_{15}$

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

\[
x = *p; \\
... \\
y = *p;
\]
Cache Model (Simplified)

- $x = *p$;
- ...
- $y = *p$;

- 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 $p$,

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

$$s_1|_L = s_2|_L \Rightarrow \quad \text{// States agree on low values}$$

$$s_1 \xrightarrow{p}^* s_1' \Rightarrow \quad \text{// Executing } p \text{ in } s_1 \text{ yields } s_1'$$

$$s_2 \xrightarrow{p}^* s_2' \Rightarrow \quad \text{// Executing } p \text{ in } s_2 \text{ yields } s_2'$$

$$s_1'|_L = s_2'|_L \quad \text{// Results agree on low values}$$
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]

\[
\begin{align*}
\text{(expressions)} & \quad e ::= x | l | n | e + e' | e - e' | e = e' | e < e' \\
\text{(commands)} & \quad c ::= e ::= e' | c ; c' | \text{if } e \text{ then } c \text{ else } c' | \\
& \quad \text{while } e \text{ do } c | \text{letvar } x ::= e \text{ in } c
\end{align*}
\]

\[
\begin{align*}
\text{(data types)} & \quad \tau ::= s \\
\text{(phrase types)} & \quad \rho ::= \tau | \tau \text{ var} | \tau \text{ cmd}
\end{align*}
\]

- 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

**(INT)** \[ \lambda; \gamma \vdash n : \tau \]

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

**(VARLOC)** \[ \lambda; \gamma \vdash l : \tau \ var \quad \text{if } \lambda(l) = \tau \]

**(ARITH)** \[
\begin{align*}
\lambda; \gamma \vdash e & : \tau, \\
\lambda; \gamma \vdash e' & : \tau
\end{align*}
\]

\[ \lambda; \gamma \vdash e + e' : \tau \]

**(ASSIGN)** \[
\begin{align*}
\lambda; \gamma \vdash e & : \tau \ var, \\
\lambda; \gamma \vdash e' & : \tau
\end{align*}
\]

\[ \lambda; \gamma \vdash e := e' : \tau \ cmd \]
Typing Rules

(COMPPOSE)

\[
\begin{align*}
\lambda; \gamma \vdash c : \tau \text{ cmd}, \\
\lambda; \gamma \vdash c' : \tau \text{ cmd} \\
\hline
\lambda; \gamma \vdash c; c' : \tau \text{ cmd}
\end{align*}
\]

(IF)

\[
\begin{align*}
\lambda; \gamma \vdash e : \tau, \\
\lambda; \gamma \vdash c : \tau \text{ cmd}, \\
\lambda; \gamma \vdash c' : \tau \text{ cmd} \\
\hline
\lambda; \gamma \vdash \text{if } e \text{ then } c \text{ else } c' : \tau \text{ cmd}
\end{align*}
\]

(WHILE)

\[
\begin{align*}
\lambda; \gamma \vdash e : \tau, \\
\lambda; \gamma \vdash c : \tau \text{ cmd} \\
\hline
\lambda; \gamma \vdash \text{while } e \text{ do } c : \tau \text{ cmd}
\end{align*}
\]
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: \( x: \text{H var}, y: \text{H var} \vdash \text{if } x = 1 \text{ then } y := 1 \text{ else } y := 0 : \text{H cmd} \)

\[
\begin{align*}
\lambda; \gamma \vdash e : \tau, \\
\lambda; \gamma \vdash c : \tau \text{ cmd}, \\
\lambda; \gamma \vdash c' : \tau \text{ cmd}
\end{align*}
\]

\[
\begin{align*}
\lambda; \gamma \vdash \text{if } e \text{ then } c \text{ else } c' : \tau \text{ cmd}
\end{align*}
\]

Need to prove

- \( x: \text{H var}, y : \text{H var} \vdash x = 1 : \text{H} \)
- \( x: \text{H var}, y : \text{H var} \vdash y := 1 : \text{H cmd} \)
- \( x: \text{H var}, y : \text{H var} \vdash y := 0 : \text{H cmd} \)
Typing Example

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

\[
\begin{align*}
\lambda; \gamma \vdash e : \tau, \\
\lambda; \gamma \vdash e' : \tau
\end{align*}
\]

\[
\lambda; \gamma \vdash e + e' : \tau
\]

(ARITH)

Need to prove

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

\[
\lambda; \gamma \vdash n : \tau
\]

(INT)

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

\[
\lambda; \gamma \vdash x : \tau \text{ var}
\]

(VAR)

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

\[
\lambda; \gamma \vdash e : \tau \text{ var}
\]

(R-VAL)

\[
\lambda; \gamma \vdash e : \tau
\]
Typing Example

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

Need to prove

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

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

\[ \frac{\lambda; \gamma \vdash e : \tau \text{ var}, \quad \lambda; \gamma \vdash e' : \tau}{\lambda; \gamma \vdash e := e' : \tau \text{ cmd}} \]

\( (\text{ASSIGN}) \)

\( (\text{VAR}) \) \hspace{2cm} \lambda; \gamma \vdash x : \tau \text{ var} \quad \text{if } \gamma(x) = \tau \text{ var} \)

\( (\text{INT}) \) \hspace{2cm} \lambda; \gamma \vdash n : \tau \)
Label Subtyping

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

$$\lambda; \gamma \vdash e : \tau,$$
$$\lambda; \gamma \vdash c : \tau \text{ cmd},$$
$$\lambda; \gamma \vdash c' : \tau \text{ cmd}$$

\[ \text{(IF)} \]

$$\lambda; \gamma \vdash \text{if } e \text{ then } c \text{ else } c' : \tau \text{ cmd}$$

• The If rule requires the condition and the commands to have the same label!
Label Subtyping

\[
\begin{align*}
\text{(BASE)} & \quad \frac{\tau \leq \tau'}{\vdash \tau \subseteq \tau'} \\
\text{(SUBTYPE)} & \quad \frac{\lambda; \gamma \vdash p : \rho, \vdash \rho \subseteq \rho'}{\lambda; \gamma \vdash p : \rho'}
\end{align*}
\]

- 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

\[(\text{CMD}^-)\]

\[\vdash \tau \subseteq \tau' \quad \frac{}{\vdash \tau' \text{ cmd } \subseteq \tau \text{ cmd}}\]

• 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: \[y := 1 : H \text{ cmd } \Rightarrow y := 1 : L \text{ cmd}\]
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 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

\[
(\text{Trace}) \ l := \varepsilon \ | \ \text{Branch} (b) . \ l \ | \ \text{Access} (n) . \ l
\]

\[
s_1 \rightarrow_p^* s'_1, l_1
\]

When executing \( \text{if} \ b \ \text{then} \ p \ \text{else} \ p' \), we record \( \text{Branch} (b) \)

When executing \( a[n] \), we record \( \text{Access} (n) \)
Non-Interference with Observations

For a given program $p$, 

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

$$s_1|_L = s_2|_L \Rightarrow s_1 \rightarrow_p s_1', l_1 \Rightarrow s_2 \rightarrow_p s_2', l_2 \Rightarrow$$

$$s_1'|_L = s_2'|_L \land l_1 = l_2$$

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*
  

• We need tools to enforce this
Non-Interference by Typing Abstraction

• Remember from last week:

  ```
  Hash.fst
  let hash = sha2
  ```

  ```
  Hash.fsti
  val hash
  ```

  ```
  HKDF.fst
  let hkdf = .... Hash.hash ...
  ```

• Client modules only have access to the interface
• Underlying implementation is hidden (true for other languages supporting abstraction)
Non-Interference by Typing Abstraction

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

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

Unused assignment

Password can leak after execution!
Assume $b$ is secret

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

Rewrite into constant-time version

```c
int mask = create_mask(b);
r := (x & mask) | (y & ~mask);
```

: 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;

• 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];
Taint Analysis: Erasing Taint

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

\[
\text{let } f (x : \text{int}, p1: \ast\text{int}, p2: \ast\text{int}) = \\
\quad \ast p1 := x; \\
\quad y := \ast p2; \\
\quad \text{if } y > 0 \ldots
\]

• 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