Transforming out timing leaks

Tobias Stadler

26. Mai 2009

Inhaltsverzeichnis

1. Einführung
2. Semantic Security Condition
3. Transformationsalgorithmus

Ausgangssituation

- ein Benutzer lädt ein Programm aus dem Internet
- das Programm bekommt geheime und nicht geheime Daten als Eingabe
- das Programm kann über das Internet kommunizieren
- der Angreifer hat keinen Zugriff auf das System, dass das Programm ausführt
- der Angreifer hat Zugriff auf das Kommunikationsverhalten des Programms (Wann? und Was?)

Arten bzw. Wege des Informationsgewinns durch Angreifer

- **direct leakage**
  der Angreifer fängt die Daten in ihrer Rohform ab.
- **indirect leakage** (auch "leakage through a *covert storage channel*")
  der Angreifer kann die geheimen Daten aus dem Kommunikationsverhalten des Programms auslesen. *indirect leakage* kann durch *non-interference* behandelt werden).
- **timing leakage** (auch "leagage through a *covert timing channel*")
  dem Angreifer ist es möglich über das zeitliche Verhalten des Programms Rückschlüsse auf die privaten Daten zu ziehen.

Beispiel: Timing leakage bei RSA (1)

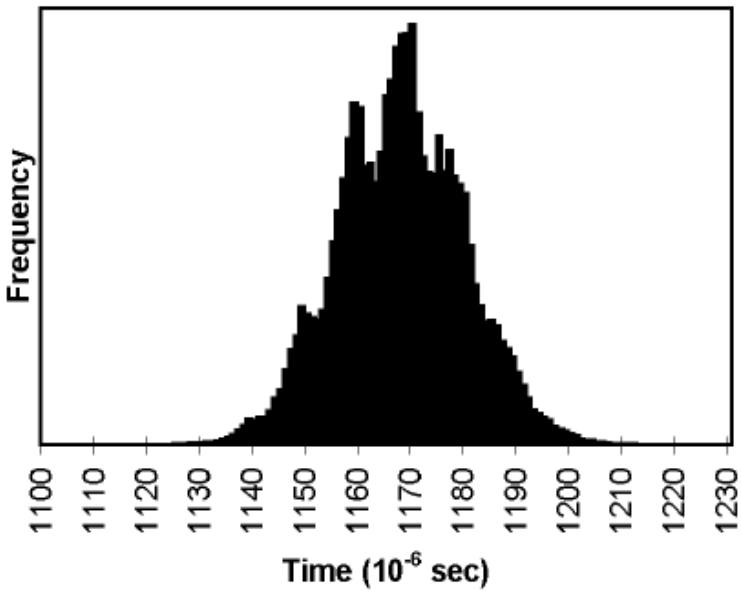
Zur Erinnerung

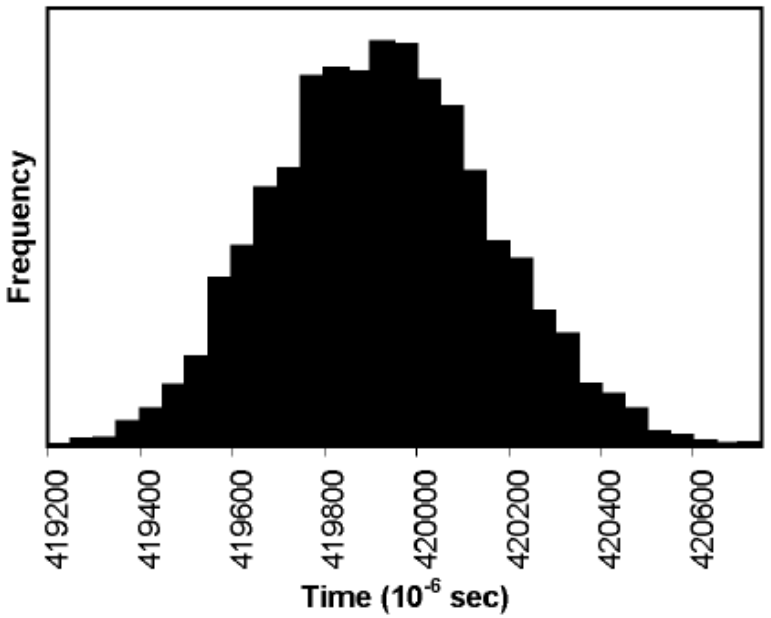
Verschlüsselung: $c = m^e \mod n$
Entschlüsselung: $m = c^d \mod n$

Modulare Exponentiation zur Berechnung von $r = y^x \mod n$

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

Beispiel: Timing leakage bei RSA (2)





Weitere Vorgehensweise

1. Definition der Syntax der Sprache
2. Definition der Semantik der Sprache
3. Definition eines Kriteriums, dass sicherstellt, dass in einem Programm keine *timing leaks* auftreten
4. Entwicklung eines Typensystems, indem alle Programme keine *timing leaks* aufweisen
5. Entwicklung des Transformationsalgorithmus

Definition der verwendeten Sprache (1)

Syntax

operators $op ::= + | * | - | = | != | < | <=$

expressions $e ::= l | e\ op\ e | le$

initialisers $ie ::= e | \text{mkarray}(e)\ ie | \{x_1 = ie_1, \ldots, x_n = ie_n\}$

commands $C, D ::= le := e | \text{skipAsn}\ le\ e | \text{if}(e)\ C\ \text{else}\ D |$
$\text{skipIf}\ e\ C | \text{let}\ x := ie\ \text{in}\ C | \text{while}(e)\ C | C; D | \text{output}\ x$

left-expressions $le ::= x | le.x | le[e]$

left-values $lv ::= x | lv.x | lv[n]$

values $v ::= l | \{x_1 = v_1, \ldots, x_n = v_n\}$

basic values $l ::= \text{n} | \text{true} | \text{false}$

Definition der verwendeten Sprache (2)

Semantik für Ausdrücke und Initialisierer

Die Semantik ist intuitiv definiert.

Semantik für Anweisungen

Die Semantik für Anweisungen wird noch durch deren zeitliches Verhalten erweitert.
Für die Anweisung skipAsn *le e* gilt, dass sie die selbe Laufzeit wie $le := e$ aufweist. Analog dazu skipIf *e C* mit if(e) C.

Γ-Sicherheit

Definition

$\sim_\Gamma$ ist die größte symmetrische Relation auf Anweisungen C_1, C_2, für die gilt: Wenn C_1 und C_2 die selben Umgebungen bzgl. der nicht privaten Daten erhalten, dann haben sie das selbe zeitliche Verhalten und haben auf die nicht privaten Daten die selbe Auswirkung

Γ-Sicherheit

Eine Anweisung C ist Γ-sicher, wenn $C \sim_\Gamma C$ gilt.

Typensystem (1)

Typen für Ausdrücke und Initialisierer

Die Basistypen der Sprache werden durch Sicherheitslevel L (für *low security*) und H (für *high security*) annotiert. Die Ableitung der Typen für Ausdrücke werden kanonisch definiert.

Definition

Der *low slice* C_L einer Anweisung C, ist syntaktisch identisch mit C, mit der Ausnahme, dass alle Zuweisungen an L-annotierte Daten durch die entsprechenden Skips ersetzt werden.

Typen für Anweisungen

Der Typ eine Anweisung C ist sein *low slice* C_L.

Typensystem (2)

Beispiele für Regeln zum Ableiten von Typen

$$\frac{\Gamma \vdash e : \overline{\tau}_L \quad \Gamma \vdash le : \overline{\tau}_L}{\Gamma \vdash le := e \; : \; le := e}$$

$$\frac{\Gamma \vdash e : \mathsf{Bool}_H \quad \Gamma \vdash C : C_L \quad \Gamma \vdash D : D_L}{\Gamma \vdash \mathsf{if}(e)\, C\, \mathsf{else}\, D \; : \; \mathsf{skipIf}\, e\; C_L} C \sim_\Gamma D_L$$

$$\frac{\Gamma \vdash e : \mathsf{Bool}_L \quad \Gamma \vdash C : C_L}{\Gamma \vdash \mathsf{while}(e)\, C \; : \; \mathsf{while}(e)\, C_L}$$

Transformationsregeln

Definition

Transformationsregeln sind von der Form $C \hookrightarrow D|D_L$.
Die Anweisung C wird durch die semantisch äquivalente Anweisung D ersetzt.

Beispiel für Transformationsregeln

$$\frac{\Gamma \vdash e : \overline{\tau}_L \quad \Gamma \vdash le : \overline{\tau}_L}{\Gamma \vdash le := e \hookrightarrow le := e | le := e}$$

$$\frac{\Gamma \vdash \mathsf{Bool}_H \quad \Gamma \vdash C_1 \hookrightarrow D_1|D_{1L} \quad \Gamma \vdash C_2 \hookrightarrow D_2|D_{2L} \quad ge(D_{1L})=\emptyset \quad ge(D_{2L})=\emptyset}{\Gamma \vdash \mathsf{if}(e)\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}(e)\ D_1;D_{2L}\ \mathsf{else}\ D_{1L};D_2|\mathsf{skipIf}\ e\ (D_{1L};D_{2L})}$$

Achtung

Der Algorithmus kann die Programme in Endlosschleifen schicken oder sie abnormal terminieren lassen.

Beispiel: Timing leakage bei RSA (3)

Transformation des mod. exp. Algorithmus

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

Viele Dank für die Aufmerksamkeit!