Transforming out Timing Leaks

Tobias Stadler

26.5.2009

1 Einführung

Der sichere Umgang mit privaten Daten wird bei zunehmender Vernetzung der Programme von immer größerer Bedeutung. Ein wichtiger Aspekt dabei ist, dass ein potenzieller Angreifer keine Informationen aus dem Kommunikationsverhalten des Programms auslesen kann. Man unterscheidet dabei drei verschiedene Arten, wie ein Angreifer an private Daten gelangen kann:

- **direct leakage**
- **indirect leakage**
- **timing leakage**

Bei *direct leakage* werden die Daten *"as is"* über den Informationskanal transportiert. Dieses Lecken kann verhindert werden, indem die Programme keine private Daten ausgeben dürfen.

Unter indirect leakage versteht man, dass aus dem beobachtbaren Verhalten des Programms Rückschlüsse auf die privaten Daten gezogen werden können. Ein Programm kann z.B. anhand der Werte der privaten Daten Datenbankzugriffe unterschiedlicher Art oder Häufigkeit machen. Um dies Art des Leckens zu verhindern ist der Begriff der **non-interference** nötig.

non-interference bedeutet, dass ein Programm auf die selbe Eingabe an nicht privaten Daten immer die selbe Ausgabe produziert.

Programme, die der *non-interference* Eigenschaft genügen, lecken keine privaten Daten durch ihr beobachtbares Verhalten.

timing leakage bedeutet, dass ein Angreifer anhand des zeitlichen Verhaltens des Programms die privaten Daten auslesen kann. Unter zeitlichem Verhalten versteht man hier die Information **Wann** und **Was** passiert ist. Beispielsweise sind viele Implementationen des RSA Algorithmus anfällig für *timing leaks*. Der Grund hierfür liegt allerdings nicht im Algorithmus selbst, sondern in der Berechnung von $r = y^x \mod n$, welche bei RSA

Abbildung 1: Der Modulare Exponentiations Algorithmus

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

zur Ver- und Entschlüsselung benötigt wird. Dafür wird anstatt der naiven Methode häufig der modulare exponentiations Algorithmus (siehe Abb. 1) verwendet Betrachtet man Zeile vier bis sechs genauer und vergleicht die beiden Zweigen der if-else-Anweisung bezüglich Laufzeit, so kann man zu dem Ergebnis kommen, dass die Verweildauer in dem if-Zweig größer als in dem else-Zweig ist. Kennt ein Angreifer, also die Zeit, die zwischen zwei Iterationen der while-Schleife vergangen ist, so kann er abschätzen, ob das $i$te Bit von x gesetzt ist oder nicht. Der Angreifer hat somit also auch ein Abschätzung für x, welches in dem RSA Beispiel den privaten Schlüssel darstellt. Die Wurzel allen Übels liegt in diesem Beispiel in der Verzweigung anhand der privaten Variable x. In [VoSm97] wird gezeigt, dass man durch den Verbot solcher Verzweigungen *timing leaks* verhindern kann. Da es aber Programme gibt, die auf solche Sprünge gar nicht oder nur sehr schwer verzichten können, wird in [Ag00] eine neue Methode zum Verhindern von *timing leaks* vorgestellt, die diese Art von Verzweigung zulässt. Die Idee dahinter ist, dass alle Zweige einer if-else Anweisung das gleiche beobachtbare und zeitliche Verhalten haben müssen. Im folgenden wird nun erläutert, wie das bewerkstelligt werden soll.

2 Semantic Security Condition

Zuerst ist es nötig ein Kriterium zu entwerfen, dass Programme beschreibt, die keine *timing leaks* besitzen. Damit dies formal gemacht werden kann, muss zuerst eine Sprache entworfen werden. Die Syntax dieser Sprache ist in Abb. 2 zu finden.

Die Semantik der Sprache ist so definiert, wie man es auch intuitiv machen würde. Allerdings wird die Semantik für die Anweisung noch um ihr zeitliches Verhalten (Laufzeit) erweitert. Zu Beachten ist auch noch die Anweisungen skipIf$(e)\,C$. Dabei handelt es sich um eine Anweisung, welche das selbe zeitliche Verhalten wie if$(e)\,C$ hat, aber selbst keine Auswirkung auf die Umgebung, also die Variablen und ihre Werten hat. Für skipAsn$\,le\,e$ gilt, dass sie das selbe zeitliche Verhalten wie $le := e$ hat, aber wiederum keine Auswirkung auf die Umgebung hat. Diese beide Anweisung werden später benötigt

Abbildung 2: Syntax der verwendeten Sprache

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

um die Zweige einer if-else Anweisung bzgl. ihrer Laufzeit anzupassen.

Nun, kann aufbauend auf dieser Sprache ein Kriterium formuliert werden, dass sicherstellt, dass ein Programm keine *timing leaks* aufweist. Dazu ist noch die Definition der $\sim_\Gamma$ Relation nötig.

$\sim_\Gamma$ ist die größte symmetrische Relation auf Anweisungen C_1, C_2, für die gilt: Wenn C_1 und C_2 die selben Umgebungen (Variablen und ihre Werte) bzgl. der nicht privaten Daten erhalten, dann haben sie das selbe zeitliche Verhalten und haben auf die jeweilige Umgebung (bzgl. der nicht privaten Daten) den selben Effekt. Die Umgebungen können sich auf den privaten Daten unterscheiden.

Vergleicht man nun eine Anweisung (bzgl. obiger Realation) mit sich selbst, so bekommt man eine Eigenschaft, die das zeitliche Verhalten auf allen möglichen Umgebungen (von privaten Daten) vergleicht. Zusätzlich wird auch noch die *non-interference* Eigenschaft betrachtet. Man definiert also:

C ist Γ-sicher, wenn $C \sim_\Gamma C$

Wenn also eine Aneweisung C Γ-sicher ist, so bedeutet das, dass sie auf allem möglichen Umgebungen mit gleichen nicht privaten Variablen das selbe zeitliche Verhalten aufweist und zudem den selben Effekt auf die nicht privaten Variablen hat.

Der letzte Schritt bevor ein Algorithmus entwickelt werden kann ist der Entwurf eines Typensystems, indem alle Programme Γ-sicher sind. Dazu werden die Basistypen in der Sprache noch mit einem Sicherheitslevel versehen, wobei L für *low security* (nicht private Daten) und H für *high security* (private Daten) steht. Die Ableitung der Typen für Ausdrücke und Initialisierer ist kanonisch definiert. Für die Ableitung der Typen bei Programmen ist noch der Begriff des *low slice* wichtig.

Der *low slice* C_L einer Anweisung C ist syntaktisch identisch mit C, mit der Ausnahmen, dass alle Zuweisungen an Variablen, die als *high security* deklariert worden sind, durch die entsprechenden Skips ersetzt wurden.

Abbildung 3: Beispiele für Regeln zum Ableiten von Typen

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

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

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

Abbildung 4: Beispiel für Transformationsregeln

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

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

Die Ableitung des Typs einer Anweisung ist dann definiert als sein *low slice*. Zu Beachten ist allerdings, dass das Typen System keine Zuweisung von *high security* Daten an *low security* Daten erlaubt (verhindert *direct leakage*), in while Schleifen nur über *low security* Daten iteriert werden darf und bei if-else Anweisungen muss das zeitliche Verhalten der beiden Zweige übereinstimmen (verhindert *indirect* und *timing leakage*). Alle Programme, die in dieses Schema fallen, sind dann Γ-sicher und lecken somit keine Daten, weder direkt noch durch ihr zeitliches oder beobachtbares Verhalten. In Abb. 3 sind Beispiele solcher Ableitungsregeln zu finden.

3 Transformationsalgorithmus

Transformationsregeln sind in der Form:

$\Gamma \vdash C \hookrightarrow D | D_L$

Eine Anweisung C wird überführt in ein semantisch äquivalente Anweisung D. Semantisch äquivalent bedeutet hier, dass sie auf die selbe Umgebung von (globalen) Variablen die selbe Sequenz von Ausgaben und Zuweisungen an *low security* Daten produzieren. Allerdings kann D evtl. mehr Berechnungen benötigen. Das Programm entsteht dann, indem alle Anweisungen C durch D ersetzt werden. In Abb. 4 sind Beispiel für solche Transformationsvorschriften zu sehen. Die Regeln sind kanonisch definiert, nur bei einer if Verzweigung auf *high security* Daten muss man aufpassen. Hier wird an den if-Zweig der *low slice* des else-Zweiges angehängt und dem else-Zweig wird der low slice des if-Zweiges vorangestellt. Dadurch erreicht man, dass die Ausführungszeit beider Zweige nach der Transformation gleich ist. Problematisch an der Transformation ist, dass sie das Terminationsverhalten des Programms ändern kann. Dies kann geschehen, da die Zweige einer if-else Anweisung das selbe zeitliche Verhalten haben. Befindet sich also

Abbildung 5: Transformation des Modularen Exponentiations 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;
}
```

z.B. in einem Zweig eine Endlosschleife, so wird diese als *low slice* in den andere Zweig kopiert und führt auch da zu einer Endlosschleife. Selbiges gilt natürlich auch für die abnormale Termination Die Transformation des modularen exponentiations Algorithmus aus Abb. 1 kann in Abb. 5 betrachtet werden. Das resultierende Programm gibt nun keine privaten Daten mehr über das zeitliche Verhalten Preis.

4 Ausblick

Obwohl Γ-Sicherheit ein gutes Kriterium ist um zu zeigen, dass ein Programm frei von *timing leaks* ist, so ist es trotzdem noch nicht präzise genug, sodass es eine Vielzahl von sicheren Programm ausschließt, welche keine *timing leaks* aufweisen. Des weiteren wurde bei der Entwicklung der Γ-Sicherheit und somit auch des Transformationsalgorithmus nicht die in modernen Rechner herrschende Speicherhierarchie betrachtet. Befinden sich Daten schon im CPU Cache oder müssen sie erst aus dem Arbeitsspeicher oder sogar der Festplatte geholt werden, so hat das empfindliche Auswirkung auf die Programmlaufzeit. Hier sind noch einige Hürden auf dem Weg zu umfassend sicheren Programmen zu nehmen.

5 Literatur

[Ko96] Paul C. Kocher: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. CRYPTO 1996
[Ag00] Johan Agat: Transforming out Timing Leaks. POPL 2000
[VoSm97] D.Volpano and G. Smith: Eliminating covert flows with minimum typings. 1997