Institut für Theoretische Informatik
Jun.-Prof. Dr. D. Hofheinz

Stammvorlesung Sicherheit im Sommersemester 2014

Übungsblatt 3

Hinweis: Übungsblätter können freiwillig bei Florian Böhl, Raum 255, Geb. 50.34 („Info-Bau“) bis zur Übung am 22.5.14 zur Korrektur abgegeben werden. Die Korrektur dient nur der Selbstkontrolle; es gibt keine Punkte und keinen Klausur-Bonus.

Aufgabe 1. Beurteilen Sie für die folgenden Konstruktionen jeweils, ob es sich für beliebige kollisionsresistente Hashfunktionen $H, H' : \{0,1\}^* \to \{0,1\}^k$ ebenfalls um kollisionsresistente Hashfunktionen handelt. Falls ja, beweisen Sie diese Aussage indem Sie zeigen, wie sich aus einer Kollision für $\hat{H}$ eine Kollision für H oder H' entwickeln ließe. Falls nicht, geben Sie konkrete Wahlen für H und H' und eine Kollision an.

(a) $\hat{H}(x) := H(x||x)$

(b) $\hat{H}(x) := H(x) \oplus H'(x)$

(c) $\hat{H}(x) := H(H'(x))$

(d) $\hat{H}(x) := F(x)||H(x)$ für eine beliebige Funktion $F : \{0,1\}^* \to \{0,1\}^k$

(e) $\hat{H}(x) := H(F(x))$ für eine beliebige Einwegfunktion $F : \{0,1\}^* \to \{0,1\}^k$

(f) $\hat{H}(x) := \begin{cases} x & \text{wenn } |x| = k \\ H(x) & \text{sonst} \end{cases}$

(g) $\hat{H}(x) := H(F(x))$ für eine beliebige injektive Funktion $F : \{0,1\}^* \to \{0,1\}^*$

Dabei bezeichne $||$ die Konkatenation von Bitstrings und $\oplus$ Verknüpfung durch bitweises XOR.

Lösungsvorschlag zu Aufgabe 1.

(a) $\hat{H}(x) := H(x||x)$ ist kollisionsresistent: Sei $\hat{H}(x) = \hat{H}(y)$ eine Kollision (also $x \neq y$) für $\hat{H}$, so ist $H(x||x) = H(y||y)$ eine Kollision für H (da $x \neq y \Rightarrow x||x \neq y||y$). Da H kollisionsresistent ist, dürfen auf Kollisionen für $\hat{H}$ nur schwer zu finden sein. Also ist $\hat{H}$ kollisionsresistent.

(b) $\hat{H}(x) := H(x) \oplus H'(x)$ ist im Allgemeinen nicht kollisionsresistent. Z.B. für $H = H'$ ist $\hat{H}(x) = 0^k$ für alle x.

(c) $\hat{H}(x) := H(H'(x))$ ist kollisionsresistent. Sei $\hat{H}(x) = \hat{H}(y)$ also $H(H'(x)) = H(H'(y))$. Ist $H'(x) \neq H'(y)$, so haben wir eine Kollision für H gefunden. Ansonsten ist $H'(x) = H'(y)$ eine Kollision für H' (da $x \neq y$).

(d) $\hat{H}(x) := F(x)||H(x)$ für eine beliebige Funktion $F : \{0,1\}^* \to \{0,1\}^k$ ist kollisionsresistent. Da $\hat{H}(x) = \hat{H}(y)$ insbesondere $H(x) = H(y)$ impliziert (die hinteren Teile der Ausgabe von $\hat{H}$ müssen übereinstimmen) liefert uns eine Kollision für $\hat{H}$ unmittelbar eine Kollision für H.

(e) $\hat{H}(x) := H(F(x))$ für eine beliebige Einwegfunktion $F : \{0,1\}^* \to \{0,1\}^k$ ist nicht kollisionsresistent. Sei beispielsweise F eine Einwegfunktion, für die sich leicht Kollisionen finden lassen. Jede Kollision $F(x) = F(y)$ ist offensichtlich eine Kollision für $\hat{H}$. Wir können ein solches F angeben: Sei

beispielsweise $\mathbb{G}$ eine Gruppe primer Ordnung mit Generator g, so dass sich die Elemente von $\mathbb{G}$ als Bitstrings der Länge k darstellen lassen (d.h., es ex. effizient berechenbare Funktion $\psi : \mathbb{G} \to \{0,1\}^k$). Sei außerdem $\varphi : \{0,1\}^* \to \mathbb{N}$ die kanonische Interpretation eines Bitstrings x als natürliche Zahl. Für hinreichend große Ordnung von $\mathbb{G}$ ist $F(x) := \psi(g^{\varphi(x)})$ eine Einwegfunktion (unter gängigen Annahmen). Wegen $\varphi(x) = \varphi(0^l||x)$ für bel. $l \in \mathbb{N}$ und $g^e = g^{e+|\mathbb{G}|}$ (wg. Satz von Lagrange) sind Kollisionen für F allerdings leicht zu finden.

(f) $\hat{H}(x) := \begin{cases} x & \text{wenn } |x| = k \\ H(x) & \text{sonst} \end{cases}$ ist nicht kollisionsresistent. Es ist $\hat{H}(y) = \hat{H}(H(y))$ für alle y mit $|y|! = k$. Für $H(y) \neq y$ (also quasi immer) liegt eine Kollision vor.

(g) $\hat{H}(x) := H(F(x))$ für eine beliebige injektive Funktion $F : \{0,1\}^* \to \{0,1\}^*$ ist kollisionsresistent. Sei $\hat{H}(x) = \hat{H}(y)$ (für $x \neq y$) und damit $H(F(x)) = H(F(y))$. Da F injektiv gilt $F(x) \neq F(y)$ und wir haben eine Kollision für H gefunden.

Aufgabe 2. Aus der Vorlesung ist bekannt, dass mithilfe einer kollisionsresistenten Kompressionsfunktion $\mathsf{F} : \{0,1\}^{2k} \to \{0,1\}^k$, mit $k \in \mathbb{N}$, eine kollisionsresistente Hashfunktion H_{MD} mittels des Merkle-Damgård-Verfahrens konstruiert werden kann. Eine mögliche Realisierung funkioniert wie folgt: Bei Eingabe von $M \in \{0,1\}^*$, mit $|M| = L < 2^k$, führe folgenden Algorithmus aus:

1. Setze $B := \lceil \frac{L}{k} \rceil$. (Dieses entspricht der Anzahl der Blöcke von M der Länge k.) Füge an das Ende der Nachricht M so viele Nullen an bis die Länge ein Vielfaches von k ergibt. Somit kann die Nachricht in die einzelnen k-Bit-Blöcke $M_1, \ldots, M_B$ aufgeteilt werden.
2. Setze $M_{B+1} := L$, wobei L mit exakt k Bits codiert wird.
3. Setze $Z_0 := 0^k$.
4. Für $i = 1, \ldots, B+1$ berechne $Z_i := \mathsf{F}(Z_{i-1}, M_i)$.
5. Gib Z_{B+1} aus.

Somit erhalten wir den Hashwert $\mathsf{H}_{\mathsf{MD}}(M) := Z_{B+1}$. Die Länge der Nachricht fließt also in den Hashwert ein. Sei nun $\mathsf{H}'_{\mathsf{MD}}$ eine Variante, die Z_B statt Z_{B+1} ausgibt. Geben Sie eine Kollision für $\mathsf{H}'_{\mathsf{MD}}$ an.

Betrachten wir nun nur Nachrichten, deren Länge ein Vielfaches der Blocklänge k ist. Lässt sich die Sicherheit von $\mathsf{H}'_{\mathsf{MD}}$ unter dieser Einschränkung für alle kollisionsresistenten Kompressionsfunktion F beweisen?

Lösungsvorschlag zu Aufgabe 2.

- **Kollisionen:** Fließt die Länge der Nachricht nicht in den Hashwert ein, lassen sich leicht Kollisionen finden, indem man einen Teil des Paddings der Nachricht zuordnet. Sei beispielsweise $M \in \{0,1\}^*$, mit $|M| \bmod k \neq 0$. Dann gilt $\mathsf{H}'_{\mathsf{MD}}(M) = \mathsf{H}'_{\mathsf{MD}}(M||0^\ell)$ für jedes $\ell \in \{0, \ldots, k - |M| \bmod k\}$. Für alle $\ell \neq 0$ ist dies eine Kollision.
- **Sicherheitsbeweis:** Nein, auch auf der Menge der Nachrichten, deren Länge ein Vielfaches der Blocklänge k ist, lassen sich Kollisionen finden. Um dieses Problem zu illustrieren, konstruieren wir uns zunächst eine spezielle Kompressionsfunktion $F_{IV,\overline{M}}$. Es sei $F : \{0,1\}^{2k} \to \{0,1\}^{k-1}$ eine kollisionsresistente Kompressionsfunktion und $IV, \overline{M} \in \{0,1\}^k$ beliebig. Wir setzen

  $$F_{IV,\overline{M}}(x) := \begin{cases} IV & \text{wenn } x = IV||\overline{M} \\ F(x)||b & \text{sonst, wobei } b := IV_k \oplus 1 \text{ und } IV_k \text{ bezeichne das letzte Bit von } IV \end{cases}$$

  Behauptung: $F_{IV,\overline{M}}$ ist kollisionsresistent.
  Beweis: Sei $F_{IV,\overline{M}}(x) = F_{IV,\overline{M}}(y)$ für $x \neq y$ eine Kollision für $F_{IV,\overline{M}}$. Ist $x \neq IV||\overline{M}$ und $y \neq IV||\overline{M}$ so haben wir eine Kollision für F gefunden; F ist aber kollisionsresistent. Also gilt o.B.d.A. $x = IV||\overline{M}$ und damit $F(y)||b = IV$. Auch das kann allerdings nicht sein, da $b \neq IV_k$. Somit ist $F_{IV,\overline{M}}$ kollisionsresistent. □

  Für die aus $F_{IV,\overline{M}}$ nach dem Merkle-Damgård-Verfahren konstruierte Hashfunktion H_{MD} gilt $\mathsf{H}_{\mathsf{MD}}(\overline{M}^a) = \mathsf{H}_{\mathsf{MD}}(\overline{M}^b) = IV$ für $a, b \in \mathbb{N}$. Sie ist also nicht kollisionsresistent.

Aufgabe 3. Im ersten Teil der RSA-Schlüsselgenerierung wurden für einen Benutzer A die Primzahlen $P = 23$ und $Q = 11$ gezogen.

(a) Setzen Sie die RSA-Schlüsselgenerierung fort. Berechnen Sie einen zu P und Q gehörigen öffentlichen RSA-Schlüssel $pk_A = (N, e)$ und einen privaten RSA-Schlüssel $sk_A = (N, d)$.
Hinweis: Führen Sie den erweiterten euklidischen Algorithmus zur Übung von Hand durch.

(b) Senden Sie die Nachricht $M = 17$ RSA-verschlüsselt an Benutzer A. Benutzen Sie dazu die Lehrbuch-Variante von RSA (ohne Padding) und den öffentlichen Schlüssel aus (a).

(c) Nehmen wir an, Sie seien Benutzer A. Entschlüsseln Sie das empfangene Chiffrat aus (b).

Lösungsvorschlag zu Aufgabe 3.

(a) $P = 23$ und $Q = 11$ führen zu $N = P \cdot Q = 253$ als RSA-Modulus. Während der Schlüsselgenerierung sind P und Q und damit insbesondere $\varphi(N) = (P-1)(Q-1) = 220$ (φ ist hier die eulersche φ-Funktion; $\varphi(N)$ die Ordnung der multiplikativen Gruppe $\mathbb{Z}_N = \mathbb{Z}/N\mathbb{Z}$). Wir führen die RSA-Schlüsselgenerierung aus der Vorlesung fort.

- Wir ziehen gleichverteilt $e \leftarrow \{3, \ldots, \varphi(N) - 1\}$ bis
- … bis $\mathsf{gcd}(e, \varphi(N)) = 1$ gilt. In unserem Fall wurde $e := 19$ gezogen.
- Wir berechnen $d = e^{-1} \bmod \varphi(N)$ mit erweitertem Euklid für $e = 19$ und $\varphi(N) = 220$:

  (i) $220 = 11 \cdot 19 + 11$
  (ii) $19 = 1 \cdot 11 + 8$
  (iii) $11 = 1 \cdot 8 + 3$
  (iv) $8 = 2 \cdot 3 + 2$
  (v) $3 = 1 \cdot 2 + 1$
  (vi) $2 = 2 \cdot 1$

  Nun zurück: $1 = 3 - 1 \cdot 2$ (wg. (v))
  $1 = 3 - 1 \cdot (8 - 2 \cdot 3) = -8 + 3 \cdot 3$ (wg. (iv), 2 ersetzen)
  $1 = -8 + 3 \cdot (11 - 1 \cdot 8) = 3 \cdot 11 - 4 \cdot 8$ (wg. (iii), 3 ersetzen)
  $1 = 3 \cdot 11 - 4 \cdot (19 - 1 \cdot 11) = -4 \cdot 19 + 7 \cdot 11$ (wg. (ii), 8 ersetzen)
  $1 = -4 \cdot 19 + 7 \cdot (220 - 11 \cdot 19) = 7 \cdot 220 - 81 \cdot 19$ (wg. (i), 11 ersetzen)
  Das inverse Element zu $e = 19$ im Exponenten (wo wir modulo $\varphi(N) = 220$ rechnen) ist also $d = -81 (\equiv 139 \bmod 220)$. Wir erhalten somit $pk_A := (N, e) = (253, 19)$ als öffentlichen Schlüssel und $sk_A := (N, d) = (253, 139)$ als geheimen Schlüssel für Benutzer A.
  Hinweis: Man kann die Koeffizienten 7 und -81 für die Gleichung $1 = 7 \cdot 220 - 81 \cdot 19$ auch direkt auf dem „Hinweg“ mit ausrechnen und spart sich so den „Rückweg“ (siehe z.B. Wikipedia). Wir fanden die Variante mit Rückweg hier allerdings pädagogisch wertvoller.

(b) Wir verschlüsseln die Nachricht $M = 17$ mittels des öffentlichen Schlüssels $pk_A = (N, e) = (253, 19)$ aus (a) zu $C_A := M^e \bmod N = 17^{19} \bmod 253 = 189$.

(c) Die Entschlüsselung mittels des geheimen Schlüssels $sk_A = (N, d) = (253, 139)$ von Benutzer A ergibt $M := C_A^d \bmod N = 189^{139} \bmod 253 = 17$.

Aufgabe 4. Auf der Webseite zur Vorlesung finden Sie die Datei `Sicherheit_UE03_Moduli.zip`, die 50.000 512-Bit RSA-Moduli enthält, von denen einige nicht vernünftig generiert wurden. Wie viele Moduli können Sie faktorisieren?

Hinweis: Mit Python können Sie die Liste beispielsweise wie folgt laden:

```
f = open('Sicherheit_UE03_Moduli.txt', 'r')
moduli = eval(f.read())
f.close()
```

Lösungsvorschlag zu Aufgabe 4. Diese Aufgabe wurde durch das Paper „Ron was wrong, Whit is right“[1] motiviert. Das Paper beleuchtet die Problematik, dass die kompliziertere Generierung von RSA-Schlüsseln (im Vergleich mit ElGamal beispielsweise) zu einem nicht kleinen Anteil an kaputten Schlüsseln führt. Entsprechend der Quote von $0,2\%$ aus dem Paper sind 100 der 50.000 auf unserer Webseite veröffentlichten Moduli bewusst schwach. Es gibt 45 Schlüsselpaare, die sich einen Primfaktor teilen und 10 Schlüssel mit einem (mehr oder weniger) kleinen Primfaktor.

Interessant ist vor allem, wie effizient sich gemeinsame Primfaktoren finden lassen. Unser erfolgreichster Mitarbeiter bei der Suche nach schwachen Moduli hat das Computer-Algebra-System Magma benutzt:

```
//  Suche nach kleinen Faktoren

load "~/Desktop/moduli.txt";
// "n_list" enthält die Liste der Moduli.

found := [];
time
for n in n_list do
   a,b :=Factorization(n: MPQSLimit := 0, ECMLimit := 7000);
   if a ne [] then
   Append(~found,n);
   print(a); print(n);
   end if;
   print(1);
end for;

// findet 6 Faktoren in ca. 2 Tagen CPU Zeit

// a,b :=Factorization(n: MPQSLimit := 0, ECMLimit := 17000);
// findet in ca. 15,5 Tagen CPU-Zeit nichts neues
// a,b :=Factorization(n: MPQSLimit := 0, ECMLimit := 25000);
// findet in ca. 35 Tagen CPU-Zeit (real 18 Tage auf schnellerem Rechner)
//       zwei weitere Faktoren (69 Bit und 91 Bit)

// ---------------------

// Suche nach gemeinsamen Teilern

load "~/Desktop/moduli.txt";
// "n_list" enthält die Liste der Moduli.

// Zerlegen der Liste der Moluli in Teil-Listen der Länge 340
// und erzeugen einer Liste der Produkte der Teil-Listen.

part_list := [];
list_of_lists := [];
list_products := [];

counter := 0;
for n in n_list  do
   counter +:= 1;
   Append(~part_list, n);
   if counter mod 400 eq 0
   then
      Append(~list_of_lists, part_list);
      Append(~list_products, &*part_list);
      part_list := [];
```

[1] https://eprint.iacr.org/2012/064.pdf

```
   end if;
end for;

Append(~list_of_lists, part_list);
Append(~list_products, &*part_list);

// In allen Teil-Listen die Elemente paarweise auf gemeinsame Teiler testen

faktoren := {};
time
for part_list in list_of_lists do
   for p1, p2 in part_list do
      if p1 ne p2 then Include(~faktoren, Gcd(p1,p2)); end if;
   end for;
end for;

// findet bei Listenlänge 400 eine gemeinsamen Teiler:
// 90802082609982494133656215088818572999954548239984721350779256518621811082279
// ****** Laufzeit 298 s  bzw. 156 s


// In der Liste der Produkte die Elemente paarweise auf gemeinsame Teiler testen

time
for p1, p2 in list_products do
       if p1 ne p2 then Include(~faktoren, Gcd(p1,p2)); end if;
end for;

// faktoren enthält jetzt alle paarweisen gemeinsamen Teiler der Moduli
// (oder auch Produkte davon)
// ****** Laufzeit  648 s   bzw. 439 s

// Alle gefundenen Faktoren (GGTs) multiplizieren zu "allfactors"
// Für alle Moduli den GGT mit "allfactors" berechnen und ggf. die Faktorisierung
// des Modulus

allfactors := &*faktoren;
found := [];

time
for n in n_list do
   f1 := Gcd (n,allfactors);
   if f1 ne 1
   then
      f2 := n div f1;
      if IsPrime(f2) then Append(~found,  [n,f1,f2]);
                          else Append(~found,  [n,f1,0]);
     end if;
  end if;
end for;

// ****** Laufzeit 12 s

// Gesamtzeit um 90 Faktoren zu finden:  16 Minuten  bzw. 10 Minuten


// jeweils nur zweier Listen machen (etwa 16 mal iterieren)
// Gesamtzeit für 90 Faktoren: 119 s  bzw. 77 s
```

Auch in Python lassen sich mit eigenen nicht besonders optimierten Implementierungen von GCD und Pollard-Rho schnell alle gemeinsamen Teiler und zumindest die zwei Moduli mit den kleinsten Primfaktoren in brauchbarer Zeit finden.