

# Security Lab – Public Key Cryptography und Hash Functions

## VMware

- Dieses Lab kann mit dem **Ubuntu-Image** bearbeitet werden, das Sie im Networking-Modus **NAT** starten sollten. In der Aufgabenstellung wird angenommen, dass Sie mit dem Ubuntu-Image arbeiten.
- Alternativ können Sie das Lab (komplett oder teilweise) auch auf Ihrem eigenen Laptop bearbeiten. Dazu benötigen Sie folgendes:
  - Für Teil 1 brauchen Sie das Programm **CrypTool 1**: Das Programm lässt sich direkt auf Ihrem Windows-Laptop installieren und mit Hilfe von Wine<sup>1</sup> auch unter Linux verwenden (das wird auch auf dem Ubuntu-Image so gemacht).
  - Für Teil 2 brauchen Sie Java und eine Entwicklungsumgebung.

## Einleitung

In diesem Praktikum werden Sie im ersten Teil mit dem Open-Source-Programm **CrypTool 1** arbeiten, ein Demonstrations- und Referenzprogramm für Kryptographie. Im zweiten Teil werden Sie einige Angriffe auf RSA kennenlernen und diese selbst durchführen und dabei auch etwas Java-Code schreiben. Wenn Sie CrypTool auf Ihrem eigenen Laptop installieren möchten, so laden Sie es von der CrypTool Website<sup>2</sup> herunter. Dieses Security Lab bezieht sich auf die **deutsche Version 1.4.40** und ist mit anderen Versionen (auch nicht mit der englischen Version) nicht zu 100% kompatibel, verwenden Sie deshalb auf jeden Fall die deutsche Version 1.4.40. CrypTool beherrscht eine Vielzahl von klassischen und modernen Verschlüsselungsalgorithmen und bietet eine Anzahl von Analysemethoden an, mit denen schwache kryptografische Algorithmen geknackt werden können. Zudem können Brute-Force-Attacken auf symmetrische Schlüssel durchgeführt werden und mit diversen Visualisierungen werden komplexere Zusammenhänge erklärt.

Im ersten Teil des Praktikums werden Sie sich durch einige der von CrypTool angebotenen Tutorials (in CrypTool als „Szenarien“ bezeichnet) durcharbeiten, um Ihr Verständnis für verschiedene kryptographische Verfahren zu vertiefen. Damit Sie möglichst viel von diesem Praktikum profitieren können, ist es wichtig, dass Sie sich jeweils genau überlegen, was passiert und wieso die demonstrierten Attacken möglich sind. Wenn Sie sich seriös durch die einzelnen Szenarien durcharbeiten, so wird das Erreichen der Praktikumpunkte für den ersten Teil kein Problem sein.

Im zweiten Teil wird detailliert auf Attacken gegen das RSA-Verfahren eingegangen. Insbesondere wird dabei auch auf praktikable Attacken eingegangen, die möglich werden, wenn das RSA-Verfahren falsch eingesetzt wird. Diese Attacken werden Sie zudem in Code umsetzen, um mit RSA verschlüsselten Ciphertext zu knacken. Auch hier sollte nach einem genauen Studium der Angriffe das Erreichen der Praktikumpunkte keine grosse Hürde darstellen.

*Hinweis:* Die von CrypTool angebotenen Szenarien sind nicht überall auf den Stand der Version 1.4.40 gebracht worden. Deshalb sind einzelne Angaben (z.B. Menüeinträge) teilweise nicht ganz korrekt oder die Szenarien enthalten Screenshots der Vorgängerversion. Dennoch sollten Sie die Szenarien problemlos durcharbeiten können; wo nötig sind klärende Hinweise direkt bei den Teilaufgaben angegeben. Bei einzelnen Ausgaben von CrypTool (insb. Histogramm und Autokorrelation bei den weiteren Aufgaben) wird zudem unter Wine die Schrift nicht sauber dargestellt. Sie sollten diese Diagramme aber zusammen mit dem Tutorial dennoch gut verstehen können.

---

<sup>1</sup> <http://www.winehq.org>

<sup>2</sup> <https://www.cryptool.org/de/cryptool1>

## Teil 1 – CrypTool-Aufgaben

Starten Sie ein Terminal und geben Sie folgendes ein:

- `cd /home/user/.wine/drive_c/Program\ Files\ \ (x86)/CrypTool`
- `wine CrypTool.exe`

Öffnen Sie dann in CrypTool den Menüpunkt *Hilfe* → *Szenarien (Tutorials)*, womit Sie für den ersten Teil des Praktikums bereit sind. Arbeiten Sie sich nun durch die in den folgenden Abschnitten dieser Praktikumsanleitung aufgeführten Szenarien durch und beantworten Sie nach jedem Szenario die in der Praktikumsanleitung zusätzlich aufgeführten Aufgaben (falls vorhanden).

### Diffie-Hellman-Demo

Diese Visualisierung zeigt, welche Schritte beim Diffie-Hellman Key-Exchange involviert sind. Wenn Sie das Verfahren nicht mehr genau im Kopf haben, so empfehlen wir Ihnen, zuerst die Vorlesungsunterlagen zu konsultieren oder die zu Beginn des Szenarios verlinkte Beschreibung (*Diffie-Hellmann Schlüsselaustauschverfahren*) durchzulesen.

**Aufgabe 1:** Der Diffie-Hellman Key-Exchange ist sicher gegenüber einem Angreifer, der als Man-in-the-Middle (MITM) die Nachrichten des Key-Exchange passiv mitlesen kann. Kann der MITM allerdings auch aktiv in den Key-Exchange eingreifen, so ist die Sicherheit nicht mehr gegeben. Beschreiben Sie kurz, wie eine solche aktive MITM-Attacke auf den Diffie-Hellman Key-Exchange funktioniert und mit welcher zusätzlichen Massnahme man sich dagegen schützen kann. Bei Bedarf finden Sie auch hierzu in den Vorlesungsunterlagen die notwendigen Informationen.

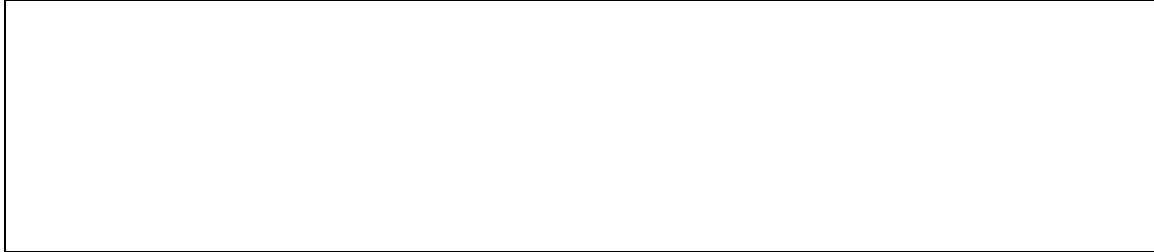
### RSA-Verfahren (Verschlüsselung)

Dieses Szenario erklärt, wie ein Plaintext mit dem Public Key des Empfängers verschlüsselt wird und wie der Ciphertext mit dem zugehörigen Private Key wieder entschlüsselt werden kann.

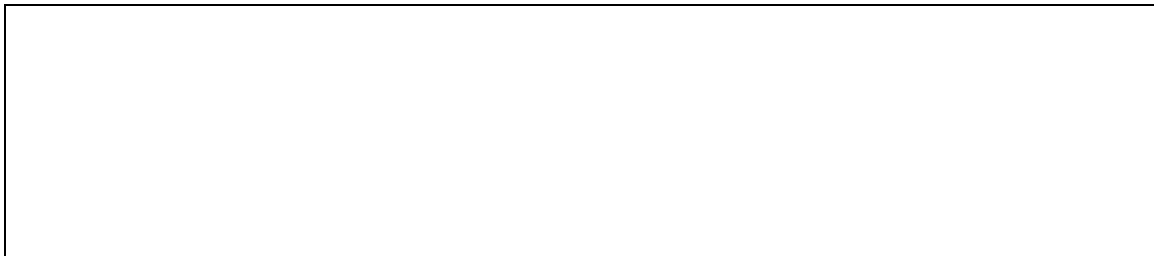
**Aufgabe 2:** Weshalb ist es sinnvoll, dass ein Passwort / Pincode nur beim Entschlüsseln eingegeben werden muss? Was wird hier durch das Passwort resp. den Pincode (vermutlich) geschützt?

**Aufgabe 3:** Mit einer einzelnen RSA-Operation kann immer nur eine begrenzte Datenmenge verschlüsselt werden. Verwendet man z.B. einen 512-Bit Schlüssel, so können damit maximal 64 Bytes Plaintext verschlüsselt werden, da der Plaintext (wenn er als Zahl interpretiert wird) numerisch kleiner sein muss als der Modulus. Müssen mehr als 64 Bytes verschlüsselt werden, so muss der Plaintext deshalb in mehrere Blöcke aufgeteilt werden, die dann jeweils mit einer separaten RSA-Operation verschlüsselt werden. Dies kann Sicherheitsprobleme mit sich bringen, was Sie im Folgenden kurz untersuchen werden. Laden Sie dazu von OLAT das File *beispieltext.txt* herunter und öffnen Sie die Datei

in CryptTool. Wie Sie sehen, beinhaltet die Datei fünfmal dasselbe, wobei jeder einzelne Teil aus 64 Bytes besteht. Verschlüsseln Sie die Datei nun mittels RSA, mit dem zuvor im Szenario verwendeten 512-Bit Schlüssel. Da die Datei mehr als 64 Bytes beinhaltet, unterteilt CryptTool den Inhalt in Blöcke der Länge 64 Bytes und verschlüsselt diese mit jeweils einer RSA-Operation. Betrachten Sie nun den Ciphertext genau. Was können Sie im Ciphertext beobachten und wieso ist dies so? Und was können Sie basierend auf dieser Beobachtung über den Plaintext aussagen, selbst wenn Sie dessen Inhalt nicht kennen würden?



**Aufgabe 4:** Nehmen Sie nun an, der Plaintext beinhaltet einen deutschen Text (ASCII-codiert) und dieser wird nun in Blöcke aufgeteilt, so dass jeder Block nur ein einzelnes Zeichen beinhaltet. Die einzelnen Blöcke werden nun mit RSA verschlüsselt, d.h. für jedes Zeichen wird eine separaten RSA-Operation verwendet. Ist es hier möglich, dass ein Angreifer basierend auf dem Ciphertext eine Frequenzanalyse durchführen kann, um den Plaintext herauszufinden? Begründen Sie Ihre Antwort.



Die zwei obigen Aufgaben sind zwar Extrembeispiele, aber sie zeigen Ihnen, dass es heikel ist, die einzelnen Blöcke „direkt mit der RSA-Verschlüsselungsoperation“ zu verschlüsseln, weil damit ein bestimmter Plaintext-Block immer auf den gleichen Ciphertext-Block abgebildet wird (solange der gleiche Public Key verwendet wird).

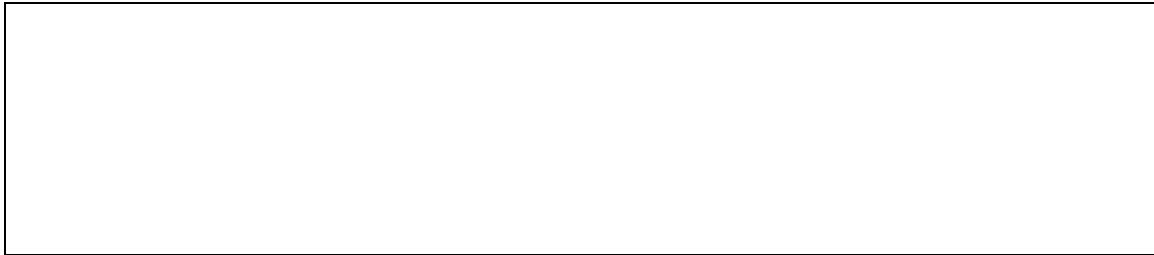
Dieses Problem kann verhindert werden, indem man einen originalen Plaintext-Block vor der Verschlüsselung mit zufälligen Bytes verknüpft, die für jeden einzelnen Plaintext-Block neu gewählt werden. Der bei dieser Verknüpfung resultierende Block wird anschliessend mit RSA verschlüsselt. Kommt ein bestimmter Plaintext-Block nun mehrere Male in einer Nachricht vor, so erhält man durch die Verwendung unterschiedlicher Zufallsbytes pro Block jedes Mal einen anderen Ciphertext-Block. Nach der Entschlüsselung können die zufälligen Bytes wieder entfernt werden und man erhält wieder den originalen Plaintext-Block. Genau dies wird mit den in der Vorlesung besprochenen standardisierten RSA-Verschlüsselungsformaten wie z.B. PKCS #1 erreicht. Wenn Sie PKCS #1 verwenden, dann sollten Sie aus Sicherheitsgründen die neueste Version 2.0 verwenden, da die Version 1.5 anfällig ist für sogenannte Padding-Oracle-Attacken. Version 2.0 verwendet mit dem Optimal Asymmetric Encryption Padding (OAEP) ein neues Padding-Verfahren, das die Probleme der früheren Version nicht mehr aufweist. Wenn Sie über das Internet eine sichere Verbindung mittels HTTPS aufbauen, kann es durchaus sein, dass bei Verwendung des RSA-Verfahrens noch PKCS #1 v1.5 eingesetzt wird. Dies weil das TLS-Protokoll, auf welchem HTTPS zum Erstellen einer sicheren Verbindung aufsetzt, OAEP erst ab TLS v1.3 verwendet (TLS 1.3 befindet sich noch im Standardisierungsprozess). Die aufgrund von PKCS #1 v1.5 möglichen gewordenen Angriffe wurden bei TLS-Versionen älter als v1.3 allerdings durch Workarounds gefixt, wodurch sie in der Realität keine Bedrohungen mehr darstellen. Bei Interesse finden Sie hier mehr Informationen dazu: <https://cryptosense.com/why-pkcs1v1-5-encryption-should-be-put-out-of-our-misery/>

**Aufgabe 5:** Diese Aufgabe hat keinen direkten Bezug zum oben durchgespielten Szenario und für die Beantwortung müssen Sie evtl. die Vorlesungsunterlagen zur Hand nehmen. Erklären Sie in eigenen Worten, wieso das effiziente Faktorisieren von grossen Zahlen für die Sicherheit von RSA verheerend wäre. Überlegen Sie zudem, was an der nachfolgenden Aussage von Bill Gates et al. falsch ist:

*The obvious mathematical breakthrough would be the development of an easy way to factor large prime numbers.*

– Bill Gates et al.<sup>3</sup>, 1995

Bill Gates hatte dabei wohl kaum an folgendes gedacht  $13 = (2 + 3i)(2 - 3i) \dots \text{☺}$



### Hybridverschlüsselung

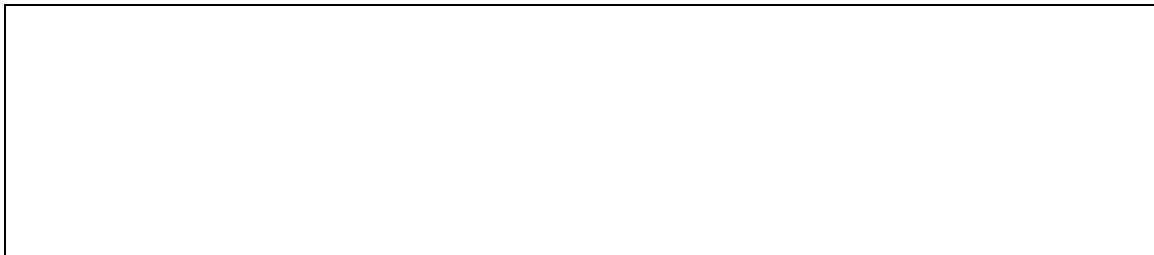
Public-Key-Operationen sind rechenaufwendig. Diese Visualisierung zeigt, wie Public-Key Verfahren mit effizienter symmetrischer Verschlüsselung gekoppelt werden können. Führen Sie zuerst eine Hybrid-Verschlüsselung und anschliessend eine entsprechende Entschlüsselung durch. Verwenden Sie dazu RSA-AES-Verschlüsselung bzw. Entschlüsselung. Als RSA-Schlüssel können Sie wiederum denselben 512-Bit Schlüssel wie oben verwenden.

Eine Aufgabe dazu finden Sie am Ende des Praktikums.

### RSA-Verfahren (Signatur)

Dieses Szenario erklärt, wie auf der Basis des eigenen Private Keys eine digitale Unterschrift (wir verwenden dazu alternativ auch den Begriff *Signatur*) auf einem Dokument erzeugt werden kann. Mit dem zugehörigen Public Key kann die Signatur von jedem verifiziert werden. Wurde das Originaldokument in irgendeiner Art manipuliert, so wird die Verifikation der Signatur fehlschlagen.

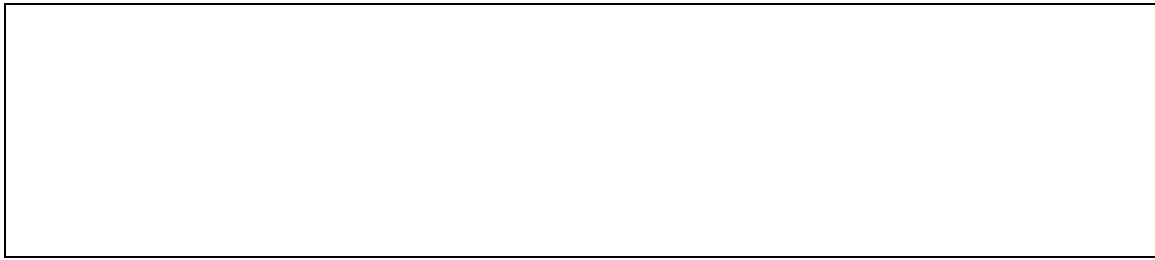
**Aufgabe 6:** Werden Signaturen zur Sicherung resp. Prüfung der Authentizität, der Integrität oder der Vertraulichkeit einer Nachricht eingesetzt?



**Aufgabe 7:** Was wird genau genommen signiert? Und welcher Teil des Schlüssels wird nochmals für das Erzeugen bzw. die Verifikation der Signatur verwendet? Wir stellen diese scheinbar triviale Frage, weil die Verwendung der Schlüssel immer wieder mal gerne verwechselt wird (z.B. in der Prüfung)...

---

<sup>3</sup> Bill Gates, Nathan Myhrvold and Peter M. Rinearson, The Road Ahead, Viking Press, 1995, p. 265



Im eben bearbeiteten Szenario wurde auch von Zertifikaten gesprochen, dazu liefert der nachfolgende Abschnitt einige Erklärungen.

Wenn eine digitale Unterschrift mit einer Identität in Verbindung gebracht werden soll, dann muss man den Public Key dieser Identität auf authentischem Weg erhalten. Ansonsten kann man zwar die Unterschrift mit dem Public Key überprüfen, man kann aber keine Aussage machen, wer die Unterschrift effektiv erzeugt hat, da man eben nicht sicher sein kann, wem der Public Key gehört. Da eine persönliche Übergabe des Public Keys durch den Besitzer der Identität nur in den wenigsten Fällen praktikabel ist, braucht es Lösungen, die ohne diesen direkten Kontakt auskommen. Hier kommen die im Szenario erwähnten digitalen Zertifikate ins Spiel (diese werden in der Vorlesung noch ausführlich besprochen oder wurden evtl. bereits besprochen). Ein digitales Zertifikat (auch Zertifikat oder Public-Key-Zertifikat genannt) enthält strukturierte Daten, die den Eigentümer sowie weitere Eigenschaften eines Public Keys bestätigen. Anhand eines solchen Zertifikats können dann andere Benutzer den Public Key im Zertifikat eindeutig einer Identität (z.B. einer Person, einer Organisation oder einem IT-System) zuordnen und seinen Geltungsbereich bestimmen. Dass dies geht basiert grundsätzlich darauf, dass Zertifikate von sogenannten Certification-Authorities (CA) ausgestellt (und dabei mit dem Private Key der CA signiert) werden, denen wir vertrauen und deren Public Keys wir mal authentisch, z.B. mit dem Betriebssystem oder zusammen mit dem Browser, erhalten haben. Die von einer CA ausgestellten Zertifikate, und somit die Zugehörigkeit eines bestimmten Public Keys zu einer Identität, können dann mit Hilfe des Public Keys der CA geprüft werden.

**Aufgabe 8:** Untersuchen Sie nochmals das im Szenario erzeugte Zertifikat mittels *Digitale Signaturen/PKI* → *Schlüssel anzeigen/exportieren*. Wenn Sie das Zertifikat genau betrachten, dann werden Sie sehen, dass es eine Signatur enthält. Welche im Zertifikat enthaltene Information sollte diese Signatur mindestens signieren (und damit bestätigen), damit das Zertifikat „seinen wichtigsten Zweck“ erfüllt? Und von wem stammt diese Signatur?



## Digitale Signatur

In dieser Visualisierung wird – im Sinne einer Repetition des oben Gelernten – grafisch gezeigt, welche Einzelschritte benötigt werden, um eine digitale Signatur zu erzeugen. Dabei kommen auch die Zertifikate explizit ins Spiel, die oben erklärt wurden.

*Hinweis:* Sie müssen eine Datei geöffnet haben, damit der entsprechende Menüpunkt *Digitale Signaturen/PKI* → *Signaturdemo* ausgewählt werden kann. In der nachfolgenden Visualisierung können Sie das Dokument aber problemlos austauschen.

Aus dieser Signaturdemo geht unter Umständen nicht klar hervor, dass das Zertifikat selbst für die Signatur nicht notwendig ist, denn der Private Key befindet sich nicht im Zertifikat, sondern nur der Public Key. Die eigentliche Signatur wird in der Visualisierung im „zweiten vertikalen Pfad von rechts“ ausgeführt: Der Private Key (hier mit *RSA-Schlüssel* bezeichnet) wird verwendet, um den

Hashwert über dem Dokument zu signieren (hier mit *Hashwert verschlüsseln* bezeichnet), woraus die Signatur (hier mit *verschlüsselter Hashwert* bezeichnet) resultiert. Damit ist der eigentliche Signaturvorgang beendet. Im vertikalen Pfad ganz rechts geht es nur noch darum, das Dokument, die Signatur und das Zertifikat zu kombinieren, denn diese drei Teile sind notwendig, wenn man die Signatur wieder verifizieren möchte. Für die Kombination dieser drei Teile gibt es verschiedene Standards, CrypTool verwendet hierzu aber ein eigenes Format.

### Angriff auf den Hashwert der digitalen Signatur

Wie Sie oben gesehen haben, wird eine Signatur üblicherweise nicht über die Nachricht selbst, sondern über den Hash dieser Nachricht erzeugt. Hat ein potentiell Opfer also eine Nachricht A signiert und schafft es ein Angreifer, eine zweite Nachricht B mit dem gleichen Hashwert wie der von Nachricht A zu erzeugen, so ist die Signatur auch auf der Nachricht B gültig. Im Idealfall kann der Angreifer beide Nachrichten A und B wählen und das Opfer die harmlose Nachricht A signieren lassen. Diese Attacke ist viel einfacher als wenn der Angreifer für eine gegebene Nachricht A eine weitere Nachricht B mit dem gleichen Hashwert suchen müsste. Der Grund liegt am „Birthday-Paradox“ und wird/wurde in der Vorlesung erklärt. Dieses Szenario geht auf genau diese Attacke ein.

Wählen Sie bei den Optionen die Hashfunktion MD5. Unten bei „Optionen für die Nachrichtenmodifikation“ sollten Sie „Zeichen anhängen“ und „Druckbare Zeichen“ wählen, damit Sie sehen, dass den Nachrichten nur ein paar zusätzliche Zeichen angehängt werden müssen, damit sie den gleichen Hashwert haben.

Natürlich können Sie mit Ihrer verfügbaren Rechenpower nicht wirklich die zwei Nachrichten so modifizieren, dass alle 128 Bits der Hashes identisch sind. Sie können bei „Signifikante Bitlänge“ aber angeben, wieviele Bits der beiden Hashes identisch sein sollen. Wenn Sie hier 32 angeben, so werden die resultierenden Hashwerte in den ersten 32 Bits identisch sein.

Wenn Sie 48 Bits wählen, dann werden Sie in vernünftiger Zeit (rund eine Minute) zu einem Ergebnis kommen. Um die Hashwerte der beiden generierten Nachrichten zu verifizieren, können Sie unter *Einzelverfahren* → *Hashverfahren* → *MD5* prüfen, ob die ersten 48 Bits wirklich identisch sind.

Was können Sie daraus lernen? Es zeigt Ihnen, wie enorm effizient diese Attacke (wegen dem Birthday-Paradox) im Vergleich zu einer „normalen“ Brute-Force-Attacke ist. Im ersten Praktikum haben Sie mit CrypTool eine Brute-Force-Attacke auf die ersten 24 Bits eines AES-Schlüssels durchgeführt, auch das dauerte „rund eine Minute“. In der gleichen Zeit haben Sie nun 48 Bits (also doppelt so viele) eines Hashwerts „geknackt“, Sie waren also  $2^{24}$  mal effizienter! Diese Attacke auf den kompletten 128-Bit Hashwert ist also etwa so komplex wie das Knacken eines 64-Bit langen symmetrischen Schlüssels und 64 Bit lange symmetrische Schlüssel gelten heute eindeutig als zu kurz. Entsprechend sollten wir also möglichst bald von den 128 Bit langen Hashwerten wegkommen, denn sie sind definitiv zu kurz.

Noch eine Schlussbemerkung: in der Vorlesung werden auch kurz Angriffe auf MD5 und SHA-1 diskutiert, die noch effizienter als die hier verwendete Brute-Force-Attacke sind, was das Problem natürlich weiter verschärft.

### Hybride Verschlüsselung selbst gemacht

**Aufgabe 9:** Laden Sie folgende Files von OLAT herunter:

- *completed.txt*
- *SecurityLabCrypTool.p12*

*SecurityLabCrypTool.p12* ist eine Datei im PKCS #12-Format, welche (weil CrypTool kein reines Public-Key-Dateiformat kennt) nebst dem Public Key auch den Private Key enthält und deshalb mit der PIN 73512 geschützt ist. Importieren Sie diese Datei in Ihren Zertifikatsspeicher über den Menüpunkt *Digitale Signaturen/PKI* → *PKI* → *Schlüssel erzeugen/importieren* und den Button *PKCS #12 Import*.

**Achtung:** Sollten Sie sich bei der Eingabe der PIN vertippen, kann es vorkommen, dass Sie anschliessend das Zertifikat überhaupt nicht mehr importieren können. Falls dies Eintritt, so können Sie das Problem mit dem am Ende dieser Praktikumsanleitung beschriebenen Workaround beheben.

Generieren Sie nun mit dem CrypTool ein Schlüsselpaar bzw. ein Zertifikat, das auf Ihren Namen lautet (Verwenden Sie den Namen eines der Gruppenmitglieder), das als Schlüsselkennung den zugehörigen ZHAW-Benutzernamen enthält und das einen 2048-Bit RSA-Schlüssel verwendet. Signieren Sie damit den Plaintext *completed.txt*, wobei Sie SHA-1 als Hashfunktion verwenden sollten. Auf diesen signierten Text wenden Sie anschliessend eine Hybrid-Verschlüsselung mit dem heruntergeladenen Public Key aus der Datei *SecurityLabCrypTool.p12* an. Speichern Sie den resultierenden hybrid-verschlüsselten (und signierten) Ciphertext, denn Sie brauchen diesen, um einen der Praktikumpunkte zu erhalten.

Sie können gleich selbst überprüfen, ob sich das Ganze auch wieder umkehren lässt (d.h. zuerst entschlüsseln und dann die Signatur verifizieren) um sicherzustellen, dass Sie den Punkt auch erhalten werden.

## Teil 2 – Angriffe auf RSA

Nachdem Sie sich nun bereits mit der Funktionsweise von RSA auseinandergesetzt haben betrachten wir einige Angriffe auf RSA. Dabei geht es insbesondere auch um drei praktikable Angriffe, die möglich werden, wenn RSA falsch eingesetzt wird. Zwei dieser Angriffe basieren auf einer naiven Umsetzung von RSA. Der dritte Angriff basiert auf einem Problem, das auftritt, wenn der Modulus (wir verwenden hier den englischen Ausdruck, auf Deutsch wäre dies *der Modul*) verschiedener Kommunikationspartner gemeinsame Faktoren aufweist. Anschliessend werden Sie diese Attacken in Code umsetzen, um mit RSA verschlüsselte Nachrichten zu knacken.

Diese Attacken sollen Ihnen vor allem aufzeigen, dass eine sichere Umsetzung des RSA-Verfahrens mehr beinhaltet als einfach «nur die direkte Implementation der entsprechenden RSA-Formeln» zur Ver- und Entschlüsselung (dies haben Sie ja auch bereits bei der Diskussion nach Aufgabe 4 weiter oben gesehen). Ebenfalls werden Sie dabei Ihr Verständnis für RSA weiter vertiefen.

Bei der Beschreibung der Attacken werden Sie sich durch einiges an Mathematik durcharbeiten. Wenn Sie diese verstanden haben, dann sollte es kein grösseres Problem sein, die Attacken in Code umzusetzen.

### Grundsätzliches

Um eine Nachricht  $m$  mit RSA zu verschlüsseln und als Ciphertext  $c$  an Bob zu schicken, verwendet Alice  $m$  und den öffentlichen Schlüssel  $(e, N)$  von Bob wie folgt:

$$(1) \quad c = m^e \bmod N$$

Da das RSA-Verfahren verlangt, dass  $m$  eine Ganzzahl ist für die zudem  $0 \leq m < N$  gilt, müssen Nachrichten, die dieser Forderung nicht entsprechen, in eine solche Ganzzahl abgebildet werden. Das macht man üblicherweise so, dass die Bitsequenz der Nachricht einfach als Ganzzahl interpretiert wird. Betrachten wir als Beispiel die ASCII-codierte Nachricht „hallo“, die mit RSA verschlüsselt werden soll. In diesem Fall entspricht die Bitsequenz der Nachricht der ASCII-Codierung der einzelnen Zeichen: 0x68 0x61 0x6c 0x6c 0x66 (hexadezimal dargestellt). Diese Bitsequenz kann man nun auch als die positive Ganzzahl 68616c6c66 (zur Basis 16) oder 448'311'094'383 (zur Basis 10) verstehen. RSA verschlüsselt im Fall der ASCII-codierten Nachricht „hallo“ also die Ganzzahl 448'311'094'383.

Nehmen wir nun an, dass  $L(N)$  die Länge von  $N$  in Bits ist, d.h. die Anzahl Bits in der Binärdarstellung des Modulus (dies entspricht bei RSA der Schlüssellänge). Wegen der Bedingung  $0 \leq m < N$  kann  $m$  damit maximal  $L(N) - 1$  Bits lang sein, zumindest dann, wenn alle Bitkombinationen dieser

Länge möglich sein sollen<sup>4</sup>. Im Folgenden nehmen wir der Einfachheit halber an, dass die Nachricht immer maximal  $L(N) - 1$  Bits lang ist, sie kann also mit einer einzelnen RSA-Operation verschlüsselt werden.

Nehmen wir weiter an, dass Eve den Ciphertext abgefangen hat, womit sie  $c$  kennt. Zudem müssen wir davon ausgehen, dass Eve auch den Public Key  $(e, N)$  von Bob kennt, da dieser ja öffentlich ist. Um  $c$  zu entschlüsseln benötigt Eve den zugehörigen Private Key  $(d, N)$  von Bob. Wie Sie aus der Vorlesung wissen, kann der Private Key durch Faktorisieren des Modulus  $N$  ermittelt werden. Da  $N$  aus zwei verschiedenen Primfaktoren besteht, muss einer der Faktoren kleiner als  $\sqrt{N}$  sein. Ein möglicher Brute-Force-Angriff, um den Modulus zu faktorisieren, funktioniert deshalb so, dass man  $N$  durch alle Zahlen von 2 bis  $\lfloor \sqrt{N} \rfloor$  teilt. Ist das Resultat ganzzahlig, so hat man den ersten Primfaktor gefunden und das Ergebnis der Division entspricht dann natürlich gleich dem zweiten Primfaktor. Das Problem dieses Ansatzes ist, dass er enorm ineffizient ist und bis zu  $\lfloor \sqrt{N} \rfloor$  Zahlen durchprobiert werden müssen. Bei einem Modulus der Länge 2048 Bits sind dies rund  $2^{1024}$  Versuche. Der Aufwand wäre also vergleichbar mit dem Knacken eines 1024-Bit langen Secret-Key und man kann deshalb auch sagen, dass der Modulus bezüglich dieser Attacke eine *Secret-Key-Bitstärkenäquivalenz* von 1024 Bits aufweist.

Es gibt aber deutlich effizientere Faktorisierungsalgorithmen als dieser naive Ansatz. Der aktuell effizienteste Algorithmus zum Faktorisieren von ganzen Zahlen ist der General-Number-Field-Sieve-Algorithmus. Gemäss NIST<sup>5</sup> hat bei Verwendung dieses Algorithmus ein Modulus von  $L$  Bits noch folgende Secret-Key-Bitstärkenäquivalenz:

$$bs = \frac{1,923 \sqrt[3]{L \log_{10} 2} \sqrt[3]{\log_{10}(L \log_{10} 2)^2} - 4.69}{\log_{10} 2}$$

Bei  $L = 2048$  Bits beträgt diese folglich rund 112 Bits<sup>6</sup> und bei  $L = 1024$  Bits rund 80 Bits. Während eine Bitstärke von 80 Bits unter Verwendung konventioneller Computertechnik (kein Quantencomputer) heute in Reichweite von sehr finanzstarken Angreifern ist, gilt eine Bitstärke von 112 Bits gemäss NIST noch schätzungsweise bis zum Jahr 2030 als sicher. Da aber die Methoden zur Faktorisierung nur besser werden können, wird die Bitstärkenäquivalenz eines RSA-Modulus mit der Zeit tendenziell sinken. Die oben angegebene Formel wird sich also mit der Zeit verändern und ist sicher nicht für die Ewigkeit.

Diese Betrachtung hat Ihnen sicher auch nochmals deutlich aufgezeigt, weshalb man bei RSA so grosse Schlüssellängen verwenden muss (als Minimum sollte man heute 2048 Bits verwenden): Dies ist notwendig, damit die Faktorisierung des Modulus auch bei Verwendung der heute effizientesten Verfahren genügend aufwändig und RSA damit genügend sicher ist.

Der soeben beschriebene Angriff auf RSA mit dem General-Number-Field-Sieve-Algorithmus ist allerdings nur dann erforderlich, wenn RSA „umsichtig“ implementiert wird. Werden bei der Implementierung von RSA Fehler gemacht, dann gibt es viel einfachere Attacken. Darauf wird im Folgenden genauer eingegangen.

<sup>4</sup> Beispiel: Der Modulus  $N = 221$  entspricht in der Binärdarstellung 11011101 und hat damit eine Länge von 8 Bits. Damit können grundsätzlich alle „Nachrichten“ von 0 bis 220 verschlüsselt werden. Nachrichten von 128 bis 220 sind dabei die 8-Bit Nachrichten, die verschlüsselt werden können. Die Nachrichten 221-255 sind auch 8-Bit Nachrichten, sie können aber nicht verschlüsselt werden, da sie nicht kleiner als der Modulus sind. Die maximale Nachrichtenlänge, die man bei diesem Modulus in jedem Fall verschlüsseln kann, ist deshalb 7 Bits und damit 1 Bit kleiner als die Länge des Modulus.

<sup>5</sup> <http://csrc.nist.gov/groups/STM/cmvp/documents/fips140-2/FIPS1402IG.pdf#page=97>

<sup>6</sup> Gemäss Formel eigentlich rund 110. Für diesen Modulus hat NIST die Näherung noch präzisiert.



## Angriff auf Nachrichten geringer Länge

Wenn RSA nach dem Lehrbuch implementiert wird, so kann man z.B. mit einem Modulus von 1024 Bits alle Nachrichten  $m$  der Längen 1 bis 1023 Bits verschlüsseln. 1024 Bits geht nicht (in jedem Fall), weil ja  $m < N$  gelten muss (die Nachricht muss „numerisch“ kleiner sein als der Modulus).

Betrachten wir nun mal den Fall, dass wir Nachrichten  $m$  verschlüsseln, für die  $m^e < N$  gilt. In diesem Fall können wir (1) vereinfachen, denn auf Zahlen, die kleiner sind als der Modulus, hat der Modulo Operator keine Wirkung und kann deshalb weggelassen werden:

$$(2) \quad c = m^e \bmod N = m^e$$

Tritt dieser Fall ein, dann kann ein Angreifer einfach die  $e$ -te Wurzel von  $c$  ziehen und den Ciphertext auf diese Weise entschlüsseln, denn Wurzelziehen ist nur dann schwierig, wenn wir «Modulo eine grosse Zahl» rechnen müssen. RSA lässt sich in diesem Fall also ganz einfach knacken.

**Aufgabe 10:** Nehmen Sie an, es sei  $e=3$  und die Länge des Modulus  $N$  sei 1024 Bits. Bis zu welcher Länge (in Bits) der Nachricht  $m$  gilt damit (2), d.h. bis zu welcher Länge der Nachricht ist die Verschlüsselung einfach umkehrbar, ohne den Modulus zu faktorisieren?

Dieses Problem kann z.B. entschärft werden, indem der Exponent so gewählt wird, dass  $m^e < N$  und damit  $m < \sqrt[e]{N}$  nicht mehr auftritt. Beim oft verwendeten  $e=65537$  und einem Modulus  $N$  mit 1024 Bit müsste  $m$  kleiner sein als  $\sqrt[65537]{N}$  sein, was bezogen auf die Länge in Bits einer Länge von etwa  $\frac{1024}{65537} = 0,016$  Bits entsprechen würde. Da  $m$  ganzzahlig ist, würde dieses  $e$  das Problem beheben.

## Low Public Exponent Attack

Statt ein grösseres  $e$  zu wählen, könnte man zur Lösung des Problems mit kurzen Nachrichten auch auf die Idee kommen, die Nachricht immer auf die Anzahl Bits des Modulus zu ergänzen, indem ein Feld  $L$  für die Länge der zu verschlüsselnden originalen Nachricht (hier mit  $m_{\text{original}}$  bezeichnet) sowie eine Anzahl Zufallsbits  $R$  verwendet werden. Die effektiv zu verschlüsselnde Nachricht würde damit beispielsweise zu  $m=L||R||m_{\text{original}}$ . Zusätzlich spezifiziert man, dass mindestens  $R$  Zufallsbits vorhanden sein sollten (dies schränkt die maximal erlaubte Länge von  $m_{\text{original}}$  ein). Eine „Randomisierung“ der Nachricht ist übrigens auch unabhängig von der Wahl von  $e$  wünschenswert, weil ansonsten ein Angreifer einfach prüfen kann, ob der abgefangene Ciphertext einen bestimmten vermuteten Inhalt hat. Dazu braucht der Angreifer nur die Vermutung mit dem öffentlichen Schlüssel des Empfängers zu verschlüsseln und das Ergebnis mit dem Ciphertext zu vergleichen.

Leider ist auch dieser schöne Ansatz angreifbar. Zumindest dann, wenn die so erzeugte Nachricht  $m$  an mehrere Empfänger geht. Erweitern wir also unser Standardszenario so, dass Alice dieselbe Nachricht  $m$  noch an zwei weitere Kommunikationspartner schickt, die alle denselben Exponenten, z.B.  $e = 3$ , aber verschiedene Moduli verwenden. Dabei wird die Nachricht natürlich jeweils mit dem öffentlichen Schlüssel dieser Kommunikationspartner verschlüsselt. Folglich haben wir nun basierend auf (1) drei Gleichungen:

$$(3) \quad \begin{aligned} c_1 &= m^3 \bmod N_1 \\ c_2 &= m^3 \bmod N_2 \end{aligned}$$

$$c_3 = m^3 \bmod N_3$$

Wollen wir nun aus diesen drei Gleichungen den Wert von  $m^3$  ermitteln, gibt es dazu in Form des sogenannten Chinesischen Restsatzes ein geeignetes Werkzeug. Dieser Satz sagt aus, dass für ein Gleichungssystem wie (3) dann Lösungen existieren, wenn die Moduli  $N_1, N_2, N_3$  teilerfremd sind, d.h. keine gemeinsamen Primfaktoren enthalten. Da diese Moduli aus je zwei zufällig gewählten Primfaktoren bei der RSA-Schlüsselerzeugung entstanden sind, können wir davon ausgehen, dass dies der Fall ist. Der Chinesische Restsatz sagt zudem aus, dass es dann nicht nur eine Lösung gibt, sondern unendlich viele. Diese haben die Form:

$$(4) \quad x_0 + n(N_1N_2N_3), \quad n = \dots, -2, -1, 0, 1, 2, \dots,$$

Dabei ist  $x_0$  irgendeine beliebige Lösung für  $m^3$  des Gleichungssystems (3). Haben wir also erst einmal irgendeine Lösung  $x_0$  gefunden, können wir danach beliebige Vielfache von  $N_1N_2N_3$  addieren oder subtrahieren und haben danach immer noch eine Lösung. Insbesondere gibt es damit *genau eine* Lösung  $x$  mit  $0 \leq x < N_1N_2N_3$ . Da die zu verschlüsselnde Nachricht bei RSA kleiner als der Modulus und  $\geq 0$  ist, müssen  $0 \leq m < N_1$ ,  $0 \leq m < N_2$  und  $0 \leq m < N_3$  gelten. Wegen der Monotoniegesetze der Multiplikation ist damit  $0 \leq m^3 < N_1N_2N_3$ . Damit liegt auf der Hand, dass  $x = m^3$  sein muss, denn  $x$  ist die einzige Lösung, die  $\geq 0$  und  $< N_1N_2N_3$  ist. Haben wir dieses  $x$  gefunden, können wir mit  $m = \sqrt[3]{x}$  die Nachricht einfach ermitteln.

Aber wie findet man nun ein  $x_0$ ? In der Praxis gibt es verschiedene Wege, Lösungen für solche Systeme zu finden. Wir betrachten hier die Lösungsverfahren nicht näher<sup>7</sup>, sondern geben den Ausdruck für ein  $x_0$  direkt an:

$$(5) \quad x_0 = c_1n_1d_1 + c_2n_2d_2 + c_3n_3d_3 \quad \text{mit } n_i = \frac{N_1 \cdot N_2 \cdot N_3}{N_i} \text{ und } d_i \equiv n_i^{-1} \bmod N_i \quad (i = 1, 2, 3)$$

Dabei ist  $d_i$  das modular inverse Element zu  $n_i$ . Dieses kann mittels des erweiterten Euklidischen Algorithmus effizient berechnet werden. Eine eigene Implementation dieses Verfahrens ist aber meist nicht nötig. So stellt in Java beispielsweise die *BigInteger*-Klasse eine eigene Methode zur Berechnung des modular inversen Elements zur Verfügung.

Bei  $e = 3$  funktioniert diese Attacke also unter der Bedingung, dass dieselbe Nachricht für drei verschiedene Kommunikationspartner verschlüsselt wird. Darüber hinaus funktioniert die Attacke grundsätzlich aber auch bei grösseren  $e$ , die einzige Bedingung ist, dass dieselbe Nachricht jeweils für  $e$  verschiedene Kommunikationspartner verschlüsselt wird. Beim häufig verwendeten  $e = 65537$  müsste dieselbe Nachricht also an 65537 verschiedene Empfänger gesendet werden. Dies ist zwar unwahrscheinlich, aber dennoch denkbar. Ein grösseres  $e$  alleine stellt deshalb noch keine „sichere“ Lösung des Problems in jedem Fall dar.

Welche Lösung gibt es dann? Diese liegt auch hier in der Verwendung eines standardisierten RSA-Verschlüsselungsverfahrens (z.B. PKCS #1), die bereits weiter oben nach der Aufgabe 4 diskutiert wurden. Dabei wird für jeden einzelnen Empfänger die original zu verschlüsselnde Nachricht mit zufälligen Bytes verknüpft, wobei pro Empfänger separate Zufallsbytes verwendet werden. Dadurch ist garantiert, dass die effektiv zu verschlüsselnde Nachricht bei jedem Empfänger anders aussieht, wodurch die obige Attacke nicht mehr funktioniert.

Zusammen mit dem, was in der Vorlesung besprochen wurde, haben Sie nun schon verschiedene Attacken auf RSA kennengelernt, die durch die Verwendung eines standardisierten RSA-Verschlüsselungsverfahrens effektiv verhindert werden können. Bei der Anwendung von RSA ist es deshalb von grosser Wichtigkeit, dass Sie konsequent ein solches Verfahren einsetzen.

<sup>7</sup> Bei Interesse finden Sie im Web unter den Stichworten „Low Public Exponent Attack“ oder „Chinese Remainder Theorem“ mehr Material zur Herleitung der Lösung.

## Common Factor Attack

*The generation of random numbers is too important to be left to chance.*

— Robert R. Coveyou

Während die beiden ersten Angriffe auf einer naiven Anwendung von RSA beruhen, zeigt dieser Angriff, wie wichtig es ist, dass die Primzahlen zur Erzeugung des RSA-Schlüsselmaterials absolut zufällig gewählt werden.

Wie Sie im Unterricht gelernt haben, beruht die Sicherheit von RSA darauf, dass es schwierig ist, grosse Zahlen in ihre Primfaktoren zu zerlegen. Ein viel einfacheres Problem ist das finden des grössten gemeinsamen Teilers (ggT) zweier Zahlen. Gehen wir davon aus, dass wir die Moduli  $N_1 = p_1 * q_1$  und  $N_2 = p_2 * q_2$  der öffentlichen Schlüssel von Bob und Carol haben. Die Primfaktoren zur Bildung der Moduli kennen wir natürlich nicht. Berechnen wir nun den ggT erhalten wir folgendes, falls  $p_1, p_2, q_1, q_2$  alles unterschiedliche Primzahlen sind:

$$\text{ggT}(N_1, N_2) = 1$$

Die Berechnung des ggT bringt uns also in diesem Fall nicht weiter. Falls aber  $N_1$  und  $N_2$  einen gemeinsamen Primfaktor  $g$  hätten, z.B.  $p_1 = q_2 = g$ , dann wäre

$$\text{ggT}(N_1, N_2) = g$$

und somit einer der Primfaktoren gefunden. Daraus lässt sich dann auch der jeweils andere Faktor einfach berechnen:  $p_2 = \frac{N_2}{g}$  resp.  $q_1 = \frac{N_1}{g}$ . Mit den Primfaktoren lassen sich dann natürlich auch die beiden privaten Schlüssel einfach berechnen (siehe Vorlesungsunterlagen zum RSA-Verfahren).

Da also das Vorkommen des gleichen Primfaktors in zwei oder mehr öffentlichen Schlüssel verheerend ist, könnte man meinen, dass dies in der Praxis vermieden wird, indem die Primfaktoren absolut zufällig gewählt werden. Wenn wir von RSA mit einem Modulus der Länge 1024 Bits ausgehen und annehmen die beiden Faktoren sind beide 512-Bit Zahlen, dann gibt es gemäss dem *Prime Number Theorem* (PNT) ca.

$$\frac{2^{512}}{\log(2^{512})} - \frac{2^{511}}{\log(2^{511})} = \frac{2^{512}}{512} - \frac{2^{511}}{511} \sim 2^{501}$$

Primzahlen dieser Länge. Die Wahrscheinlichkeit ist also praktisch gleich Null, dass bei absolut zufälliger Wahl der Primfaktoren zweimal der gleiche Primfaktor gewählt wird, auch wenn viele Milliarden von RSA Schlüsseln erzeugt würden.

Leider werden diese Primzahlen aber nicht immer völlig zufällig gewählt. In Ihrer Studie mit dem Titel „Ron was wrong, Whit is right“ berechneten Arjen Lenstra et al. für einige Millionen von öffentlichen Schlüsseln den ggT und konnten so insgesamt zu 13'000 dieser Schlüssel den privaten Schlüssel herausfinden. Nadia Heninger<sup>8</sup> et al. haben in einer ähnlichen Studie ähnliche Ergebnisse erhalten und noch etwas mehr Details zu den Gründen für dieses Ergebnis geliefert. Einer der identifizierten Gründe war, dass Geräte wie Router, VPN-Geräte, Firewalls usw., ihr Schlüsselmaterial oft beim erstmaligen Start des Gerätes erzeugen. Da die Geräte alle vom gleichen Ausgangszustand frisch ab Fabrik starten, fehlt es kurz nach dem Start den Entropiequellen im System, die als Basis für die Erzeugung des Schlüsselmaterials verwendet werden, oft an genügend Entropie<sup>9</sup>.

Betrachten wir kurz, wie die grundlegenden Schritte zum Erzeugen eines Modulus  $N$  sind. Für die Erzeugung der Primzahlen wird meist ein *Pseudo Random Number Generator* (PRNG) verwendet, der zu Beginn mit einem sogenannten *Seed* initialisiert werden muss. Der Vorgang könnte also in Pseudo-Code so formuliert werden:

<sup>8</sup> Mining your Ps and Qs: Detection of widespread weak keys in network devices, Nadia Heninger, Zakir Durumeric, Eric Wustrow, J. Alex Halderman. Usenix Security 2012

<sup>9</sup> Eine detaillierte Betrachtung der Entropiequelle und des PRNGs von Linux (/dev/(u)random) finden Sie hier: [https://www.bsi.bund.de/DE/Publikationen/Studien/LinuxRNG/index\\_hm.html](https://www.bsi.bund.de/DE/Publikationen/Studien/LinuxRNG/index_hm.html)

```
prng.seed(seed)
p = prng.generate_random_prime()
q = prng.generate_random_prime()
N = p*q
```

Da ein PRNG für den gleichen *Seed* jeweils die gleiche Zufallszahlenfolge erzeugt, ist es absolut zentral, dass der *Seed* genügend Entropie hat, also genügend zufällig ist.

Das Problem ist nun aber, dass wenn der *Seed* bei zwei Systemen identisch ist – weil sie eben z.B. vom gleichen Ausgangszustand frisch ab Fabrik gestartet wurden – und der Vorgang wirklich gemäss obigem Pseudo-Code stattfindet, nicht nur ein gemeinsamer Faktor vorhanden ist, sondern gleich **beide Faktoren** identisch sind. Diese Systeme haben also dasselbe RSA Schlüsselmaterial. Dies ist natürlich bis auf Spezialfälle kaum wünschenswert, da so beide Systeme jeweils die verschlüsselten Nachrichten für das andere System entschlüsseln können. Ein solcher Spezialfall könnte z.B. eine Organisation sein, die mehrere Domänen besitzt, deren zugehörigen Webseiten von verschiedenen Servern ausgeliefert werden. Der Einfachheit halber könnte die Organisation nun dasselbe Schlüsselmaterial für alle Domänen auf allen Servern verwenden. Einfach ist das z.B. deshalb, weil die Organisation dann nur ein einziges Zertifikat (mehr zu Zertifikaten erfahren Sie später in diesem Modul) für all ihre Domänen bzw. Server benötigt. Bei Interesse vergleichen Sie mal den öffentlichen Schlüssel resp. die Zertifikate der Hosts *www.google.com* und *www.youtube.com* und die IP-Adressen der Server, die diese Seiten ausliefern.

Aus Sicht eines Angreifers ist es kaum interessant, wenn zwei Systeme dasselbe Schlüsselmaterial verwenden. Die Situation ist bis auf die leicht vergrösserte Angriffsfläche – der Angreifer kann nun bei zwei Systeme nach Schwachstellen Ausschau halten die Ihm Zugang zum privaten Schlüssel verschaffen könnten – analog zur Situation mit nur einem System.

Damit nur ein Faktor gemeinsam ist und somit der skizzierte Angriff mit dem ggT möglich wird, müsste auf den betroffenen Systemen die Primzahlen wie folgt generiert werden:

```
prng.seed(seed)
p = prng.generate_random_prime()
prng.add_randomness(bits)
q = prng.generate_random_prime()
N = p*q
```

Wie Nadia Heninger et al. herausgefunden haben, entspricht dies dem in OpenSSL<sup>10</sup> implementierten Vorgang zur Erzeugung des RSA-Modulus. Gemäss Heininger et al. wurden die meisten der entdeckten verwundbaren Schlüssel dann auch effektiv mittels OpenSSL erzeugt.

## Alice's geheime Nachrichten

**Aufgabe 11:** Zum Abschluss des zweiten Teils erhalten Sie 6 mit RSA verschlüsselte Nachrichten, die Alice an 6 verschiedene Empfänger Bob, Carol, Dave, Fred, Gustav und Hans geschickt hat, und die von Eve für Sie abgefangen wurden. Die Nachrichten bestehen (in unverschlüsselter Form) jeweils aus ASCII-Text (ein Byte = ein Zeichen). Die Dateien, welche die Ciphertexts und die Public Keys der Empfänger beinhalten, sind die folgenden:

Empfänger	Ciphertext Datei	Public Key Datei
<b>Bob</b>	message-bob.enc	bob.pub
<b>Carol</b>	message-carol.enc	carol.pub
<b>Dave</b>	message-dave.enc	dave.pub
<b>Fred</b>	message-fred.enc	fred.pub

<sup>10</sup> OpenSSL umfasst Implementierungen der Netzwerkprotokolle und verschiedener Verschlüsselungen (insb. Auch TLS) sowie das Programm *openssl* für die Kommandozeile zum Beantragen, Erzeugen und Verwalten von Zertifikaten und zugehörigem Schlüsselmaterial.

<b>Gustav</b>	message-gustav.enc	gustav.pub
<b>Hans</b>	message-hans.enc	hans.pub

Die Moduli der Public Keys haben alle eine Länge von 1024 Bits. Die Ciphertexts haben alle die Länge 128 Bytes, d.h. sie wurden mit jeweils einer 1024-Bit RSA-Operation erzeugt. Ihre Aufgabe ist es nun, den Inhalt von allen 6 Nachrichten zu entschlüsseln. Setzen Sie dazu Ihr Wissen über die drei soeben vorgestellten Angriffe – „Angriff auf Nachrichten geringer Länge“, „Low Public Exponent Attack“ und „Common Factor Attack“ – ein.

Damit Sie nicht bei Null anfangen müssen, stellen wir Ihnen die Java-Klassen `RSAAattacks.java` und `RSACryptionUtil.java` zur Verfügung. Um diese Klassen und die oben aufgeführten Files mit den Ciphertexts und Public Keys zu erhalten, laden Sie `RSAAattacks.zip` von OLAT herunter, verschieben Sie die Datei an einen geeigneten Ort (auf dem Ubuntu-Image am besten auf den Desktop) und entzippen Sie sie. Dies erzeugt ein Verzeichnis `RSAAattacks`. Erstellen Sie dann auf der Basis der Files in diesem Verzeichnis ein Projekt in Ihrer Entwicklungsumgebung.

Wenn Sie mit dem Ubuntu-Image arbeiten, dann gehen Sie wie folgt vor, um das Projekt in *NetBeans* zu erzeugen und zu builden und um das Programm zu starten:

- Wählen Sie *File* → *New Project...* → *Java* → *Java Application*, dann *Next*.
- *Name and Location*: Spezifizieren Sie bei *Project Name* den Namen `RSAAattacks` und bei *Project Location* das Verzeichnis `/home/user/securitylabs`. Stellen Sie sicher, dass die Checkbox bei *Create Main Class* nicht markiert ist. Klicken Sie *Finish*.
- Beenden Sie *NetBeans* und kopieren Sie alle Files aus dem oben beim Entzippen erzeugten Verzeichnis `RSAAattacks` in das Verzeichnis `/home/user/securitylabs/RSAAattacks/src/`.
- Starten Sie *NetBeans* erneut. Wenn Sie im linken Teilfenster das Projekt `RSAAattacks` expandieren, so sollten die kopierten Files unter *Source Packages* ersichtlich sein. Nun können Sie mit der Implementierung starten
- Um das Projekt zu builden machen Sie einen Rechtsklick auf das Projekt `RSAAattacks` im linken Teilfenster und wählen Sie *Clean and Build*.
- Um das Programm zu starten, öffnen Sie ein Terminal, wechseln Sie in das Verzeichnis `/home/user/securitylabs/RSAAattacks/build/classes` und geben Sie `java RSAAattacks` ein.

Die beiden Java-Klassen sind wie folgt zu verstehen: Die Klasse `RSAAattacks.java` stellt die Anwendung dar, welche die Nachrichten entschlüsseln soll. Die `main`-Methode soll die Steuerung der Angriffe übernehmen und die entschlüsselten Nachrichtentexte ausgeben, während die beiden Methoden `tryCommonFactorAttack` und `tryLowExponentAttack` die oben beschriebenen Attacken „Common Factor Attack“ und „Low Public Exponent Attack“ implementieren sollen. Die Javadoc-Kommentare der beiden Methoden geben an, welche Eingabedaten benötigt werden und was das Ergebnis des jeweiligen Angriffs ist. Die `main`-Methode enthält zudem einige Codezeilen in Form von Kommentaren, die bei der Lösung der Aufgabe hilfreich sind. Die Klasse `RSACryptionUtil.java` stellt Methoden zur Verfügung, die zur Erfüllung der Aufgabe hilfreich sind – studieren Sie diese, bevor Sie mit der Implementierung beginnen. Dazu gehört auch die Methode `cubeRoot`, mit der die Kubikwurzel (3-te Wurzel) berechnet werden kann. Damit können Sie direkt den „Angriff auf Nachrichten geringer Länge“ durchführen.

*Hinweise:*

- Laden und betrachten Sie die Public Keys und berücksichtigen Sie die gewonnenen Erkenntnisse um zu entscheiden, welche Ciphertexts mit welchem Verfahren geknackt werden könnten.

- Verwenden Sie grundsätzlich die Klasse `BigInteger` um korrekt mit grossen Zahlen umzugehen. Die Klasse bietet mehrere Methoden für Rechenoperationen wie `add`, `subtract`, `multiply`, `divide`, `mod` etc.
- Die `BigInteger` Klasse bietet weitere nützliche Methoden wie z.B. `modInverse` (zur Berechnung des modular Inversen Elements) und `gcd` (zur Berechnung des ggT).
- Einen `BigInteger` kann man mit dessen Methode `toByteArray` in einen Byte-Array umwandeln und daraus mittels `new String(bytearray, StandardCharsets.US_ASCII)` einen String erzeugen, wobei die Bytes als ASCII-Zeichen interpretiert werden.
- In `tryCommonFactorAttack` müssen Sie den privaten Exponenten aus den beiden Primfaktoren des Modulus und dem öffentlichen Exponenten berechnen. Die zugehörigen Formeln finden Sie in den Vorlesungsunterlagen.

## Praktikumspunkte

In diesem Praktikum können Sie **4 Praktikumspunkte** erreichen:

- Einen Punkt erhalten Sie, wenn Sie dem Betreuer Ihre Antworten auf die Fragen (Aufgaben 1-8 und 10) in der Praktikumsanleitung zeigen und diese Antworten mehrheitlich korrekt sind. Ebenfalls müssen Sie allfällige Kontrollfragen des Betreuers richtig beantworten können.
- Einen weiteren Punkt erhalten Sie für das Lösen der Aufgabe 9 zum Thema Hybride Verschlüsselung im Teil 1. Zeigen Sie den hybrid-verschlüsselten (und signierten) Ciphertext dem Betreuer und demonstrieren Sie ihm, dass Sie den Ciphertext wieder entschlüsseln und die Signatur erfolgreich verifizieren können.
- Zwei weitere Punkte erhalten Sie für das Lösen der Aufgabe 11 im Teil 2. Zeigen Sie den Java-Code und die entschlüsselten Nachrichtentexte dem Betreuer.

## Anhang: CrypTool Workaround bei Problemen mit dem Zertifikatsimport

Wegen eines Bugs kommt es gelegentlich vor, dass CrypTool beim Import eines Zertifikats ein «Durcheinander» mit der Verwaltung der Schlüsselpaare erhält. Das kann z.B. passieren, wenn man sich beim Import eines Zertifikats bei der PIN vertippt, worauf sich das Zertifikat bei einem erneuten Versuch trotz richtiger PIN-Eingabe manchmal überhaupt nicht mehr importieren lässt. Dieses „Durcheinander« bei der Verwaltung der Schlüsselpaare kann zudem auch dazu führen, dass bei der Durchführung einer Hybrid-Entschlüsselung CrypTool nicht das im GUI gewählte Zertifikat verwendet und dadurch nur zufällige Daten resultieren (statt der ursprüngliche Klartext inkl. der Signatur). In diesem Fall können Sie den folgenden Workaround anwenden:

- CrypTool schliessen.
- Registry öffnen (bei Verwendung von Windows *Start* → *regedit* eingeben; bei Verwendung von Wine *wine regedit* in einem Terminal eingeben) und den Eintrag *CrypTool* unter *HKEY\_CURRENT\_USER/Software* löschen.
- Bei Verwendung von Windows den Ordner *CrypTool* unter *C:\Users\user\AppData\Roaming* löschen. Wenn Sie mit Wine arbeiten, so müssen Sie den entsprechenden Ordner im angelegten Windows-Verzeichnisbaum löschen. Auf dem Ubuntu-Image befindet sich dieser unter */home/user/.wine/drive\_c/users/user/Application Data*.
- CrypTool neu starten, worauf die Zertifikate wieder korrekt erzeugt / importiert werden können.

Wenn auch dies nichts hilft, so wenden Sie sich an den Betreuer.