Fast Batch Verification for Modular Exponentiation and Digital Signatures

Mihir Bellare
Juan. A. Garay
Tal Rabin

Bearbeitet von: Enrico Hartema

28. Juli 2009

Zusammenfassung

Inhalt dieser Ausarbeitung des im Titel genannten Papers ([1]) ist das schnelle Verifizieren von kryptographischen Operationen. So geschieht es relativ häufig, dass eine Anwendung $g^x = y$ verifiziert werden muss. Das simple Berechnen dieser Verfifikation kann dabei bei vielen Elementen dieser Form mitunter sehr zeitaufwändig sein, in dieser Ausarbeitung werden Algorithmen beschrieben, die dieses durch den Einsatz von probabilistischen Methoden beschleunigen können.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Das Verifizieren von Nachrichten ist in der Kryptographie nichts Ungewöhnliches. So vertraut man, um eine Analogie zum realen Leben als Beispiel zu nennen, einer Unterschrift ohne Vergleichsmaterial auch nicht ohne Weiteres, da jeder Mensch diese einfach angefertigt haben könnte. In der Kryptographie verhält es sich genauso, eine Signatur ohne Verifikation ist als wertlos zu betrachten.

Der Großteil dieser Ausarbeitung wird sich mit folgendem Problem beschäftigen. Gegeben seien die Parameter (g, x, y), wobei g ein Generatorelement einer Gruppe G darstellen soll, x der Exponent (oder einer anderen Operation, Exponentiation ist aber bei einer Vielzahl von kryptographischen Verfahren üblich) für die benötigte Verifizierung und y der Wert, mit dem verglichen werden soll.

Es soll also $g^x = y$ gelten für mehrere x und y gelten. Diese kann man natürlich simpel mit Algorithmen wie dem Square-and-Multiply-Algorithmus (6.1) berechnen, allerdings känn bei großen Exponenten dieser Vorgang einiges an Zeit beanspruchen. In dieser Ausarbeitung werden Methoden präsentiert, die die Berechnung von mehreren dieser sogenannten Instanzen schneller als ein simples Nachrechnen durchführen können. Offensichtlich ist dieses nur durch den Einsatz von probabilistischen Methoden möglich, denn nur durch ein komplettes Nachrechnen ist ein maximaler Grad an Sicherheit vorhanden, der garantiert, dass die Signatur zumindest im mathematischen Sinne korrekt ist. Dadurch sinkt natürlich besagter Grad, denn aufgrund von Vernachlässigung steigt eine mögliche Fehlerquote. Diese gilt es klein zu halten.

Gerade in Bereichen, in denen die Hardwareaustattung eher minimalistisch ausgelegt ist, wäre eine schnelle Verfifikation von Signaturen von Vorteil. Ein Beispiel hierfür wäre ein eingebettetes System, dass die Aufgaben hat, mehrere Schlüssel auf Smartcards zu verifizieren. Besonders bei hohem Andrang möchte man hier nicht unnötige Wartezeiten in Kauf nehmen. Ein weiteres Beispiel ist die Car2Car-Kommunikation ([6]), bei der viele Autos bestimmte Signale senden und man schnell verifizieren möchte, dass das Signal auch zu einem entsprechenden Auto gehört, da ansonsten ein Angreifer in der Lage wäre, durch falsche Daten Verkehrsunfälle zu realisieren. An einer Kreuzzung könnte es dabei Signale von einer Vielzahl von Autos geben, hierbei ist daher unbedingt eine schnelle Verifikation erforderlich.

1.2 Kryptographische Algorithmen

Um zu verdeutlichen, welche Algorithmen kryptographische Verifikationen offerieren, sollen an dieser Stelle noch einmal häufig verwendete genannt und kurz erklärt werden. Sind Kenntnisse über diese vorhanden, so kann dieser Abschnitt übersprungen werden.

Weiterhin gibt es, wie beispielsweise beim DSS-Algorithmus, zum Teil einzelne Unterschiede gegenüber dem Standardalgorithmus. Auch diese sollen hier erklärt werden, in den entsprechenden Kapiteln wird genauer auf die Beweggründe eingegangen.

1.2.1 RSA

RSA ist ein asymmetrisches Kryptoverfahren, bei dem die Signatur nach folgender Berechnungsvorschrift erstellt wird: $M \mapsto M^d = s \bmod N$. Dabei ist M die zu signierende Nachricht, s die erstellte Signatur und d der private Schlüssel. N setzt sich dabei aus zwei Primfaktoren p und q zusammen, der private Schlüssel d steht mit dem öffentlichen e wie folgt im Zusammenhang: $e * d = 1 \bmod \phi(N)$. Da einem Angreifer aufgrund der Größe von N (üblich sind mindestens 1024 Bit) das Faktorisieren von N schwer fallen wird, kann er e nicht trivial bestimmen. Die Verifikation gestaltet sich ebenfalls simpel, es gilt $s^e = (M^d)^e \mapsto m \bmod N$. Oftmals wird nicht die komplette Nachricht, sondern nur der Hashwert einer Nachricht $H(M)$ signiert. Dieses nennt man auch das Hash-Then-Sign-Paradigma und wird aus Effizienzgründen durchgeführt. Eine Operation $Sign_{N,d}(M)$ wird also mit $H(M)^d$ berechnet, eine Verifikation ist erfolgreich, wenn $x^e = H(M)$ gilt. Im Folgenden soll das Full Domain Hash Schema mit RSA genutzt werden (FDH-RSA), hierbei wird einfach $\{0,1\}^n$ auf $\mathbb{Z}_{\mathbb{N}}^*$ abgebildet. Ein Standard für RSA ist unter [3] zu finden.

1.2.2 DSS

Das Digital Signature Standard-Schema ([2]) spezifiziert einen Algorithmus, der primär für Signaturen ausgelegt worden ist. Dieser Algorithmus wird auch Digital Signature Algorithm (DSA) genannt. Wir wollen im Folgenden die klassische Variante betrachten.

Folgende Werte sind öffentlich und als konstant anzunehmen: Eine Primzahl p (bspw. 512-Bit lang), eine Primzahl q mit 160 Bit, die so gewählt worden ist, dass q $p-1$ teilt. Ebenso wird ein Generator h für $\mathbb{Z}_p^*$ benötigt. Nun soll $g = h^{(p-1)/q} \bmod p$ sein, was dazu führt, dass g eine Untergruppe G von $\mathbb{Z}_p^*$ erzeugt. Wir werden bei verschiedenen Batch Verifizierern benötigen, dass die Gruppenordnung prim ist, dieses ist hier gegeben, da q eine Primzahl sein soll.

DER Signierer besitzt einen geheimen Schlüssel $x \in \mathbb{Z}_q$. Weiterhin existiert ein öffentlicher Schlüssel $y = g^x \in G$. Die Nachricht, die zu signieren ist, soll mit $m \in \mathbb{Z}_q$ bezeichnet werden (hier wird jedoch eigentlich $H(m)$, also die Hashfunktion, gemeint, aufgrund der Eindeutigkeit werde ich jedoch die Notation des Papers beibehalten, zumal dieses durch die Angabe der Gruppe eindeutig ist). Es wird ein Zufallswert gewählt, um deterministisches Signieren zu vermeiden, dieser wird mit $k \in \mathbb{Z}_q$ bezeichnet. Wir wollen einen Wert $\lambda = g^k \in G$ und $r = \lambda \bmod q$ einführen. Der Wert λ, welcher mit r bis auf die Reduktion $\bmod q$ identisch ist, ist für den klassischen Algorithmus nicht von Belang, allerdings soll im folgenden Abschnitt eine abgewandelte Version von dieser DSS dargestellt werden, welche längere Signaturen erzeugt (da die Reduktion nicht stattfindet), aber im Rahmen der Batch Verifikation sich als nützlich erweisen wird. Nun soll $s = k^{-1}(m + xr) \bmod q$ gelten. Damit ist die Signatur auch vollständig, diese ist (r, s) und ist bei den oberen angenommen Werten 320 Bit lang. Die Verifikation gestaltet sich einfach, hier muss nur $w = s^{-1} \bmod q$ berechnet werden und danach kann man überprüfen, ob $r = (g^{mw} y^{rw} \bmod p) \bmod q$ gilt. Durch Einsetzen kann man sehr einfach die Korrektheit zeigen, ein Sicherheitsbeweis existiert nicht [7]. Wie man sieht, sind zwei Exponentiationen in $\mathbb{Z}_{\text{!}}^*$ durchzuführen, ergo haben wir es hier mit 512 Bit Exponentiationen zu tun.

1.2.3 DSS*

DSS* soll sich von dem vorherigen Algorithmus nur in einer Hinsicht unterscheiden: Die Signatur soll (λ, s), nicht (r, s) sein. Dieses ist auch der Grund, warum wir λ spezifiziert haben. Durch die fehlende Reduktion $\bmod q$, welche einen 160-Bit Wert zur Folge hatte, steigt natürlich auch die Größe der Signatur an, wir behalten einen 512-Bit Wert und senden diesen zusammen mit s, welches eine Bitlänge von 160 Bit besitzt. Dieses macht eine Gesamtlänge von 672 Bit. Die Verifikation muss natürlich ebenfalls angepasst werden, nun muss bei erfolgreicher Verifikation $\lambda = g^{mw} y^{rw} \bmod p$ gelten. Ansonsten sind keine Unterschiede vorhanden.

2 Batch Verifikation

2.1 Allgemeines Vorgehensweise

Schon der Name Batch (engl. für Schwung, Stapel) lässt das Prinzip erahnen, dass nun einem kompletten Ausrechnen vorgezogen wird. Anstatt einer kompletten Menge an Nachrichten wird nur eine Teilmenge eben dieser betrachtet und auf Korrektheit untersucht. Welche Verfahren sich inwiefern für diese Art der Verfikation eignen, wollen wir nun genauer untersuchen und beschreiben.

Zunächst soll R eine boolsche Relation darstellen, so dass gilt, dass jedes $R(inst) \in \{0,1\}$ ist. Dabei sollen die Instanzen $inst \in R$ sein. Was genau eine Instanz ist, soll später noch erklärt werden. Beispielsweise haben wir eine Relation $R(x,y) = 1$, wenn $g^x = y$. g ist dabei ein Generator und R ein Signaturverifikationsverfahren, wie aus der Einleitung bekannt.

Bezogen auf das Batch-Verfifikationsproblem muss nun für eine Instanzensequenz $inst_1, ..., inst_n \forall\ i \in [n]$ gelten, dass $R(inst_i) = 1\ \forall\ i \in [n]$ ist. Natürlich könnte man einfach alle Instanzen nachrechnen, aber dann hätte man das gleiche Problem des simplen Nachrechnens. Um dieses schneller durchführen zu können, muss man in die Wahrscheinlichkeitsrechnung ausweichen, was ebenfalls auch zu einer bestimmten Fehlerwahrscheinlichkeit führen wird, die es möglichst klein zu halten gilt.

Daher wollen wir einen Batch-Verifizierer einen Algorithmus V nennen, welcher $inst_1, ..., inst_n$ als Eingabe besitzt und ein Ergebnisbit als Ausgabe liefert. So soll $V(inst_1, ..., inst_n) = 1$ liefern, wenn $R(inst_i) = 1\ \forall\ i \in [n]$ gilt. $V(inst_1, ..., inst_n) = 1$ soll allerdings nur mit einer geringen Fehlerwahrscheinlichkeit gelten, wenn ein $R(inst_i) = 0$ existiert. Dabei soll ein Sicherheitsparameter l eingeführt werden, der für die oben geschilderte Situation die Fehlerwahrscheinlichkeit 2^{-l} annimmt.

2.2 Batch Verifizierer bei modularen Exponentiationen

Da viele Signaturverfahren in einer zyklischen Gruppe operieren und demnach Rechnenoperationen auf Basis modularer Exponentiationen durchzuführen sind, gilt es nun, Batch Verifizierer diesbezüglich näher zu untersuchen. Dabei gilt, dass G eine zyklische Gruppe mit der Ordnung q sei und in dieser Exponentiationsfunktion $x \mapsto g^x$ mit $x \in \mathbb{Z}_q$ die Relation $EXP_{G,q}(x,y) = 1$ gelten soll.

Wie in der allgemeinen Vorgehensweise beschrieben, benötigt diese Relation nun auch Instanzen, die eben diese Relation erfüllen können. Anstatt den Werten x und y betrachtet man nun Sequenzen $(x_1, y_1), ..., (x_n, y_n)$, für die wir die Relationsbedingung $EXP_{G,q}(x,y) = 1$ verifizieren wollen. Auch hier ist es natürlich möglich, einfach g_i^x zu berechnen und mit $x_i\ \forall\ i \in [n]$ zu vergleichen. Diesen naiven Ansatz mit den Kosten von n Exponentiationen gilt es aber gerade, zu vermeiden.

Wir werden daher in Kapitel 3 verschiedene Algorithmen betrachten, die diese Art der Verfikation unter unterschiedlichen Rahmenbedingungen durchführen können.

2.3 Formale Definition Batch Verifikation

Sei $R(.)$ eine boolsche Relation ($R(.) \in \{0,1\}$). Eine Instanz für $R(.)$ ist eine Eingabe $inst$, die mit Hilfe der Relation ausgewertet wird. Eine Batch Instanz ist eine Sequenz $inst_1, ..., inst_n$ von Instanzen, dabei bezeichnet n die Größe der Instanz. Eine Batchistanz ist korrekt, wenn $R(inst_i) = 1\ \forall\ i \in [n]$ und inkorrekt, wenn $\exists\ i$ mit $R(inst_i) = 0$.

Ein Batch Verifizierer für eine Relation R ist ein probabilistischer Algorithmus V, der eine Eingabe $X = (inst_1, ..., inst_n)$ enthält und einen Sicherheitsparameter l besitzt. Wenn X korrekt ist, ist die Ausgabe 1, wenn X nicht korrekt ist, ist $P(V \text{ gibt 1 aus}) = 2^{-l}$.

2.4 Screening

Screening geht einen etwas anderen Weg als die Batch Verifikation. Es geht hierbei nicht unbedingt um die Tatsache, ob ein bestimmter Text M_i von jemandem authentisiert wurde, sondern vielmehr um die

Eigenschaft, dass ein bestimmter String x_i eine valide Signatur von M_i ist.

Somit gehen wir von einem festen Signaturverfahren und einem bestimmten öffentlichen Schlüssel pk aus. $Verfify_pk(M, x)$ soll 1 ausgeben, wenn die Signatur x einer Nachricht M valide ist. Würden wir nun eine Batch Verifikation betrachten, hätte man eine Sequenz $(M_1, x_1), ..., (M_n, x_n)$ gegeben (natürlich bezogen auf den öffentlichen Schlüssel pk) und $Verfify_pk(M_i, x_i)$ würde 1 nur mit einer (bei einem guten Algorithmus) geringen Fehlerwahrscheinlichkeit ausgeben, falls ein (M_i, x_i) nicht korrekt wäre.

Screening geht einen anderen Weg. Immer noch von einer Sequenz $(M_1, x_1), ..., (M_n, x_n)$ ausgehend, wird in eben dieser Sequenz nach einer Fälschung gesucht, also ein M_i, was beispielsweise niemals signiert worden ist. Ein sinnvolles Beispiel wären elektronische Münzen, hier möchte man nur wissen, ob alle Münzen valide sind, nicht, ob man jedes Mal eine korrekte Signatur hat.

2.5 Formale Definition Screening

Gegeben sei ein Signaturschema $(Gen, Sign, Verify)$, bestehend aus einem Schlüsselerzeugung-, einem Signier- und einem Verfikationsalgorithmus. Offensichtlich ist der erste probabilistisch, der zweite kann es sein (sollte es meistens, um Determinismus zu verhindern), der dritte ist klar deterministisch. Der Schlüsselerzeugnungsalgorithmus führt $(pk, sk) \xleftarrow{R} Gen(1^k)$ aus, erzeugt also ein Schlüsselpaar mit der Länge k. Signiert wird eine Nachricht M via $x \xleftarrow{R} Sign_{sk}(M)$. Ein Nachrichtensignaturpaar (M, x) wird verifiziert, falls $Verfify_{pk}(M, x) = 1$ ist.

Sei die Sequenz $(M_1, x_1), ..., (M_n, x_n)$ eine Batch Instanz für ein Signaturverfahren $(Gen, Sign, Verify)$ und x_i die angenommene Signatur zu einer Nachricht M_i zu einem gegebenen öffentlichen Schlüssel pk. Sei weiterhin $ScreenTest$ ein Algorithmus, der ein Bit ausgibt. Nun muss eine Fallunterscheidung durchgeführt werden:

- Falls $Verify_pk(M_i, x_i) = 1 \; \forall \; i \in [n]$, dann ergibt $ScreenTest_pk((M_1, x_1), ..., (M_n, x_n))$ ebenfalls 1.
- Ansonsten ist $ScreenTest_pk((M_1, x_1), ..., (M_n, x_n)) = 0$

Angenommen, ein Angreifer A habe Orakelzugriff zu $Sign_sk(.)$, das heißt, er kann sich unbegrenzt neue Nachrichten signieren (ergo einen Chosen-Message-Angriff durchführen). Nach n Anfragen besitzt A eine Batch Instanz $(M_1, x_1), ..., (M_n, x_n)$. Ein $M_i^{'}$ soll nicht legal signiert sein, wenn kein Orakelzugriff stattgefunden hat. A soll jedoch erfolgreich sein, falls seine Batch Instanz ein solches $M_i^{'}$ enth
ält, aber $ScreenTest_pk((M_1, x_1), ..., (M_n, x_n)) = 1$ ergibt. $Succ(A)$ soll dabei die Erfolgswahrscheinlichkeit für einen solchen Angreifer A darstellen. Ein entsprechender Screening-Algorithmus ist also gut, wenn $Succ(A)$ vernachlässigbar ist für einen begrenzten (ppt) Angreifer.

2.6 Kosten modularer Exponentiationen und Multiplikationen

Um nachher überhaupt eine Laufzeitanalyse durchführen zu können, muss vorher geklärt werden, welche Algorithmen als Basis für modulare Exponentiation und Multiplikation eingesetzt werden. Hierfür wird der übliche Square-and-Multiply-Algorithmus (6.1) genutzt, der bei gegebenen $a, b \in G$ a^b mit durchschnittliche $1,5|b|$Multiplikationskosten berechnet. Die Windowing-Methode bietet schon einen Faktor von $1,2|b|$ Multiplikationen, es gibt noch bessere Methoden, die jedoch auf Kosten des Speichers gehen. Entsprechende Erklärungen sind der Literatur zu entnehmen, die im Literaturverzeichnis des Papers angegeben wird (u.a. [8]).

Damit kein expliziter Algorithmus angeben werden muss, werden wir eben diesen als Parameter in die Laufzeitanalyse mit einfließen lassen, dieser soll $ExpCost_G(k_1)$ genannt werden. Dieses sind also die Kosten, um a^b in einer Gruppe G mit der Länge $k_1 = |b|$ zu berechnen. Dieses spart ein Festlegen auf einen bestimmten Algorithmus, da je nach Situation ein anderer präferiert werden könnte, weiterhin ist dieses ein logischer Schritt, da wir die Optimierung auf einem anderen Abstraktionslevel erreichen wollen.

Beispielsweise soll $a^{b_1}, ..., a^{b_n}$ mit $|b_i| = t \; \forall \; i \in [n]$ berechnet werden. Dieses Beispiel ist im Rahmen einer Batch Instanz nicht zufällig so gewählt worden, da es eine typische Sequenz darstellt. Dann ist dieses in

$n * ExpCost_G(t)$ möglich. Der einfacheren Notation halber schreiben wir dieses als $ExpCost^n_G(t)$, die Berechnung der oben genannten Exponentiationen kann mit jedem der genannten Algorithmen durchgeführt werden, da wir diesen als Parameter betrachten, ist dieses vollkommen irrelevant für unsere Betrachtung der Laufzeit. Anzumerken ist ebenfalls, dass wir Additionen vernachlässigen werden, da diese in der Regel auf Hardware deutlich schnell durchzuführen sind als Multiplikationen.

3 Algorithmen für die Batch Verfikation

Zunächst soll G wieder die Gruppe mit Ordnung q mit $|q| = k_1$ darrstellen, in der wir uns befinden. Dabei soll g mit der Länge $|g| = k_2$ ein Erzeuger von G sein. $\forall\ y \in G$ soll gelten, dass ein eindeutiges $i \in \mathbb{Z}_q$ existiert, so dass $y = g^i$ gilt. Wir definieren $EXP_{G,q}(x, y)$ als wahr, wenn $g^x = y$ ist. Äquivalent dazu gilt der diskrete Logarithmus $x = \log_g(y)$. G, q sollen bei der Analyse als fest angenommen werden.

3.1 Atomic-Random-Subset-Test

Der Atomic-Random-Subset-Test soll der erste hier vorgestellte Batch Verifzierer sein. Wenn man sich die Potenzgesetze anschaut, könnte eine einfache Methode könnte vielleicht das Aufaddieren der x_i, also $x = \sum_{i=1}^{n} x_i \bmod q$ und das Multiplizieren der y_i in G, also $y = \prod_{i=1}^{n} y_i$ sein. Danach könnte man einfach verifizieren, ob $g^x = y$ ist. Dieses wäre bei korrekten Instanzen aufgrund der Potenzgesetze der Fall.

Diese simple Idee scheitert allerdinges ebenso einfach, denn eine Batch Instanz $(x + \alpha, g^x), (x - \alpha, g^x)$ mit $\alpha \neq 0$ wird sicherlich korrekt verifiziert, da das α während des Summierens herausfallen wird und die Gleichung $g^{2x} = g^{2x}$ korrekt ist.

Um dieses zu vermeiden, wird eine zufällige Untermenge $S \subset [n]$ ausgewählt und nur für Elemente der Untermenge die oben genannten Operationen durchgeführt. Somit gilt nun, dass $x = \sum_{i \in S} x_i \bmod q$ und $y = \prod_{i \in S} y_i$ ist. Danach muss natürlich noch verifiziert werden, ob $g^x = y$ gilt. Um beim oben genannten Beispiel zu bleiben, würde nun (vielleicht) $S = 1$ sein, also das erste Element ausgewählt werden. Dieses würde zu der Ungleichung $g^{x+\alpha} \neq g^x$ führen, was sicherlich nicht korrekt ist. Die Idee hinter diesem Test ist also, dass das Auswählen einer Teilmenge bösartige Elementpaare splittet.

Lemma

Sei eine Gruppe G gegeben und g ein Generator von G. Bei einer inkorrekten Batch-Instanz $(x_1, y_1), ..., (x_n, y_n)$ für das Batch Verifikationsproblem für $EXP_{G,g}(.,.)$ akzeptiert der Atomic-Random-Subset-Test diese Instanz mit einer Wahrscheinlichkeit von $0,5$.

Beweis

Sei p die Kardinalität von G, also $p = |G|$. Da g Generator von G ist, existieren eindeutige Werte $x_1^{'}, ..., x_n^{'} \in \mathbb{Z}_p^*$, so dass $g^{x_i^{'}} = y_i$ gilt $\forall\ i \in [n]$. Kurz gesagt, g kann alle Werte von y_i erzeugen.

Nun nehmen wir an, dass $\alpha_i = x_i - x_i^{'}$ sei. Nach Annahme sollten hierbei $\alpha_i \neq 0$ existieren. Der Einfachheit halber nehmen wir weiterhin an, dass dieses für α_1 gelten solle (dieses bedeutet aber nicht, dass alle anderen $\alpha_j = 0$ sind fur $j > 1$, es kännten viele $\alpha_j \neq 0$ existieren). Der Test soll nun eine bestimmte Teilmenge S akzeptieren.

Wenn der Test die Menge S akzeptiert, muss gelten, dass $\sum_{i \in S} x_i = \sum_{i \in S} x_i^{'} \bmod p$ ist, da offenbar unser α_1 nicht Teil der Operation war, die Differenz ergo nicht auftrat. Demnach muss $\sum_{i \in S} \alpha_i = 0$ sein, da, wie gesagt, keine Differenzen zwischen x_i und $x_i^{'}$ auftraten.

Nun betrachten wir eine Teilmenge $T \subseteq 2, ..., N$. Wenn die Summe $\sum_{i \in T} \alpha_i = 0$ ist, folgt daraus, dass $\sum_{i \in T \cup \{1\}} = 0$ ist, da α_1 gerade unsere Differenz war. Da fur S keine Fehler auftraten, muss ergo außerdem $S = T$ gelten, ebenso muss der Test für $S = T \cup 1$ scheitern. Somit muss der Test bei mindestens der Hälfte aller Teilmengen von S scheitern.

Eine andere Möglichkeit, dieses einzusehen, ist die Argumentation ueber die Potenzmenge. Angenommen, wir haben zwei Batch-Instanzen (x_1, y_1) und (x_2, y_2), wobei oBdA die erste Instanz inkorrekt sei. Der Random-Subset-Test würde dennoch akzeptieren (fälschlicherweise), wenn er nur die zweite (x_2, y_2) oder keine Instanz als Eingabe bekommen würde, d.h., $S = \{0, 0\}$ oder $S = \{0, 1\}$. Die Kardinalität der Potenzmenge $P(S)$ ist 4, bei nur einem Element ergo scheint die Fehlerwahrscheinlichkeit $1/2$ zu stimmen.

Wenn wir jedoch das Beispiel mit den zwei Batch-Instanzen betrachten, bei denen sich der Fehler hinauskürzt, scheint es so zu sein, dass ein Element nicht den Worst-Case darstellt. Bei $(x+\alpha, g^x), (x-\alpha, g^x)$ haben wir den Fall, dass der Random-Subset-Test akzeptiert, wenn keine Instanz oder beide Instanzen ausgewählt werden, die Fehlerwahrscheinlichkeit ist jedoch immer noch 1/2. Bei drei Elementen funktioniert diese Argumentation, wie man durch leichtes Nachrechnen sehen kann, nicht mehr, die Fehlerwahrscheinlichkeit sinkt.

Natürlich ist 1/2 keine tolle Fehlerwahrscheinlichkeit. Zwar erfüllt sie das geforderte 2^{-l} , jedoch wären größere Werte fur l wünschenswert. Daher muss man diesen Test mehrmals ausführen, um sukzessive zu einer gewünschten Fehlerwahrscheinlichkeit zu kommen. Ein l-maliges Ausführen fuhrt zur geforderten Wahrscheinlichkeit.

Dieses abermalige Ausführen muss natürlich zwangsweise zu eine Verschlechterung der Laufzeit führen. Hatten wir im naiven Fall noch $ExpCost^n_G(k_1)$ als Laufzeit, beträgt diese nun $nl/2 + ExpCost^l_G(k_1)$ Mul- tiplikationen. Der erste Summand folgt aus den Multiplikationen der y_i, bei denen im Durchschnitt immer die Hälfte ausgewählt werden, der zweite aus der Berechnung von g^{x_i} (Dabei ist $k_1 = \lceil \lg(|G|\rceil)$). Wie man sieht, ist der Atomic-Subset-Test fur $n \leq l$ sogar schlechter als das naive Berechnen und laut den Autoren kein unwahrscheinlicher Fall.

Pseudocode

Folgende Operation kann l-Mal ausgeführt werden:

1. For each $i \in [n]$ wähle zufällig ein $b_i \in \{0, 1\}$
2. Sei $S = \{i : b_i = 1\}$
3. Berechne $x = \sum_{i \in S} x_i \bmod q$ und $y = \prod_{i \in S} y_i$
4. Falls $g^x = y$ gilt, akzeptiere, ansonsten verwerfe

3.2 Fast-Mult-Algorithmus

Als kleinen Einschub werden wir nun den Fast-Mult-Algorithmus vorstellen, der fur den nachfolgenden Batch Verifikiationsalgorithmus als eine Subroutine fungieren wird. Dazu wird angenommen, dass $a_1, ..., a_n \in G$ sind und $b_1, ..., b_n$ Ganzzahlen im Bereich zwischen $0, ..., 2^t - 1 \leq |G|$. Wir schreiben diese Ganzzahlen b_i als binäre Strings der Länge t, also $b_i = b_i[t]...b_i[1]$. Dieses geschieht ganz analog zum Square-and-Multiply-Algorithmus (6.1), zu dem Fast-Mult tatsächlich Parallelen aufweist.

Unser vorliegendes Problem soll das Berechnen von $a = \prod_{i=1}^n a_i^{b_i}$ sein, dabei werden alle Operationen in G ausgeführt. Naiv könnte man natürlich nun $c_i = a_i^{b_i} \ \forall \ i \in [n]$ und danach $a = \prod_{i=1}^n c_i$ berechnen, dieses würde jedoch zu $ExpCost^n_G(t) + \ n - 1$ Multiplikationskosten führen. Man ahnt es schon, im Rahmen eines Batch Verifizierers sind diese natürlich fur eine Subroutine eindeutig zu hoch. Folgender Algorithmus macht es jedoch besser:

1. $a := 1$
2. for $j = t$ downto 1 do
3. for $i = 1$ to n do
4. if $b_i[j] = 1$ then
5. $a := a * a_i$
6. $a := a^2$
7. return a

Dabei werden t Multiplikationen in der äußeren Schleife durch das Quadrieren von a ausgeführt, in der inneren Schleife muss der Durchschnitt betrachtet werden. Immer, wenn $b_{i[j]} = 1$ ist, wird ebenfalls eine Multiplikation durchgeführt. Dieses ist, wenn wir von einer Gleichverteilung der Bits ausgehen, bei der Hälfte aller Bits der Fall, was zu dem Wert $nt/2$ Multiplikationen führt. Insgesamt macht dieses ergo $t + nt/2$ Multiplikationen, was im Vergleich zu den obigen Kosten doch einen deutlichen Fortschritt darstellt. Die Korrektheit ist der zusätzlichen Literatur zu entnehmen [8].

3.3 Small-Exponents-Test

Im Grunde schauen wir uns hier nur eine Variante des Atomic-Random-Subset-Test an. Zunächst wählen wir unsere Bits $s_1, ..., s_n$ zufällig aus $\{0, 1\}$. Daraus folgen für x und y, dass $x = \sum_{i=1}^{n} s_i x_i$ und $y = \prod_{i=1}^{n} y_i^{s_i}$ ist. Nach dem Ende dieser Berechnung muss natürlich wieder geschaut werden, ob $g^x = y$ ist. Bisher unterscheidet diese Vorgehensweise also nichts von einer Variante des Atomic-Random-Subset-Tests mit der Menge $S = \{i : s_i = 1\}$ mit der Fehlerwahrscheinlichkeit $1/2$.

Nun muss man sich fragen, warum diese Vorgehensweise vielleicht besser sein könnte. Zunächst werden die $s_1, ..., s_n$ aus größeren Bitstrings gewählt und zwar mit der Länge t mit $t > 1$. Das heißt, wenn wir t klein halten können, führt dies zu einer Exponentiation bei einem (beispielsweise) k_1-langem Exponenten. Bei dem Atomic-Random-Subset-Test hatten wir durch das l-malige Ausführen entsprechend mehr Exponentiationen, hierdurch gewinnt man also an Performance. Allerdings haben wir n neue Exponentiationen hinzugefugt, nämlich fuer das Berechnen von $\prod_{i=1}^{n} y_i^{s_i}$, allerdings natürlich bei einem deutlich kleineren Exponenten, denn k_1 ist oftmals ≥ 1024, bei $t = l$ nur 60. Es stellt sich also die Frage nach der Balance - wie muss t gewählt werden, um zur gewünschten Fehlerwahrscheinlichkeit $1/2$ zu gelangen, ohne einen zu großen Einfluss auf die Laufzeit zu haben. Weiterhin gilt es noch zu zeigen, ob dieser Test diese Fehlerwahrscheinlichkeit auch erreicht.

Nun wollen wir zeigen, dass die Kosten für die boolsche Relation $EXP_{G,g}(.,.)$ (mit g als Generator von G und der Ordnung q) $l + n(1 + l/2) + ExpCost_G(k1)$ Multiplikationen betragen. Dabei gilt für k_1, wie schon in den vorherigen Algorithmen mehrmals adäquat notiert, $k_1 = |q|$.

Beweis

Kosten

Durch das Nutzen des Fast-Mult-Algorithmus (3.2) muss nicht $y_i^{s_i}$ $\forall i$ berechnet und schlussendlich multipliziert werden. Die Operation $y = FastMult((y_1, s_1), ..., (y_n, s_n))$ wird daher aufgerufen. Die Kosten für diesen Aufruf betragen, da $|s_i| = l \; \forall \; i$ ist, $l + nl/2$ Multiplikationen, die genaue Aufschlüsselung dieser Kosten ist im Abschnitt zum Fast-Mult-Algorithmus zu finden. Da $x = \sum_{i=1}^{n} x_i s_i \bmod q$ ist, werden hier n Multiplikationen benötigt. Zuletzt berechnet man noch g^x, was zu den Kosten $ExpCost_G(k_1)$ führt. So erklären sich die oben genannten Gesamtkosten.

Korrektheit

Sei die Eingabe $(x_1, y_1), ..., (x_n, y_n)$ inkorrekt. Damit existiert irgendein (oder auch mehrere) (x_i, y_i), welches die Relation nicht erfüllt. Nun soll $x_i^{'} = \log_g(y_i)$ sein. Diese stellen also die Werte da, die existierten müssten, damit die Eingabe korrekt wäre. Je nachdem, wie viele x_i korrupt sind, gibt es mehr oder weniger i, für die $x_i \neq x_i^{'}$ gilt, nach Annahme ist dieses aber für mindestens ein i korrekt. Somit gibt es auch ein oder mehrere $\alpha_i = x_i - x_i^{'}$ mit $\alpha_i \neq 0$. Der Einfachheit halber soll dieses einmal mehr für $i = 1$ gelten. Auch hier können dennoch weitere α_j mit $j > 1$ ebenfalls ungleich 0 sein.

Wenn der Test nun eine Menge $S := \{s_1, ...s_n\}$ akzeptiert, gilt offenbar
$g^{s_1 x_1 + ... + s_n x_n} = y_1^{s_1} ... y_n^{s_n} = g^{s_1 x_1^{'} + ... + s_n x_n^{'}}$. Wenn wir nun $g^{s_1 x_1 + ... + s_n x_n}$ durch $g^{s_1 x_1^{'} + ... + s_n x_n^{'}}$ dividieren (was wir tun können, da aufgrund der Generatoreigenschaft von g dieser Ausdruck niemals 0 sein kann) und durch geschickte Anwendung von Potenzgesetzen und Ausklammern der s_i den entstandenen Ausdruck umformen, können wir schreiben, dass $g^{s_1 \alpha_1 + ... + s_n \alpha_n} = 1$ gilt. Weiterhin wissen wir, dass g ein Erzeuger von G ist und demnach muss $s_1 \alpha_1 + ... + s_n \alpha_n = 0 \bmod q$ sein, da q die Gruppenordnung

darstellt und g die Gruppe erzeugt, die Elementordnung ergo der Gruppenordnung entspricht. Allerdings haben wir angenommen, dass α_1 gar nicht 0 sein soll. Ebenfalls nach Annahme ist q prim und damit gilt $ggT(\alpha_1, p) = 1$. α_1 hat also eine Inverse β_1 in G und es gilt, dass $\alpha_1\beta_1 \equiv 1 \bmod q$. Nun können wir den Exponenten folgendermaßen umstellen:

$$\begin{aligned} s_1\alpha_1 + ... + s_n\alpha_n &\equiv 0 \bmod q \\ s_2\alpha_2 + ... + s_n\alpha_n &\equiv -s_1\alpha_1 \bmod q \\ -\beta_1 \cdot (s_2\alpha_2 + ... + s_n\alpha_2) &\equiv s_1 \end{aligned}$$

Es gibt also nur genau ein s_1, für welches dieser Ausdruck wahr ist. Um dieses zufällig zu wählen, muss jedes Bit entsprechend gewählt werden, bei einer Länge von l Bit ist die Fehlerwahrscheinlichkeit also wie gefordert 2^{-l}. Weiterhin ist zu beachten, dass der Algorithmus selbst die jeweiligen s_i wählt, somit hat der Angreifer keine Kontrolle hierüber, was eine zusätzliche Einschränkung darstellt.

Pseudocode

1. Wähle zufälligein $s_1, ..., s_n \in \{0,1\}^l$
2. Berechne $x = \sum_{i=1}^{n} x_i s_i \bmod q$ und $y = \prod_{i=1}^{n} y_i^{s_i}$
3. Falls $g^x = y$ gilt, akzeptiere, ansonsten verwerfe

3.4 Bucket-Test

Der letzte hier vorgestellte Batch Verifizierer steht nun in Konkurrenz zum Small-Exponents-Test, der für ein relativ kleines n besonders effizient wirkt. Der Bucket-Test soll hier nun die Lücke für große n schließen.

Ähnlich dem Random-Subset-Test mit dem wiederholten Ausführen des Atomic-Random-Subset-Tests führt der Bucket-Test m-Mal einen Atomic-Bucket-Test aus. Dabei sammeln sich mehrere „Buckets“(engl. für Körbe) an, die wir als $B_1, ..., B_M$ bezeichnen wollen. Für jedes i wird dann zufällig einer dieser Körbe ausgewählt und ein Paar x_i, y_i in diesem abgelegt. In einem bestimmten Korb werden danach die Werte x_i addiert und die Werte y_i multipliziert. Die Ergebnisse eines bestimmten Korbes B_i werden dann dem Paar c_j, d_j zugeordnet für $j \in [M]$. Im gültigen Fall gilt daher, dass $g^{c_j} = d_j$ ist, da wir nur ein paar gültige Paare x_i, y_i ausgewählt haben und sie einer Datenstruktur zugeordnet haben. Falls es ein ungültiges Paar x_i, y_i gegeben hat, was nicht korrekt war, gibt es mit der Fehlerwahrscheinlichkeit $2^{-(m-1)}$ (siehe unten) einen Korb, der diese korrupten Werte beinhaltet, ergo ist $g^{c_j} \neq d_j$. Die Aufgabe dieses Tests ist es also, nachzuschauen, ob für alle $(c_1, d_1), ..., (c_M, d_M) g^{c_j} = d_j \; \forall \; j \in [M]$ gilt. Für dieses Problem kann man dann den Small-Exponents-Test (3.3)verwenden. Dieses führt zu einer einer gesamten Fehlerwahrscheinlichkeit von $2^{-(m-1)}$), wie wir im folgenden Beweis sehen werden:

Beweis

Sei G eine Gruppe mit der Primordnung q und g Generator von G. $(x_1, y_1), ..., x_n, y_n)$ soll eine inkorrekte Batch Instanz für $EXP_g(.,.)$ darstellen. Wie in den Beweisen zuvor (3.3) gilt, dass $x_i^{'} = \log_g(y_i)$ und $\alpha_i = x_i - x_i^{'} \; \forall \; i \in [n]$, dabei soll $\alpha_1 \neq 0$ sein. Ein Korb B_j mit $j \in [M]$ bezeichnen wir als gut, wenn $g^{c_j} = d_j$ gilt. Da für alle i ein Korb B_j gewählt wird, wollen wir mit t_i kennzeichnen, welcher Korb bei der Ausführung des Algorithmus in der i-ten Runde ausgewählt worden ist. Nun soll r die Wahrscheinlichkeit darstellen, dass die Korbauswahl $t_1, ..., t_n$ mit den entsprechenden Körben $B_1, ..., B_M$ gut sei. Dabei behaupten wir, dass $r \leq 1/M = 2^{-m}$ gilt.

Offensichtlich gilt, dass ein Korb B_j gut ist, wenn in ihm keine korrupten Paare (x_i, y_i) vorhanden sind, also nur $\alpha_i = 0$ vorhanden sind. Man kann also $\sum_{i \in B_j} \alpha_i \equiv 0 \bmod q$ schreiben. Nun soll weiterhin angenommen werden, dass alle Paare außer das angenomme korrupte erste Paar (x_1, y_1) in entsprechenden Körben verteilt, also $t_2, ..., t_n$ mit entsprechenden Werten versehen worden sind. Mit diesem Wissen können wir nun eine Menge $B_j^{'}$ definieren, welche alle Körbe beinhaltet, in denen alle Paare (x_i, y_i) verteilt

sind bis auf das erste, von dem wir annehmen, dass es korrupt sei. Ergo ist die Menge $B_j^{'} = \{i > 1 : t_i = j\}$ nach unserer Definition gut. Nun muss aber noch das x_1 verteilt werden, was zur Folge hat, dass ein Korb $B_j^{'}$ nach der Zuteilung als schlecht zu werten ist, da $\alpha_1 \neq 0$ gilt. Die Wahrscheinlichkeit, dass nun das x_1 in einen bestimmten Korb B_j fällt, ist offenbar $1/M$, da es M Körbe gibt. Bei der Auwahl dieser Körbe ist dieses ebenfalls die Wahrscheinlichkeit, dass wir den korrupten Korb ebenfalls erwischen. Somit muss $r \leq 1/M$ gelten. Dawir für den vierten Schritt im Pseudocode (4) eine Fehlerwahrscheinlichkeit von 2^{-m} annehmen (bspw. führen wir den Small-Exponents-Test m-Mal aus), haben wir eine Gesamtfehlerwahrscheinlichkeit für den Atomic-Bucket-Test von $2 \cdot 2^{-m} = 2^{-(m-1)}$.

Nun sollen die Gesamtkosten betrachtet werden. Da wir nach dem Einteilen in Körben (wahrscheinlich) eine kleinere Batch Instanz als vorher haben, müssen wir diese differenziert betrachten. Da wir mehrmals den Small-Exponents-Test ausführen (auf $M = 2^m$ Körbe), müssen wir hier die nur die Laufzeitkosten von eben diesem Test anpassen. Dieser hatte die Kosten von $l + n(1 + l/2) + ExpCost_G(k_1)$. Der Sicherheitsparameter l ist hier das wiederholende Ausführen, also m. Anstatt von 1 bis n berechnen wir nun die Summen und Produkte von 1 bis $M = 2^m$, was im Normalfall weniger sind. Setzen wir diese Werte einfach ein, erhalten wir die Kosten von $m+2^m(1+m/2)+ExpCost_G(|q|)$ Multiplikationen. Dieser Prozess soll insgesamt $\lceil \frac{l}{m-1} \rceil$-Mal ausgeführt werden. Unser Parameter m kann laut Paper durch Suche bestimmt werden, was uns zu den folgenden Gesamtkosten führt:

$\min_{m \geq 2}\{\lceil \frac{l}{m-1} \cdot (n + m + 2^{m-1}(m + 2) + ExpCost_G(k_1)\}$

Dabei ist $k_1 = |q|$. Die hinteren Summanden sind dabei mit dem im vorherigen Abschnitt beschrieben Kosten des Small-Exponents-Test äquivalent, das n folgt aus den Multiplikationen $d_j = \prod_{i \in B_j} y_i$, da alle y_i in den Körben verteilt worden sind und $i \in [n]$ ist. Wie schon erwähnt, ist es laut Paper am besten, den Faktor m durch Suche zu bestimmen

Pseudocode

Benötigen hier einen weiteren Sicherheitsparameter $m \geq 2$, $M = 2^m$. Wiederhole den folgenden Test $\lceil l/(m-1) \rceil$-Mal.

1. Wähle zufällig ein $t_i \in [M] \; \forall \; i \in [n]$
2. Sei $B_j = \{i : t_i = j\} \; \forall j \; \in [M]$
3. Sei $c_j = \sum_{i \in B_j} x_i \bmod q$ und $d_j = \prod_{i \in B_j} y_i \; \forall \; j \in M$
4. Führe Small-Exponents-Test$(c_1, d_1), ..., (c_M, d_M)$ m-Mal (Sicherheitsparameter) aus

Falls alle Subtests akzeptieren, akzeptiert auch der Bucket-Test.

3.5 Primordnung

Für den Small-Exponents-Test (3.3) haben wir beim Beweis angenommen, dass die Gruppenordnung q prim sein muss. Laut Paper ist dieses nicht wirklich eine Einschränkung, da man zu einer Gruppe $\mathbb{Z}_q$ eine adäquate Untergruppe $\mathbb{Z}_p^*$ finden kann. Dabei ist p prim und so gewählt, dass q $p - 1$ teilt. Diese Gruppen haben weiterhin Vorteile aufzuweisen. So gilt das Diskrete-Logarithmus-Problem als schwerer.

Was passiert, wenn die Gruppenordnung nicht prim ist? Dann kann man einfach Gegenbeispiele entwickeln, die nicht mehr die Fehlerwahrscheinlichkeit von 2^{-l}, sondern von $1/2$ aufweisen. Wählt man also eine Gruppe $G = \mathbb{Z}_p^*$ mit der Ordnung $p-1$ (p ist natürlich prim, ergo die Gruppenordnung nicht) und g soll ein Generator von G sein, dann kann man folgende Batch Instanz konstruieren: $(x, p - y), (x, y)$ mit $x \in \mathbb{Z}_{\text{ı}-\text{ȷ}^{\kappa}}$ und $y = g^x \bmod p$. Nun geht man davon aus, dass s_1 gerade ist. Da s_1 zufällig gewählt wird, ist dieses bei der Hälfte aller Operationen der Fall. Daher wird der Small-Exponents-Test diese Instanz auch bei der Hälfte aller Eingaben akzeptieren.

3.6 Laufzeitvergleich

Zunächst sollen abermals alle Algorithmen und deren Multiplikationskosten in einer Tabelle gegenübergestellt werden, um nachhinein durch Einsetzen besagte Kosten interpretieren zu können:

Test	Anzahl Multiplikationen
Naiv	$ExpCost_G^n(k_1)$
Random-Subset-Test	$nl/2 + ExpCost_G^l(k_1)$
Small-Exponents-Test	$l + nl/2 + ExpCost_G(k_1)$
Bucket-Test	$\min_{m \geq 2} \lceil \frac{l}{m-1} \rceil \cdot (n + m + 2^{m-1}m + ExpCost_G(k_1)$

Nun sollen alle Algorithmen mit den Laufzeitkosten von der obrigen Tabelle ausgewertet werden. Dafür wählen wir das n, also die Anzahl der Batch-Instanzen von 5 bis 5000, um verschiedene Größen zu testen. Dabei soll von 1024-Bit-Multiplikationen ausgegangen werden, ergo entspricht $k_1 = 1024$. Ein Sicherheitsparameter von 60 soll laut Paper als sicher angenommen werden, ergo gilt ebenfalls, dass $l = 60$ gesetzt wird. Die Exponentiationen selbst sollen mit Hilfe von Fast-Exponentiationsalgorithmen durchgeführt werden, auf die hier nicht näher eingegangen werden soll, diese können mit Hilfe einer Vorberechnung eine bessere Laufzeit vorweisen, als der Square-and-Multiply-Algorithmus. Die Werte stellen dabei für die naive Berechnung eine gute Referenz dar. Bei unserern angenommen Werten soll eine Exponentiation 200 Multiplikationen benötigen (beim Square-and-Multiply wären das schon ~ 1500, da im Durchschnitt $1,5$ Multiplikationen pro Bit durchgeführt werden).

n	Naiv	Random-Subset	Small-Exponents	Bucket
5	$1K$	$12K$	$\underline{0,4K}$	$4,3K$
10	$2K$	$12,5K$	$\underline{0,6K}$	$4,4K$
50	$10K$	$13,5K$	$\underline{1,8K}$	$5K$
100	$20K$	$15K$	$\underline{3,2K}$	$5,7K$
200	$40K$	$18K$	$\underline{6,2K}$	$7,1K$
500	$100K$	$27K$	$15,2K$	$\underline{10,7K}$
1000	$200K$	$42K$	$30,2K$	$\underline{16,5K}$
5000	$1000K$	$162K$	$150K$	$\underline{56K}$

Da wir 200 Multiplikationen pro Exponentiationen angenommen haben, berechnet sich der naive Wert durch $n \cdot 200$. Der Einfachheit halber werden die Angaben in K für 1000 angegeben, der jeweils beste Wert ist unterstrichen. Wie man sieht, erzielt der Random-Subset-Test eher bescheidene Ergebnisse und ist für kleinere Werte von n sogar schlechter als das naive Nachrechnen. Jedoch eignet er sich gut als einführender Algorithmus, um die Vorgehensweise von Batch Verifizierern zu verdeutlichen. Der Small-Exponents-Test erzielt für besagte n sehr gute Werte, wird jedoch bei großen Werten vom Atomic-Bucket-Test abgelöst.

4 Screening

Bei den Batch Verifizierern sind wir von einer festen Basis ausgegangen mit einem stets wechselnden Exponenten. Dieses trifft auf Signaturen wie bei RSA (1.2.1) nicht zu, hier haben wir einen gemeinsamen Exponenten. Es gilt also die Relation $R_{(N,e)}(x, y) = 1$, falls $x^e \equiv y \bmod N$.

Oftmals wird als öffentlicher Exponent 3 oder ein ähnlich einfacher Wert gewählt, um das Verifizieren schnell ausführen zu können. Da vielleicht aber auch schnelles Signieren ein Ziel ist und privater und öffentlicher Exponent voneinander abhähngen, könnten sich folgende Algorithmen dennoch als nützlich erweisen, weiterhin beschleunigen die vorgestellten Algorithmen auch das Verifizieren für kleinere Exponenten. Hierfür nutzen wir das Hash-Then-Decrypt-Paradigma aus, welches aus Effizienzgründen angewandt wird.

Der Screening Algorithmus, der vorgestellt wird, soll mit FDH-RSA-Signature-Screening bezeichnet werden. Hier bekommen wir als Eingabe $(M_1, x_1), ..., (M_n, x_n)$, also Signaturen und ursprüngliche Nachrichten. Dabei ist $x_i \in \mathbb{Z}_{\mathbb{N}}^*$, N, e sind als öffentlicher Schlüssel gegeben und man hat Orakel-Zugriff auf die Hashfunktion H, da ansonsten keine Verifikation möglich ist. Der Test selbst ist recht einfach. Zunächst wird geschaut, dass alle Nachrichten M_i disjunkt sind. Danach werden die Signaturen multipliziert und die Nachrichten gehasht und ebenfalls multipliziert. Die Ausgabe ist 1, falls beide Produkte gleich sind, ansonsten 0.

Pseudocode

Gegeben sind: N, e und $(M_1, x_1), ..., (M_n, x_n)$

1. Gebe eine Subliste $(\bar{M}_{\bar{1}}, \bar{x}_{\bar{1}}), ..., (\bar{M}_{\bar{n}}, \bar{x}_{\bar{n}})$ aus, so dass alle $\bar{M}_{\bar{i}}$ disjunkt sind
2. Wenn $(\prod_{i=1}^{\bar{n}} \bar{x}_i)^e = \prod_{i=1}^{\bar{n}} H(\bar{M}_{\bar{i}}) \bmod N$ gilt, gebe 1, ansonsten 0 aus

Offensichtlich gibt es nur $2n$ Multiplikationen aus den beiden Produkten und eine Exponentiation. Dieses führt zu den Gesamtkosten von $2n + ExpCost_{\mathbb{Z}_{\mathbb{N}}^*}(|e|)$ Multiplikationen. Für das Eliminieren von doppelten Nachrichten gibt es mehrere Algorithmen, die dieses in akzeptabler Laufzeit vollführen können, auf die wir hier nicht näher eingehen werden.

Dieser Test soll aber kein Batch Verifizierer darstellen. Das erkennt man schon, wenn man ein Gegenbeispiel analog zum Random-Subset-Test auswählt. Man wähle ein M mit x als valide Signatur und einen Wert $\alpha \in \mathbb{Z}_{\mathbb{N}}^* - 1$. Die Batch Instanz $(M, x\alpha), (M, x/\alpha)$ wäre natürlich korrekt, da sich das α genau herausküerzen würde. Ein Angreifer, der diese Signaturen erstellen müsste, müsste Zugriff auf den Wert x haben. Damit würde eine korrekte Signatur existieren, die Nachrichten also valide sein, egal, ob der Angreifer nun die Signaturen manipulieren würde oder nicht. Dieses Beispiel ist hier also nicht relevant.

Korrektheit

Bei der Generierung eines RSA-Schlüssels (N, e, d) werden bei der Eingabe von 1^k als Schlüssellänge zwei $k/2$-Bit lange Primzahlen p, q erzeugt. Die Erfolgswahrscheinlichkeit eines Invertierungsalgorithmus I soll die Wahrscheinlichkeit sein, dass dieser $y^d \bmod N$ ausgibt, also die korrekte Entschlüsselung von $y = x^e \bmod N$. Diese Operation wollen wir als $RSA_{(e)}(1^k)$ bezeichnen. Ein Algorithmus $I(t, \epsilon)$ bricht $RSA_{(e)}$ (dabei ist $t : \mathbb{N} \mapsto \mathbb{N}$ und $\epsilon : \mathbb{N} \mapsto [0, 1]$), falls I maximal $t(k)$ Schritte benötigt und eine Erfolgswahrscheinlichkeit von mindestens $\epsilon(k)$ aufweisen kann. Er führt also endlich viele Schritte aus und gelangt dabei schlussendlich zu einem sicheren Ergebnis. Man kann demnach auch sagen, dass RSA_e eine (t, ϵ)-sichere Auswahl von Einweigfunktionen ist, wenn kein beschriebener Inverter existiert. Wir wollen daher zeigen, dass, wenn RSA eine Einwegfunktion ist, ein Angreifer keine Batch Instanz für den FDH-RSA-Signature-Screening-Test kreieren kann, die eine Nachricht beinhaltet, die niemals von dem Signierer signiert worden ist, aber dennoch durch den Test verifiziert wird.

Wir nehmen an, dass $RSA_{(e)}$ eine (t', ϵ')-sichere Auswahl von Einwegfunktionen. A soll einen Angreifer

darstellen, der einen Chosen-Message-Angriff auf $FDH - RSA$ ausführt und eine vermeintlich korrekte Batch Instanz bekommt, die jedoch nicht vom Signierer signiert worden ist. Diese Instanz soll die Größe n haben und der Angreifer darf q_{sig} FDH-Signaturanfragen und q_{hash} Hashanfragen stellen. Die maximale Laufzeit soll maximal $t(k) = t'(k) - \Omega(nk\log(nk)) - \Omega(k^3) \cdot (n + q_{sig} + q_{hash})$ sein. Für diese gilt, dass der in dieser Laufzeit erzeugte Batch Verifizierer der FDH-RSA-Signature-Screening-Test mit Wahrscheinlichkeit $\epsilon(k) = \epsilon'(k) \cdot (n + q_{sig} + q_{hash})$ den Batch Verifizierer akzeptiert.

Beweis

Zunächst konstruieren wir für unseren Angreifer A einen Invertierungsalgorithmus I für $RSA_{(e)}$ und passen dann die Parameter entsprechend an. I bekommt als Eingabe N, e, y, was keine Probleme darstellt, da der öffentliche Schlüssel und die Signatur/Verschlüsselung immer zur Verfügung stehen sollten. I soll $x = y^d \bmod N$ finden. Wir wollen als pk, also Public-Key, (N, e) bezeichnen. I führt danach $A(pk)$ aus. Dieser wird mehrere Signatur- und Hashanfragen stellen und schlussendlich soll A eine Batch Instanz ausgeben, mit der I das gesuchte x findet. Nun wollen wir I näher beschreiben.

Sei $q = n + q_{sig} + q_{hash}$. I wählt zufällig $l \in \{1, ..., q\}$. Weiterhin soll ein Zähler c mit $c \leftarrow 0$ initialisiert werden. Dabei wird ein Orakel mehrmals angefragt und eine Tabelle aufgebaut, in welcher für jede Nachricht M ein Paar (x_M, y_M) gespeichert wird. Dabei soll jedoch eine Besonderheit berücksichtigt werden. Wenn die Anfrage zum l-ten Mal stattgefunden hat, soll x_M als undefiniert und $y_M = y$ gelten. Ein Hashorakel soll dabei eine Subroutine $H(M)$ und ein Signierorakel eine Subroutine $Sign(M)$ darstellen. Diese sollen im Folgenden vorgestellt werden:

Subroutine $H(M)$

1. $c \leftarrow c + 1$
2. If $c = l$
3. then $y(M) \leftarrow y$; $M^* \leftarrow M$
4. else $x(M) \xleftarrow{R} \mathbb{Z}_M^*$; $y(M) \leftarrow x(M)^e \bmod N$
5. return y_M

Subroutine $Sign(M)$

1. If M keine bisherige Hashanfrage
2. then $y(M) \leftarrow H(M)$
3. If $c = l$
4. then Abbruch
5. else return $x(M)$

Nachdem alle Anfragen gemacht worden sind und der Angreifer alle Antworten bekommen hat, kann A eine Batch Instanz $(M_1, x_1), ..., (M_n, x_n)$ ausgeben. Um definitiv eine disjunkte Nachrichtenmenge zu erhalten, wird noch der Säuberungsschritt durchgeführt. Somit hat man nun die Teilliste $(\bar{M}_1, \bar{x}_1), ..., (\bar{M}_n, \bar{x}_n)$. Natürlich haben wir immer noch, sofern verhanden, eine Nachricht, die vielleicht nicht korrekt signiert worden ist, der Säuberungsschnitt ändert daran nichts.

Wie man in der zweiten Subroutine sieht, wird für jedes i, für das $\bar{M}_i$ noch keine keine Hashanfrage gemacht worden ist, eine durchgeführt. Somit können wir annehmen, dass für jedes $\bar{M}_i$ eine korrespondierende Hashanfrage existiert. Nach Annahme existiert ein nicht korrekt signiertes $\bar{M}_m$. Ein entsprechendes m wollen wir für die weitere Analyse festhalten. Falls $M \neq M^*$ ist, dann hat I nicht korrekt eine

Nachricht geraten, die in der Ausgabe-Batch Instanz sein sollte, aber falsch signiert worden ist. I bricht daraufhin ab. Ansonsten, falls $M^* = M$ gilt, wird x folgendermaßen gesetzt:

$$x = \frac{\prod_{i=1}^{\bar{n}} \bar{x}_i}{[\prod_{i=1}^{m-1} x(\bar{M}_i)] \cdot [\prod_{i=m+1}^{\bar{n}} x(\bar{M}_i)]} \bmod N$$

Dann wird x ausgegeben und gehalten.

Behauptung: $x^e = y \bmod N$ gilt mit Wahrscheinlichkeit $(Succ(A)/q$.
Beweis: Falls die Batch Instanz-Ausgabe von A den $FDH - RSA$-Signatur Batch Verifikation-Test besteht, wissen wir das Folgendes gilt:

$$(\prod_{i=1}^{\bar{n}} \bar{x}_i)^e = \prod_{i=1}^{\bar{n}} H(\bar{M}_i) \bmod N$$

Weiterhin wissen wir, dass $(H(\bar{M}_i) = y(\bar{M}_i) \; \forall \; i \in [\bar{n}]$ gilt. Dieses setzen wir nun einfach ein und erhalten folgenden Ausdruck für $y(\bar{M}_m)$:

$$y(\bar{M}_m) = \frac{\prod_{i=1}^{\bar{n}} \bar{x}_i^e}{[\prod_{i=1}^{m-1} y(\bar{M}_i)] \cdot [\prod_{i=m+1}^{\bar{n}} y(\bar{M}_i)]} \bmod N$$

Nun wollen wir annehmen, dass $M^* = \bar{M}_m$ gilt. Dieses bedeutet, dass der l-te Rateschritt von I korrekt war. Weiterhin wissen wir dann, dass $x(\bar{M}_i)^e = y(\bar{M}_i \; \forall \; i \neq m$ gilt. Dieses können wir dank des Säuberungsschrittes annehmen. Nun gilt nach der weiteren Annahme Folgendes:

$$y(\bar{M}_m) = \frac{\prod_{i=1}^{\bar{n}} \bar{x}_i^e}{[\prod_{i=1}^{m-1} x(\bar{M}_i)^e] \cdot [\prod_{i=m+1}^{\bar{n}} x(\bar{M}_i)^e]} \bmod N$$

Mit der ersten Gleichung für x oben und der vorherigen Gleichung können wir sagen, dass $y(\bar{M}_m = x^e \bmod N$ gilt. Aber ebenfalls gilt $M^* = \bar{M}_m$ und damit $y = y(\bar{M}_m)$. Beides ineinander eingesetzt ergibt die gewünschte Gleichung $x^e = y \bmod N$.

Jedoch muss noch die Ereigniswahrscheinlichkeit des beschriebenen Szenarios überprüft werden. Mit einer Mindestwahrscheinlichkeit von $1/q$ gilt bei einem gewählten l, dass $M^* = \bar{M}_m$ ist. Damit bricht I auch nicht ab und gibt x aus. Weiterhin wissen wir, dass die Erfolgswahrscheinlichkeit von A während des Interagierens mit I die Wahrscheinlichkeit ist, die A beim Interagieren mit den Orakeln hat. Hierfür haben wir $Succ(A)$ angenommen und entsprechend hat I die Wahrscheinlichkeit von mindestens $Succ(A)/q$.

Die Laufzeit von ist maximal $t(k) + \Omega(nk \log(nk)) + \Omega(k^3) \cdot q$. Dabei ist $t(k)$ die Laufzeit von A, $\Omega(nk \log(nk))$ die Laufzeit des Säuberns von den Nachrichten (hierfür wird einfach ein entsprechender Sortieralgorithmus ausgeführt). Die Wahl von t maximiert diese Laufzeit maximal zu $t^{'}(k)$. Da wir angenommen hatten, dass $RSA_{(e)}(t^{'}, \epsilon^{'})$-sicher ist, kann I maximal nur mit einer Erfolgswahrscheinlichkeit von $\epsilon'(k)$ seinen Angriff durchführen. Damit gilt $Succ(A)/q \leq \epsilon'(k)$. Multiplizieren mit q ergibt $Succ(A) \leq q\epsilon'(k)$, was gefordert wurde.

5 Anwendungen

Die Frage nach den eigentlichen Anwendungen zu Batch Verifizierern soll hier beantwortet werden. Während RSA-Signaturen, wie im vorherhigen Kapitel erklärt, nicht fr Batching geeignet sind, kommen nun dennoch häufig genutzte Anwendungen zum Tragen wie beispielsweise der bekannte DSS-Algorithmus.

5.1 Batch Verifikation für DSS-Signaturen

Für die folgende Batch Verifikation wird der angepasste DSS*-Algorithmus (1.2.3) verwendet. Ergo bekommen wir eine Batch Instanz $(\lambda_1, s_1, m_1), ..., (\lambda_n, s_n, m_n)$. Wie bei den vorherigen vorgestellten Algorithmen gilt es nachzuschauen, ob $(\lambda_i, s_i)\ \forall\ i \in [n]$ gilt. Aufgrund der zusätzlichen Operation, die beim Verifizieren ausgeführt werden muss und der Tatsache, dass in den Exponenten von g und y immer m vorkommt, passen wir die Batch Instanz dem Algorithmus ein wenig an. Diese wird daher $(\lambda_1, a_1, b_1), ..., (\lambda_n, a_n, b_n)$ sein, dabei gilt, dass $a_i = s_i^{-1} m_i$ und $b_i = s_i^{-1}\lambda_i$. Nun muss man nur noch nachschauen, ob $\lambda_i = g^{a_i} y^{b_i} \bmod p$ gilt $\forall\ i \in [n]$. Somit haben wir die Multiplikationen im Exponenten aus der Berechnung, müssen diese jedoch dennoch später bei den Gesamtkosten berücksichtigen.

Aufgrund der Ineffizienz des Random-Bucket-Tests (3.6) werden wir nur angepasste Algorithmen des Small-Exponents-Tests (3.3) und des Bucket-Tests (3.4) vorstellen. Wie man sehen wird, sind die Unterschiede nur gering:

Pseudocode Small-Exponents-Test für DSS

Eingabe: DSS* Parameter p, q, g, y (öffentlich) und $(\lambda_1, a_1, b_1), ..., (\lambda_n, a_n, b_n)$

1. Wähle zufälligein $l_1, ..., l_n \in \{0,1\}^l$
2. Berechne $A = \sum_{i=1}^{n} a_i l_i \bmod q$, $B = \sum_{i=1}^{n} b_i l_i \bmod q$ und $R = \prod_{i=1}^{n} \lambda_i^{l_i}$
3. Falls $g^A y^B = R$ gilt, akzeptiere, ansonsten verwerfe

Wie man deutlich erkennen kann, sind die Unterschiede zum originalen Small-Exponents-Algorithmus gering. Durch die zweite Exponentiation benötigen wir noch eine zweite Summe, weiterhin haben wir 2 Exponentiationen $\bmod p$ anstatt nur einer. Es gilt, dieses bei den Kosten der Multiplikationen zu berücksichtigen, was bei Werten für $p = 512$ Bits und $q = 160$ Bits zu den Gesamtkosten von $480 + 60n$ Multiplikationen $\bmod p$ führt.

Pseudocode Bucket-Test für DSS

Benötigen hier einen weiteren Sicherheitsparameter $m \geq 2$, $M = 2^m$.
Wiederhole den folgenden Test $\lceil l/(m-1) \rceil$-Mal.

1. Wähle zufällig ein $t_i \in [M]\ \forall\ i \in [n]$
2. Sei $B_j = \{i : t_i = j\}\ \forall j\ \in [M]$
3. Sei $d_j = \sum_{i \in B_j} a_i \bmod q$, $e_j = \sum_{i \in B_j} b_i \bmod q$ und $c_j = \prod_{i \in B_j} \lambda_i\ \forall\ j \in M$
4. Führe Small-Exponents-Test$(c_1, d_1, e_1), ..., (c_M, d_M, e_M)$ m-Mal (Sicherheitsparameter) aus

Der Bucket-Test akzeptiert, falls alle anderen Subtests akzeptieren.

Auch hier hat sich nur wenig geändert. Ebenfalls muss eine zweite Summe eingeführt werden, die Unterschiede auf den Sub-Test beziehen sich auf die Unterschiede zum eigentlichen Small-Exponents-Test und wurden bereits erklärt. Bei den Kosten ergeben sich somit, bei identischen Annahmen wie oben, $\frac{60(480+60\cdot 2^m)}{m-1}$ Multiplikationen. Im Abschnitt, der sich mit der Laufzeitanalyse für verschiedene n auseinandersetzt (3.6), wurde bereits darauf hingewiesen, dass sich der Bucket-Test für große n gegenüber dem Small-Exponents-Test durchsetzt.

Korrektheit

Wir haben ja schon für die Gruppenordnung eine prime Ordnung vorausgesetzt, da ansonsten der Small-Exponents-Test (und demnach auch der Bucket-Test) leicht anzugreifen sind (3.5). Da unsere Gruppe G die Ordnung q hat und alle Operanden innerhalb dieser Gruppe vorhanden sind, gilt dieses auch bei DSS*. Zwar finden Operationen z.T. auch in der Gruppe $\mathbb{Z}_i^*$ statt, jedoch bleiben die Operanden immer in der Untergruppe, es gibt keine Operanden, die nur in der größeren Gruppe existieren.

5.2 n-Parteien-Signatur-Protokoll

Nun wollen wir ein eigenes Protokoll einführen, mit dessen Hilfe es möglich ist, dass n Parteien eine Nachricht gemeinsam signieren können. Dabei sind alle Parteien über eine sogenannte Brücke verbunden, welche Signale von den Teilnehmern erhält und Operationen auf diese Signale ausführt. Dieses kann eine zentrale Instanz, beispielsweise ein Server, sein. Die Parteien benötigen dabei nur ein Geheimnis x_i, um erfolgreich einen Signaturvorgang durchführen zu können.

Pseudocode

Gegeben soll g als Generator von G sein, eine Liste Γ aller n Teilnehmer, jeder besitzt einen privaten Schlüssel x_i und einen dazugehörigen öffentlichen Schlüssel g^{x_i}. Weiterhin benötigt wird eine kollisionsresistente ([4]) Hashfunktion h und natürlich eine zu signierende Nachricht.

1. $\forall\, i \in [n]$ wählt Teilnehmer i zufällig ein $r_i \in_R \mathbb{Z}_p^*$, berechnet $a_i = g^{r_i}$ und sendet es zu der Brücke
2. Die Brücke berechnet $A = \prod_{i=1}^{n} a_i$ und sendet es an alle Teilnehmer
3. Jeder Teilnehmer i berechnet $s_1, ..., s_n \in \{0,1\}^l$, indem er $h(m, A, \gamma, j$ mit $j = 1, 2, ...$ berechnet Danach berechnet jeder Teilnehmer i $y_i = r_i + s_i x_i \bmod p$ und sendet es zur Brücke.
4. Die Brücke berechnet $Y = \sum_{i=1}^{n} y_i$ und sendet es an alle Teilnehmer
5. Jeder Teilnehmer i berechnet $W = \prod_{i=1}^{n} w_i^{s_i}$ und danach $Z = g^Y \cdot W$.
6. Falls $Z = A$ gilt, ist die Verifikation erfolgreich

Hier wurde abermals der Small-Exponents-Test (3.3) verwendet. Die Schwierigkeit besteht darin, dass alle Parteien dieselben s_i berechnen, dieses wird durch das identische Berechnen mit Hilfe der Hashfunktion gelöst. Die Brücke bekommt alle individuellen Parameter, bspw. a_i, berechnet die Gesamtwerte und verteilt diese wieder, falls benötigt. Die Brücke sollte daher ein leistungsstarker Rechner sein, weitere Informationen sind dem Paper und den Literaturangaben in eben diesem zu entnehmen ([1]).

5.3 Batch Verifikation für den Grad von Polynomen

Nun soll eine von der Kryptographie eher entfernte Möglichkeit zum Einsetzen von Batch Verifizierern besprochen werden. Gegeben soll eine Menge von Punkten $S = (\alpha_1, ..., \alpha_m)$ sein, die Schwierigkeit besteht nun darin, zu entscheiden, ob es ein Polynom mit einem bestimmten Grad gibt, welches diese Punkte besitzt. Dafür führen wir eine boolsche Relation $DEG_{\mathbb{F},t,(\beta_1,...,\beta_m)}(S)$ ein, welche 1 ausgibt, falls es ein Polynom $f(x)$ mit Maximalgrad t gibt. Weiterhin soll es Funktionswerte α_i für alle Stellen β_i geben, ergo muss $f(\beta_i) = \alpha_i\ \forall i \in [m]$ gelten. Die Operationen sollen in einem endlichen Körper $\mathbb{F}$ ([5]) durchgeführt werden.

Nun führen wir eine Batch Instanz $S_1, ..., S_N$ ein, wobei $S_i = (\alpha_{i,1}, ..., \alpha_{i,m})$ gilt. Die Instanz ist korrekt, falls für alle S_i die oben genannte Relation erfüllt wird, also 1 ausgegeben wird.

Um ein Polynom mit einem Grad t mit gegebenen Punkten auszuwerten, benötigt man mindestens $t+1$ Punkte. Genauso wird im Algorithmus die Relation auch ausgewertet, danach gilt es nur noch zu schauen, ob die verbleibenden Punkte ebenfalls auf diesem Graphen sind. Dies würde aber auch bedeuteten, dass eine einfache Verifikation bedeutet, dass man für jede Instanz ein Polynom interpolieren muss, was relativ kostenaufwändig ist. Dieses naive Nachrechnen wollen wir vermeiden. Der Batch Verifizierer, der

hier angenommen werden soll, führt nur eine einfache Interpolation in einem endlichen Körper mit der Größe $|\mathbb{F}| = p$ aus. Die Idee ist, dass die Koeffizienten einfach zufällig gewählt werden, somit entstehen zufällige Linerarkombinationen für ein S_i. Der Algorithmus soll dieses nun verdeutlichen:

Pseudocode

Gegeben sind $S_1, ..., S_N$ mit $S_i = (\alpha_{i,1}, ..., \alpha_{i,m})$ und die Funktionswerte β_i.

1. Wähle zufällig ein $r \in_R \mathbb{F}$
2. Berechne $\gamma_i = r^n \alpha_{i,n} + ... + r\alpha_{i,1}$. Dieses ist effizient durch das Berechnen von $(...((r\alpha_{i,n} + (\alpha_{i,(n-1)}r + \alpha_{i,(n-2)})...)r + \alpha_{i,1}r$ möglich.
3. Gebe 1 aus, falls $DEG_{\mathbb{F},t,(\beta_1,...,\beta_m)}(\gamma_1, ..., \gamma_n) = 1$ gilt, ansonten 0

Wie man sieht, gibt es $O(mn)$ Multipliationen bei der Berechnung von γ_i. Dieses ist dadurch zu erklären, dass es n Mengen S_i gibt und jeweils m Werte α_i. Wie schon erwähnt, arbeiten wir im endlichen Körper $\mathbb{F}$ mit $|\mathbb{F}| = p$. Die Größe des endlichen Körpers ist hierbei für die Multiplikationskosten interessant, da, wenn wir annehmen, dass in einem endlichen Körper der Größe 2^k die Multiplikation $O(k^2)$ Schritte benötigen. Der Algorithmus oben soll als Random-Linear-Combination-Test bezeichnet werden.

Nun wollen wir annehmen, dass ein Polynom $f_j(x)$ existiert, welches $\forall\ i \in [m]$ Funktionswerte besitzt, also $f_j(\beta_i) = \alpha_i$ gilt. Der Grad soll größer als t sein. Zeigen wollen wir, dass der Random-Linear-Combination-Test ein Batch Verifizieer für die Relation $DEG_{\mathbb{F},t,(\beta_1,...,\beta_m)}(.)$ ist und dabei die Laufzeit $O(mn)$ und Fehlerwahrscheinlichkeit von maximal $\frac{n}{p}$ aufweist.

Beweis

Eine Relation ist korrekt, wenn $DEG_{\mathbb{F},t,(\beta_1,...,\beta_m)}(\gamma_1, ..., \gamma_n) = 1$ gilt. Dabei existiert ein Polynom $F(x)$ mit Grad von maximal t, welches alle Werte in der Menge S, welche unsere Funktionswerte enthält, besitzt. Unser oben genannter Algorithmus soll nun ein Polynom $f_i(x)$ erzeugt haben, welches mit Hilfe der Funktionswerte der Menge S_i interpoliert wurde. Dabei kann es sein, dass der Grad größer ist als t, also $deg(f_i) > t$ gilt. Da unser Polynom $F(x)$ alle Werte von S beinahalten soll, also insbesondere auch die Funktionswerte der Submengen S_i, gilt, dass $F(x) = \sum_{i=1}^{n} r^i f_i(x)$ ist. Da nach Annahme das Polynom $F(x)$ den Grad t nicht überschreiten darf, muss die Summe aller Subfunktionen, welche Werte aufwiesen, die einen Grad größer t besaßen, gleich 0 sein, da ansonsten unsere Annahme nicht erfüllt ist. Es gilt also, dass $\sum_{i=1}^{n} r_i f_i(x)|^{t+1} = 0$ ist. Dieses ist nichts Anderes als eine Schreibweise für die Funktionswerte mit Grad größer t. Die Instanz, die eigentlich inkorrekt ist, kann also dennoch vom Random-Linear-Combination-Test akzepiert werden, wenn sich Funktionswerte herauskürzen. Dieses kann jedoch, da unser endlicher Körper die Kardinalität p hat und n Summationen durchgeführt werden, maximal mit $\frac{n}{p}$ geschehen.
Wie schon oben erwähnt, benötigt jede individuelle Interpolation $O(mn)$, die finale und engültige Interpolation jedoch $O(m^2)$ Multiplikationen.

6 Anhang

6.1 Der Square-and-Multiply-Algorithmus

Der Square-and-Multiply-Algorithmus ist ein handelsübliche Werkzeug zum Berechnen modularer Exponentiationen.

1. $z \leftarrow 1$
2. For $k = \log n$ down to 0
3. If $d_k = 1$ then $z \leftarrow z^2 \cdot x \mod n$
4. Else $z \leftarrow z^2 \mod n$
5. Ausgabe: z

Dabei soll hier $x^d \bmod n$ berechnet werden. Die Binärdarstellung von d soll mit $d_{\log n}...d_1$ bezeichnet werden, wobei mit $\log n$ der Logarithmus von n zur Basis 2 bezeichnet wird. Der Ausgabewert z wird quadriert, falls d_k (k ist unser Zähler, der jeweils dekrementiert wird) 1 ist und mit x multipliziert, ansonsten nur. Dieser Algorithmus nutzt eine Schreibweise aus, bei der man den Exponenten in fortgesetzte Quadrate zerlegt. Da man im Durchschnitt annehmen kann, dass die Hälfte der Bits von $d = 1$ sind und in diesem Fall 2 Multiplikationen auszuführen sind (ansonsten nur eine), kann man bei den Gesamtkosten von $1,5 * |n|$ Multiplikationen ausgehen.

Literatur

[1] (1998,Jun) Mihir Bellare, Juan A. Garay, Tal Rabin - Fast Batch Verification for Modular Exponentiation and Digital Signatures

[2] (1994,May) National Institute of Standards and Technology (NIST) - Digital Signature Standard (DSS) [Online] Available: `http://www.itl.nist.gov/fipspubs/fip186.htm`

[3] (2002,Jun) RSA Laboratories - PKCS #1 v2.1: RSA Cryptography Standard [Online] Available: `ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf`

[4] (2006,Sep) G. Leander u.a. - Vorlesungsskript Kryptographie 1 - Kapitel 4, S.83 [Online] Available: `http://caccioppoli.mac.rub.de/website/teachingmaterial/k1-ws0607/SkriptKryptoI.pdf`

[5] (2005,Okt) R. Avanzi - Vorlesungsskript Endliche Körper und ihre Anwendungen [Online] Available: `http://caccioppoli.mac.rub.de/website/teachingmaterial/ek-ss07/ek-skript.pdf`

[6] (2009,Jan) Anna Lisa Ferrara u.a. - Practical Short Signature Batch Verification [Online] Available: `http://eprint.iacr.org/2008/015.pdf`

[7] (2009, Jul) A. May - Vorlesungsskript Kryptographie 2 - S.118 [Online] Available: `http://www.cits.rub.de/imperia/md/content/may/krypto2ss09/kryptoii.pdf`

[8] (1995, Mar) Ernest F. Brickell u.a. - Fast Exponentiation with Precomputation [Online] Available: `http://www.ccrwest.org/gordon/fast.pdf`