Chameleon-Hashing für Interaktive Beweise der Verfügbarkeit dynamischer Daten in der Cloud*

Stefan Rass, Peter Schartner

Forschungsgruppe Systemsicherheit, Institut für Angewandte Informatik
Alpen-Adria Universität Klagenfurt
Universitätsstraße 65-67
9020 Klagenfurt
stefan.rass@aau.at
peter.schartner@aau.at

Abstract: Proofs of Retrievability (PoR) sind interaktive Verfahren, welche die konsistente Existenz von Daten, im Kontext von Cloud-Storage-Diensten, überprüfen lassen. Der triviale Ansatz durch Download und Überprüfung des gesamten Datenvolumens ist bei großen Datenmengen nicht praktikabel, und PoR-Verfahren verfolgen das Ziel, diese Überprüfung wesentlich effizienter und mit geringem Aufwand für den Kunden zu ermöglichen. Dynamische PoR-Verfahren bieten überdies die Möglichkeit, die Daten am Server nachträglich zu verändern. In der vorliegenden Arbeit beschreiben wir eine einfache und effiziente Konstruktion eines dynamischen proof of retrievability auf Basis von Chameleon-Hashfunktionen.

1 Einleitung

Eine häufig genutzte Form von Dienstleistungen im Rahmen von Cloud Computing ist das Angebot von Speicherplatz für Zwecke der Archivierung bzw. des Austausches großer Datenmengen. Hierbei steht neben Vertraulichkeit insbesondere Verfügbarkeit und Integrität im Zentrum der Interessen der Kunden. Im einfachsten Fall können beide Eigenschaften durch den Download und die lokale Überprüfung des gesamten Datenbestandes beim Kunden sichergestellt werden. Da dieses Vorgehen für große Datenmengen ineffizient und nicht praktikabel ist, wurden von [JK07, LEB+03] interaktive Protokolle zum Nachweis von Integrität und Verfügbarkeit großer Datenmengen vorgeschlagen, sogenannte *proofs of retrievability* (PoR). Hierbei handelt es sich um Challenge-Response-Protokolle, welche unter Zuhilfenahme einer geringen Menge von Verifikationsdaten (welche der Kunde speichert), effiziente Überprüfungen von Verfügbarkeit und Integrität großer Datenbestände ermöglichen.

*Die vorliegende Arbeit ist eine modifizierte und erweiterte Fassung des Artikels "Dynamic Proofs of Retrievability from Chameleon-Hashes", erschienen in: Proceedings of the 10th International Conference on Security and Cryptography (SECRYPT), IEEE, 2013, pp. 296-304. [Ras13]

Die im Folgenden verwendete Terminologie ist jener von interaktiven Beweisen angepasst: der *Verifizierer* (*Client*) ist der Kunde, welcher seine Datenbestände bei einem externen (Cloud) Provider (auch *Beweiser, Prover* oder *Server* genannt), speichert. Ein PoR-Protokoll ist ein Beweis von Wissen (proof of knowledge), in dem Sinne, dass ein Wissensextraktor (knowledge extractor; Algorithmus bzw. Protokoll) existiert, welcher aus den Antworten des Servers den gesamten Datenbestand rekonstruieren kann. Obgleich diese Funktionalität wesentlich der Sicherstellung theoretischer Eigenschaften von PoR-Verfahren dient, kann die Wissensextraktion intuitiv als gewöhnlicher "Download" der Daten betrachtet werden.

Im einfachsten Fall kann eine Challenge-Response-Überprüfung der Verfügbarkeit einer Datei F wie folgt ablaufen: der Client sendet einen (zufälligen) Schlüssel k an den Server, welcher mit einem Message-Authentication Code $MAC_k(F)$ der Datei F antwortet. Diesen MAC kann der Client mit einem bei ihm lokal gespeicherten Wert vergleichen (eine endliche Liste gültiger MACs wird *vor* Ausführung des Protokolls berechnet und beim Client gespeichert), um die Integrität und Verfügbarkeit der Daten zu überprüfen. Dieses sehr einfache Verfahren besitzt mehrere Nachteile, welche durch verbesserte Verfahren behoben werden:

N1: Das Protokoll lässt nur eine begrenzte Anzahl an Überprüfungen zu, da die Verifikationsdatenmenge beim Client auf einen kleinen Bruchteil der Originaldatenmenge begrenzt sein muss (andernfalls wäre es effizienter, die gesamten Daten beim Client zu belassen).

N2: Der Server muss für die Antwort den gesamten Datenbestand verarbeiten, was im Falle großer Datenmengen zu langen Antwortzeiten führen kann.

N3: Die Daten dürfen sich über ihre gesamte Lebensdauer nicht verändern, da andernfalls die beim Client gespeicherten Antworten ungültig werden würden.

Insbesondere Eigenschaft N3 führt für diese Methode zu der Bezeichnung *statisches* PoR-Verfahren. Diese Einschränkung ist vielen in jüngerer Zeit verfügbaren PoR-Verfahren gemeinsam. Im Gegensatz dazu erlauben *dynamische* PoR-Verfahren auch nachträgliche Veränderungen der Daten. Wir beschreiben nachfolgend ein solches dynamisches PoR-Verfahren, welches explizit alle drei genannten Mängel behebt.

Eine weitere Klassifizierung von PoR-Verfahren kann auf Basis von Eigenschaft N1 getroffen werden: hierbei wird zwischen *schlüsselabhängigen* und *schlüssellosen* Verfahren unterschieden. Letztgenannte erlauben nur eine feste (beschränkte) Anzahl von Wiederholungen (d.h. Eigenschaft N1 gilt), während schlüsselabhängige Verfahren i.d.R. eine unbegrenzte Anzahl an Challenge-Response-Verifikationen gestatten (Eigenschaft N1 gilt nicht). Hierbei bestehen enge theoretische Beziehungen zu dem verwandten Konzept des beweisbaren Datenbesitzes (*provable data possession* PDP [ABC$^+$07]). Auch diese Verfahren existieren in statischen und dynamischen Ausprägungen ([ADPMT08, EKPT09]), werden im Allgemeinen jedoch als schwächer als PoR angesehen, da der Wissensextraktor bei einem PDP Verfahren nicht gefordert wird (obgleich dieser implizit im Rahmen der Sicherheitsbeweise dennoch oft zu finden ist). Die genauen Verbindungen zwischen PoR und

PDP sind aktuell noch Gegenstand laufender und intensiver Untersuchungen, und werden im Weiteren nicht näher diskutiert.

Verwandte Arbeiten: Das erste als solches bezeichnete PoR-Verfahren wurde in [JK07] vorgeschlagen, wobei die Grundideen bereits in [LEB+03] skizziert wurden. Insbesondere Eigenschaft N1 motivierte Vorschläge schlüsselabhängiger Verfahren [SW08, XC12], welche eine unbegrenzte Anzahl an Verifikationszyklen gestatten. Die Arbeit von [PSU12] zeigt eine theoretische Äquivalenz zwischen PoR-Verfahren und fehlerkorrigierenden Codes auf, welche als Bausteine fast aller PoR-Verfahren zum Einsatz kommen, insbesondere da der Angreifer als Kanal mit Rauschen modelliert werden kann (etwa in [BJO09]). Das nachfolgend angegebene Verfahren ist berechenmäßig sicher; informationstheoretisch sichere Verfahren wurden u.a. in [DVW09] vorgeschlagen. Dynamische Varianten von PoR-Verfahren sind in [ZX11, CKW12, CC12] zu finden, wobei hier insbesondere weitere Sicherheitsanforderungen eingeführt werden, wie etwa Robustheit (Wiederherstellbarkeit auch im Lichte umfangreicher Modifikationen der Daten) oder Fairness (Sicherheit des Servers gegen unrechtmäßigen Beschuldigungen des Clients, dass die Daten manipuliert wurden). An dieser Stelle sei insbesondere auf die Arbeit von [WWR+11] hingewiesen, welcher Aktualisierungen der Daten mit Hilfe von Merkle-Hashbäumen realisiert. Wir werden ähnliche Techniken anwenden, sind jedoch flexibler als [WWR+11] im Hinblick auf die verwendbaren Krypto-Verfahren. Da hier beschriebene Technik ist in weiten Teilen generisch und kann mit verschiedenen Verfahren instantiiert werden. Neben PDP existieren auch die verwandten Konzepte des *proof of storage* [AKK09] und des *proof of ownership* [HHPSP11], auf welche wir nicht näher eingehen.

Beiträge und Konstruktion: Aktuelle PoR-Verfahren können im Allgemeinen auf Basis von Stichprobenüberprüfungen oder unter Ausnützung von Homomorphie-Eigenschaften [LC11] konstruiert werden. Bei einer stichprobenbasierten Überprüfung bettet der Client Prüfdaten, sog. *Marker* (engl. *sentinels*), in die Datei ein, deren Wert später im Rahmen einer PoR-Überprüfung abgefragt werden kann. Diese PoR-Instanzierungen sind im Allgemeinen schlüssellos und unterscheiden sich im Wesentlichen in der Art und Weise, wie diese Marker in die Datei eingebettet werden, um deren Position und Inhalt vor einem nicht vertrauenswürdigen Server zu verbergen. PoR-Instanzierungen auf Basis von Homomorphie-Eigenschaften von digitalen Signaturen oder MACs sind schlüsselabhängig und erfordern häufig die Verarbeitung der gesamten Datenmenge zur korrekten Beantwortung der Anfrage. Der Vorteil einer unbeschränkten Anzahl von Verifikationszyklen wird somit durch erhöhten Rechenaufwand erkauft.

Das in diesem Beitrag vorgestellte Verfahren fällt nicht direkt in eine dieser beiden Kategorien. Die Konstruktion basiert auf Stichproben-Überprüfungen, bettet jedoch keine Verifikationsdaten in die Datei ein, und gestattet – anders als andere Vertreter dieser Kategorie – auch ein nachträgliches Erweitern der lokalen Verifikationsdaten beim Client.

Die nachfolgend beschriebene Konstruktion verfolgt im Wesentlichen die bereits in Abschnitt 1 skizzierte Idee. Der Client sendet dem Server eine Anfrage (challenge) c, welche eine bestimmte Modifikation der Daten F am Server beschreibt. Der Server antwor-

tet mit dem Hashwert der durch c modifizierten Daten F', welche der Client mit seinen lokal gespeicherten Verifikationsdaten vergleicht. Zur Effizienzsteigerung berechnet der Server den Hashwert der gesamten Daten mit Hilfe eines Merkle-Hashbaumes (Eigenschaft N2 fällt somit als Nachteil weg). Modifikationen an den Daten F sind für den Client nur dann möglich, wenn diese den Hashwert des betreffenden Datensatz nicht verändern. Dies kann durch Einsatz von Chameleon-Hashfunktionen realisiert werden (Einschränkung/Eigenschaft N3 fällt weg), welche insbesondere dann auch erneute Abfragen des Datensatzes unter anderen Modifikationen gestatten. Das Verfahren ermöglicht damit eine unbegrenzte Anzahl an Modifikationen der Datei F, und aus diesem Grund auch eine quasi unbegrenzte Anzahl an Verifikationszyklen (Eigenschaft/Nachteil N1 ist somit hinfällig).

Abschnitt 2 stellt die im Weiteren benötigten Konzepte zur Verfügung, bevor wir uns der detaillierten Konstruktion in Abschnitt 3 widmen.

2 Definitionen

Wir nennen eine Funktion $\nu(t)$ *vernachlässigbar*, wenn $\nu(t) < 1/|Q(t)|$ für jedes Polynom Q und alle hinreichend großen $t \in \mathbb{N}$. Eine Wahrscheinlichkeit p nennen wir *überwältigend*, wenn $1 - p$ vernachlässigbar ist. Die Notation $x\|y$ stellt eine Codierung von x und y als String dar, aus welcher x und y eindeutig rekonstruierbar sind. Wir nehmen o.B.d.A. an, dass eine Datei $F = x_1\|x_2\|\ldots\|x_n$ als Folge von Datensätzen, *Blöcken*, x_i, $i = 1, 2, \ldots, n$, angesehen werden kann. An dieser Stelle sei darauf hingewiesen, dass das nachfolgend beschriebene Verfahren sich in kanonischer Weise auf baumstrukturierte Daten (etwa XML-Dateien) übertragen lässt, und – anders als andere PoR-Verfahren – keine festgelegte Datenstruktur voraussetzt. Insbesondere kann die Partitionierung der Daten F in Blöcke beliebig und sinngemäß, etwa durch ein XML-Schema, festgelegt werden.

2.1 Struktur eines PoR-Verfahrens

Wir verwenden die für statische PoR typische Definition [JK07], welche wir entsprechend um Funktionalität für Datenaktualisierungen erweitern. Ein proof of retrievability besteht (im Allgemeinen) aus folgenden Komponenten:

`Setup`(t)**:** Ein probabilistischer Algorithmus, welcher mit einem Sicherheitsparameter $t \in \mathbb{N}$ sämtliche beteiligten kryptographischen Systeme initialisiert und die erforderlichen Schlüssel und Systemparameter zurückgibt. Diese werden allen nachfolgenden Algorithmen zur Verfügung gestellt, und in der Auflistung der Parameter nicht mehr explizit genannt.

`Encode`(F)**:** Algorithmus, welcher die Daten F entgegennimmt, und in einer (fehlerkorrigierenden) Art und Weise codiert, sodass spätere Challenge-Response-Verifikationszyklen ermöglicht werden. Im Allgemeinen können sowohl der Client als auch der

Server eine Fehlerkorrektur durchführen (um etwa Übertragungsfehlern entgegenzuwirken), wobei wir nachfolgend davon ausgehen, dass die Fehlerkorrektur implizit geschieht und wir hierauf nicht weiter eingehen werden. Das Ergebnis von `Encode` ist die codierte Datei F^* und die Information β, aus welcher sich die späteren Anfragen und Verifikationsdaten ableiten lassen.

`Challenge`(β)**:** Der Algorithmus $\mathcal{V}_{\text{chal}}$ erzeugt aus den Daten β eine Anfrage (challenge) c.

`Response`(c, β)**:** Erzeugt aus der Anfrage c die zugehörige Antwort r.

`Verify`(c, r, β)**:** Der Algorithmus $\mathcal{V}_{\text{verify}}$ überprüft ein gegebenes Challenge-Response-Paar (c, r) auf Korrektheit, und liefert entweder 1 (korrekt) oder 0 (inkorrekt).

`Update`: Das interaktive Protokoll $\mathcal{V}_{\text{upd}}$ zwischen dem Client und dem Server ersetzt einen durch den Index i adressierten Datensatz x_i mit einem neuen Wert $\widetilde{x}_i$. Gleichzeitig werden die Daten β beim Client aktualisiert.

`Extract`(β)**:** Algorithmus, welcher eine Sequenz von Anfragen (challenges) erzeugt, aus welchen die Daten F' rekonstruiert werden (entspricht dem Download der gesamten Datenmenge).

Angreifer und Sicherheitsmodell: Wir verwenden das in der Originalarbeit [JK07] vorgeschlagene Sicherheitsmodell in einer geeigneten Erweiterung. Der *Angreifer* $\mathcal{A}$ ist ein Paar von probabilistischen Algorithmen $\mathcal{A}_{\mathsf{setup}}, \mathcal{A}_{\mathsf{resp}}$. Algorithmus $\mathcal{A}_{\mathsf{setup}}$ interagiert mit dem Client (Verifizierer) $\mathcal{V}$ um das PoR-Protokoll zu initialisieren. Insbesondere erzeugt der Angreifer (Server) eine Datei F^*, welche er lokal speichert und dem Verifizierer (Client) die für die Verifikation benötigten Daten β und PoR-Systemparameter α zur Verfügung stellt. Während eines Challenge-Response-Zyklusses berechnet der Angreifer mit dem Algorithmus $\mathcal{A}_{\text{resp}}$ die erforderlichen Antworten. Das Verfahren wird durch eine Ausführung von `Extract` beendet, wobei der Client die Datei F' rekonstruiert. Wir betrachten einen Angriff als erfolgreich, wenn der Client $F' \neq F^*$ erhält.

Es bezeichne $t \in \mathbb{N}$ den Sicherheitsparameter und α die Systemparameter. Die Schreibweise $\mathcal{A}^{\mathcal{B}}$ notiert Orakel-Zugriff (bzw. Interaktion) des Algorithmus $\mathcal{A}$ auf (bzw. mit) Algorithmus $\mathcal{B}$. Mit $x \in_R X$ bezeichnen wir das gleichverteilt zufällige Ziehen eines Elements $x \in X$. Die vorhin beschriebenen Abläufe sind als folgende Experimente formalisierbar:

Experiment $\mathbf{Exp}^{\mathcal{A}}_{\mathsf{setup}}(t)$	Experiment $\mathbf{Exp}^{\mathcal{A}}_{\mathsf{chal}}(F^*, \alpha)$
$\kappa \leftarrow \mathtt{KeyGen}(t)$	$\mathsf{action} \leftarrow_R \{\mathsf{chal}, \mathsf{upd}\}$
$(F^*, \alpha) \leftarrow \mathcal{A}^{\mathcal{V}}_{\mathsf{setup}}$	$c \leftarrow \mathcal{V}_{\mathsf{action}}(\alpha)$
übergebe α an $\mathcal{V}$	$r \leftarrow \mathcal{A}_{\mathsf{resp}}(F^*, \alpha, c)$
	return $\mathcal{V}_{\mathsf{verify}}(r, \alpha)$

Wir betrachten ein PoR-Protokoll als *sicher*, wenn ein Angreifer, welcher in Experiment $\mathbf{Exp}^{\mathcal{A}}_{\mathsf{chal}}(F^*, \alpha)$ mit überwältigender Wahrscheinlichkeit $\geq 1 - \zeta$ erfolgreich ist, den Verifizierer *nicht* dazu bringen kann, etwas anderes als F^* herunterzuladen (durch `Extract`).

Anders ausgedrückt soll Erfolg im Experiment $\mathbf{Exp}^{\mathcal{A}}_{\mathsf{chal}}(F^*, \alpha)$ (also das korrekte Antworten auf Challenges) auch nachfolgend die Rekonstruktion der korrekten Daten F^* implizieren. Die Erfolgswahrscheinlichkeit in $\mathbf{Exp}^{\mathcal{A}}_{\mathsf{chal}}(F^*, \alpha)$ wird bezeichnet mit

$$\mathbf{Succ}^{\mathcal{A}}_{\mathsf{chal}}(F^*, \alpha) := \Pr\left[\mathbf{Exp}^{\mathcal{A}}_{\mathsf{chal}}(F^*, \alpha) = 1\right].$$

Das Sicherheitsmodell ist nun wie folgt festgelegt: Der Angreifer $\mathcal{A}$ besitze die Datei F^*, welche im Rahmen von $\mathbf{Exp}^{\mathcal{A}}_{\mathsf{setup}}(t)$ erzeugt wurde. Der Client $\mathcal{V}$ interagiert mit $\mathcal{A}_{\mathsf{resp}}$ und führt Challenge-Response-Zyklen durch, mit einem abschließenden Aufruf von `Extract`. Der Angriff gilt als erfolgreich, wenn der Client $F \neq F^*$ als Ergebnis erhält. Die Wahrscheinlichkeit, dass dies *nicht* passiert, ist

$$\mathbf{Succ}^{\mathcal{A}}_{\mathsf{extract}}(F^*, \alpha) := \Pr\left[F = F^* | F \leftarrow \mathtt{extract}^{\mathcal{A}_{\mathsf{resp}}}(\alpha)\right].$$

Definition 2.1 *Wir nennen ein PoR-Verfahren (ρ, λ)-sicher, wenn eine im Sicherheitsparameter t vernachlässigbare Funktion $\zeta(t)$ existiert, sodass*

$$\Pr\left[\begin{array}{c|l} \mathbf{Succ}^{\mathcal{A}}_{\mathsf{chal}}(F^*, \alpha) \geq \lambda, & (F^*, \alpha) \leftarrow \mathbf{Exp}^{\mathcal{A}}_{\mathsf{setup}}(t), \\ \mathbf{Succ}^{\mathcal{A}}_{\mathsf{extract}}(F^*, \alpha) < 1 - \zeta & F \leftarrow \mathtt{extract}^{\mathcal{A}_{\mathsf{resp}}}(\alpha) \end{array}\right] \leq \rho.$$

Man beachte, dass die Parameter ρ und λ gegenläufig sind. Wir sind an Verfahren mit *großen* Werten für λ und *kleinen* Werten für ρ interessiert. In solchen Fällen gilt: mit hoher Wahrscheinlichkeit ($> 1 - \rho$) können die Daten F korrekt heruntergeladen werden, oder andernfalls wird die Manipulation durch den Server im Zuge eines Challenge-Response-Verifikationszyklusses mit hoher Wahrscheinlichkeit ($\geq \lambda$) entdeckt.

2.2 Merkle-Hashbäume und Chameleon-Hashing

Es seien die Daten (die Datei) $F = x_1 \| \ldots \| x_n$ gegeben. Eine Änderung an einem einzelnen Datensatz x_i der Datei erfordert i.d.R. das erneute datensatzweise Hashen von F, was im Allgemeinen $O(n)$ Aufrufe einer (kryptographischen) Hashfunktion H erfordert. Ein Merkle-Hashbaum ist eine Methode, um das Hashing so zu organisieren, dass eine derartige Änderung in $O(\log n)$ Aufrufen von H gelingt. Hierzu ordnet man die Blöcke von F als Blattknoten in einem binären Baum mit dem Wurzelknoten r^* an, und führt das Hashing rekursiv beginnend bei den Blättern durch. Der Hashwert eines Blattknotens x ist dabei festgelegt als $H(x)$, wobei x die im Knoten enthaltenen Daten repräsentiert. Der Hashwert eines inneren Knotens v mit den Kindknoten x, y ist definiert als $H(v) := H(H(x) \| H(y))$. Der Hashwert von F ist der Hashwert r^* des Wurzelknotens. Man beachte insbesondere, dass bei einer Änderung des Datensatzes $x_i \leftarrow \widetilde{x}_i$, eine Aktualisierung von r^* lediglich die Bekanntgabe von $O(\log n)$ vielen Hashwerten für die Geschwisterknoten entlang des Pfades vom Blatt x_i zur Wurzel (*Authentifizierungspfad*) erfordert.

Chameleon-Hashing: Eine *Chameleon-Hashfunktion* (auch *trapdoor commitment* genannt), ist eine (im Allgemeinen kollisionresistente) kryptographische Hashfunktion, für welche Kollisionen unter Zuhilfenahme eines geheimen Schlüssels effizient berechnet werden können. Wir geben hier nur die Definition einer solchen Hashfunktion wieder, und verweisen auf [AdM04] für Details und Sicherheitsbeweise. Eine Chameleon-Hashfunktion CH besteht aus drei Algorithmen:

KeyGen: Ein probabilitischer Algorithmus, welcher aus einem Sicherheitsparameter $t \in \mathbb{N}$ ein öffentliches/privates Schlüsselpaar (pk, sk) erzeugt.

Hash: Ein deterministischer Algorithmus, welcher aus den Eingabedaten x, dem öffentlichen Schlüssel pk und einer Zufallszahl r einen Hashwert $CH_{pk}(x, r)$ berechnet.

Forge: Ein deterministischer Algorithmus, welcher aus dem geheimen Schlüssel sk, einem Urbild (x, r) und dem zugehörigen Hashwert $CH_{pk}(x, r)$ eine Kollision (y, s) erzeugt für die gilt $CH_{pk}(x, r) = CH_{pk}(y, s)$.

Im Bezug auf Sicherheit benötigen wir im Folgenden nur die Kollisions-Resistenz von CH, im Sinne dass $\Pr[CH_{pk}(y, s) = CH_{pk}(x, r) | (y, s) \leftarrow \mathcal{A}(x, r, pk)]$ vernachlässigbar für alle (im Sicherheitsparameter) polynomiell beschränkten Angreifer $\mathcal{A}$ ist. Eine Beispielkonstruktion einer solchen Funktion wurde in [AdM04] angegeben:

KeyGen: Es seien p, q zwei große Primzahlen mit $p = uq + 1$, und g sei ein erzeugendes Element der Untergruppe der quadratischen Reste modulo p. Die Ordnung von g sei q. Setze $sk \in_R \{1, 2, \ldots, q-1\}$ und $pk \leftarrow g^{sk} \bmod p$. Wähle eine (herkömmliche) kryptographisch sichere Hashfunktion $H : \{0,1\}^* \to \{0,1\}^\ell$ mit $\ell \geq \lceil \log_2 p \rceil$.

Hash: Wähle $\rho, \delta \in_R \mathbb{Z}_p$, berechne $e \leftarrow H(m \| \rho)$ und definiere den Chameleon-Hashwert als $CH_{pk}(m, \rho, \delta) := (\rho - (pk^e g^\delta \bmod p)) \bmod q$.

Forge: Sei $C = CH_{pk}(m, \rho, \delta)$ gegeben. Wir wählen ein *beliebiges* $m' \neq m$ und eine Zufallszahl $k \in_R \{1, 2, \ldots, q-1\}$. Setze $\rho' \leftarrow (C + (g^k \bmod p)) \bmod q$, $e' \leftarrow H(m' \| \rho')$ und $\delta' \leftarrow (k - e' \cdot sk) \bmod q$. Die gesuchte Kollision tritt auf bei (m', ρ', δ'), zumal

$$\begin{aligned} CH_{pk}(m', \rho', \delta') &= \rho' - (pk^{e'} g^{\delta'} \bmod p) \bmod q \\ &= C + \left(g^k \bmod p\right) - \left(g^{sk \cdot e'} g^{\delta'} \bmod p\right) \bmod q \\ &= C = CH_{pk}(m, \rho, \delta). \end{aligned}$$

Zur Vereinfachung der Notation verzichten wir nachfolgend auf die explizite Angabe der Randomisierer ρ, δ und des öffentlichen Schlüssels pk, und schreiben kurz $CH(m)$ für $CH_{pk}(m, \rho, \delta)$. Man beachte hierbei insbesondere, dass der Teil m' des zweiten Urbildes (m', ρ', δ') *frei wählbar* ist. Diese Möglichkeit werden wir bei der Konstruktion unseres PoR ausnützen.

3 Konstruktion

Wir beschreiben die Komponenten des PoR-Protokolls einzeln, gemäß der in Abschnitt 2 dargestellten Struktur:

`Setup`: Initialisierung aller kryptographischen Parameter, insbesondere der Chameleon-Hashfunktion.

`Encode`: Wir nehmen an, dass die Datei $F = x_1 \| \dots \| x_n$ bereits fehlerkorrigierend codiert ist. Es sei $S \subseteq \{1, 2, \dots, n\}$ eine Teilmenge von Blöcken, welche wir stichprobenartig im Zuge des PoR überprüfen möchten. Wir erzeugen Paare der Form (c_i, r_i), wobei c_i ein zufälliger Bitstring (beliebiger Länge) ist, und r_i der Wurzel-Hash der mit c_i modifizierten Datei $F' = x_1 \| \dots \| (c_i \| x_i) \| \dots \| x_n$, d.h. wir konkatenieren c_i als Präfix des i-ten Datensatzes von F. Der Hashwert r_i wird gebildet durch Anwendung von CH in den Blattknoten, und Anwendung von H in den inneren Knoten (Merkle-Hashbaum). Man beachte, dass hierdurch die für CH verwendeten Randomisierer auf Seiten des Cloud-Provider geeignet codiert (etwa Base64) in die Blattknoten einzubetten sind. Der Client speichert den Wurzel-Hashwert r^* lokal.

`Challenge`: Wähle $i \in_R \{1, 2, \dots, n\}$. Falls $i \in S$, sende (i, c_i) an den Server, andernfalls sende (i, c) mit einem Zufallsstring c.

`Response`: Der Server antwortet auf die Anfrage (i, c), indem er seinen lokal verwalteten Merkle-Hashbaum gemäß der Modifikation $x_i \leftarrow c \| x_i$ erneut erstellt (unter Verwendung von CH_{pk} in den Blattknoten). Er antwortet mit dem neuen Wurzelhashwert r', sowie dem Datum x_i einschließlich der zugehörigen Randomisierer (für CH). Diese Aktualisierung kann in $O(\log n)$ Aufrufen der Hashfunktion durchgeführt werden, und erfordert auch keinen zusätzlichen Speicherplatz, wenn der Merkle-Hashbaum in den zur Datei F gehörigen Index (Suchbaum) integriert wird (von dessen Existenz bei großen Datenmengen oder Datenbanken sinnvollerweise ausgegangen werden kann).

`Verify`: Die Antwort r' des Servers wird genau dann akzeptiert, wenn $(i \in S \wedge r_i = r') \vee (i \notin S)$. Die Menge S stellt hierbei die Menge bekannter Hashwerte dar, welche für die Verifikation der Existenz der Daten vorab beim Client berechnet und abgelegt wurde (anstelle der gesamten Datei F).

`Update`: Soll der Datensatz x_i durch $\widetilde{x}_i$ ersetzt werden, so fordert der Client den i-ten Datensatz an (per `Challenge`), und erhält (x_i, ρ_i, δ_i). Er konstruiert eine Hash-Kollision der Form $(\widetilde{x}_i, \rho'_i, \delta'_i)$ und sendet diese an den Server. Man beachte, dass aufgrund der unveränderten Hashwerte die Konsistenz sowohl der gesamten Daten am Server als auch der gesamten Verifikationsdaten des Clients erhalten bleibt.

`Extract`: Sequentielle Abfrage aller Datenblöcke und Verifikation des Wurzelhashwertes r^* gegen die erhaltenen Daten.

Für die Komplexitäten der einzelnen Operationen sei auf Anhang A, Tabelle 1 verwiesen.

4 Sicherheit

Anders als bei anderen PoR-Verfahren auf Basis von Stichprobenverifikationen erfolgt in dem hier beschriebenen System keine explizite Einbettung von Markern. Es entfallen somit die sonst üblichen Maßnahmen zur Verschleierung der Prüfdaten. Darüber steht dem Client die Möglichkeit offen, weitere Challenge-Response-Paare auf Basis des neuen Datensatzes $\widetilde{x}_i$ zu erzeugen. Da dieser Vorgang ohne Interaktion mit dem Server möglich ist, besitzt der Server keine Möglichkeit, diese neuen und implizit eingebetteten Marker zu entdecken. Gleichwohl muss der Server das Update durchführen (*freshness*-Eigenschaft), da spätere Anfragen genau dieses Datenwort betreffen könnten. Es ist somit eine quasi unbeschränkte Anzahl von Ausführungen des PoR möglich, welche gleichzeitig den Server zur Befolgung des Protokolls nötigen.

Der `Verify`-Algorithmus akzeptiert alle Antworten auf Anfragen betreffend Blöcke außerhalb von S. Dies dient der andauernden Verschleierung der Position von Prüfdaten im Zuge potentiell vieler Ausführungen des PoR, eröffnet jedoch gleichzeitig dem Server auch die Möglichkeit, die Position der Prüfdaten durch falsche Antworten quasi zu "testen". Die Anzahl der Prüfblöcke muss somit hinreichend groß gewählt werden, um dieser Attacke entgegenzuwirken. Der Beweis von Satz 4.1 ist in Appendix B angegeben.

Theorem 4.1 ([Ras13]) *Das beschriebene PoR-Verfahren ist $(\rho, 1-|S|/n)$-sicher gemäß Definition 2.1, wobei ρ vernachlässigbar im Sicherheitsparameter t ist.*

Insbesondere ist auch bei der Wahl der Hashfunktion H (auch als Baustein von CH) Vorsicht geboten: ein in [Ras13] nicht betrachteter Angriff nützt folgende Implementierungvariante aus, bei welcher die Zufallsdaten c als *Suffix* an das Datenwort x_i angehängt werden. In dieser Form erlauben standardisierte Hashfunktionen (wie MD5 [Riv92], SHA-1 [EJ01] und RIPEMD160 [DBP96]), eine Vorausberechnung des Hashwertes unter Verwendung aller Blöcke von x_i, bis zu dem Punkt, an dem die Daten c verarbeitet werden würden. Abbildung 1 zeigt das Prinzip einer iterativen Hashfunktion nach der Merkle-Damgård-Konstruktion [Mer89, Dam89] und dem bei MD5, SHA-1 und RIPEMD verwendeten Padding (Auffüllen des letzten 512 Bit Blocks mit $10\ldots0$ gefolgt von der Länge der zu hashenden Nachricht codiert als 64 Bit Wert). Da die Challenge c erst den Block b_{r-1} verändert, muss der Server in diesem Szenario nur das Zwischenergebnis h_{r-2} und den unvollständigen Block b_{r-1} speichern. Die übrigen Nutzdaten x_i – welche im Allgemeinen wesentlich mehr Speicherplatz beanspruchen – können verworfen werden. Die Werte $H(x_i\|c)$ können für beliebiges c jedoch weiterhin korrekt ermittelt werden.

Aus diesem Grund ist bei einer realen Implementierung, die Anfrage c stets als *Präfix* an das Datenwort zu konkatenieren. Inwieweit dieser Umstand zu Sicherheitslücken bei anderen Verfahren führen kann, welche ähnliche Konstruktionen verwenden (etwa auch die Funktion CH) ist eine aktuell offene Forschungsfrage. Für alternative Möglichkeiten um Hashfunktionen schlüsselabhängig zu machen, d.h. in sichere MACs umzuformen, sei hier außerdem auf [BCK96, Bel06] verwiesen. *Anmerkung:* Auch der neue Hash-Standard Keccak [BDPVA09] folgt in der "Absorptionsphase" ebenfalls dem Prinzip, wiederholt ein Zwischenergebnis mit Blöcken der zu hashenden Nachricht zu "mischen". D.h. gleiche Präfixe bei zu hashenden Nachrichten führen zu gleichen Zwischenergebnissen. Der

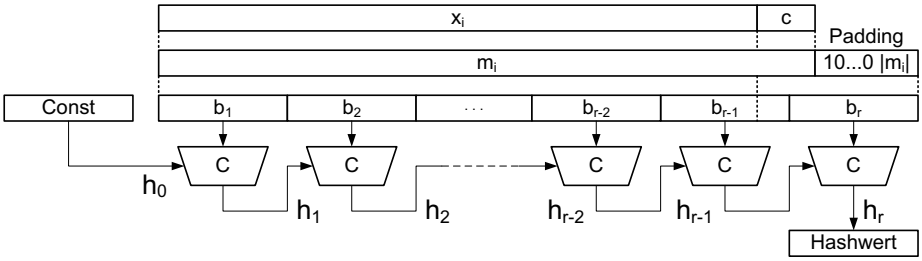

Abbildung 1: Hashen von $x_i \| c$: Merkle-Damgård-Konstruktion inkl. Padding

bei Keccak genutzte Sicherheitsparameter r, welcher die Anzahl der Nachrichten-Bits festlegt, die pro Iteration verarbeitet werden, könnte Teil der Challenge sein. Eine Vorausberechnung ist nun aber immer noch möglich, sie benötige nur mehr Speicherplatz (da die Ergebnisse der Vorausberechnungen für jeden Wert von r gespeichert werden müssen). Die Forderung nach *fairness* stellt Sicherheit für den Server her, im Sinne der Aufklärbarkeit von Anschuldigungen des Clients, dass die Daten modifiziert wurden. Auf technischer Ebene kann dies durch eine signierte Kette von Revisionen der Daten erreicht werden, welche ein gutes Cloud-Storage-System ohnedies durchführen sollte.

5 Resümee

Varianten und Erweiterungen: Eine Verallgemeinerung des Merkle-Hashbaumes von einer strikt binären Baumstruktur auf eine allgemeine Baumstruktur wie jene von XML-Dokumenten ist einfach. Da der Verzweigungsgrad des Hashbaumes für die Sicherheitsargumente keine Rolle spielt, bleiben die Sicherheitseigenschaften (insbesondere Theorem 4.1) gültig und erhalten. Allgemeinere Modifikationen wie das Einfügen oder Löschen von Datensätzen, oder auch strukturelle Änderungen der Daten bedürfen komplexerer Datenstrukturen als Merkle-Hashbäume. Dies ist Gegenstand aktueller Forschung. Im Hinblick auf den Schutz der Privatsphäre auch bei der Abfrage von Daten, kann sowohl eine Verschlüsselung der Datensätze vorgesehen werden, als auch eine Abfrage von wenigstens k Blöcken in jedem Schritt, von denen nur ein einziger wirklich von Interesse ist. In diesem Fall bleibt eine Rest-Unsicherheit von $\Theta(\log k)$ Bit für den Server, welcher aus k Datensätzen tatsächlich abgefragt wurde. Elegantere Methoden, welche den gleichen Effekt zum Ziel haben, bieten Techniken des *private information retrieval* [Gas04], auf welche wir hier nicht eingehen.

Ausblick: Dynamische PoR-Verfahren genießen aktuell großes Interesse seitens der wissenschaftlichen Gemeinschaft und stellen wichtige Bausteine für die Konstruktion sicherer Cloud-Storage-Dienste dar. Die vorliegende Arbeit zeigt eine einfache Konstruktion solcher dynamischen Verfahren auf Basis einfacher kryptographischer Standard-Bausteine. Praktische Implementierungen zum Zwecke der Messung von Effizienz und Aufwand sowohl für den Client als auch für den Server, sind Gegenstand laufender Betrachtungen.

Literatur

[ABC+07] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson und D. Song. Provable data possession at untrusted stores. In *CCS*, Seiten 598–609. ACM, 2007.

[AdM04] G. Ateniese und B. de Medeiros. On the key exposure problem in chameleon hashes. In *SCN*, Seiten 165–179. Springer, 2004.

[ADPMT08] G. Ateniese, R. Di Pietro, L. V. Mancini und G. Tsudik. Scalable and efficient provable data possession. In *SecureComm*, Seiten 9:1–9:10. ACM, 2008.

[AKK09] G. Ateniese, S. Kamara und J. Katz. Proofs of Storage from Homomorphic Identification Protocols. In *ASIACRYPT*, Seiten 319–333. Springer, 2009.

[BCK96] Mihir Bellare, Ran Canetti und Hugo Krawczyk. Keying Hash Functions for Message Authentication. In Neal Koblitz, Hrsg., *CRYPTO*, Jgg. 1109 of *LNCS*, Seiten 1–15. Springer, 1996.

[BDPVA09] G. Bertoni, J. Daemen, M. Peeters und G. Van Assche. Keccak specifications. online: http://keccak.noekeon.org/, 2009.

[Bel06] Mihir Bellare. New Proofs for NMAC and HMAC: Security Without Collision-Resistance. In Cynthia Dwork, Hrsg., *Advances in Cryptology – CRYPTO*, Jgg. 4117 of *Lecture Notes in Computer Science*, Seiten 602–619. Springer Berlin Heidelberg, 2006.

[BJO09] K. D. Bowers, A. Juels und A. Oprea. HAIL: a high-availability and integrity layer for cloud storage. In *CCS*, Seiten 187–198, 2009.

[CC12] B. Chen und R. Curtmola. Robust Dynamic Provable Data Possession. In *ICDCS Workshops*, Seiten 515–525. IEEE Computer Society, 2012.

[CKW12] D. Cash, A. Küpçü und D. Wichs. Dynamic Proofs of Retrievability via Oblivious RAM. In *IACR Cryptology ePrint Archive*, 2012. Report 2012/550.

[Dam89] I. Damgaard. A Design Principle for Hash Functions. In Gilles Brassard, Hrsg., *CRYPTO*, Jgg. 435 of *LNCS*, Seiten 416–427. Springer, 1989.

[DBP96] H. Dobbertin, A. Bosselaers und B. Preneel. RIPEMD-160: A Strengthened Version of RIPEMD. In D. Gollmann, Hrsg., *FSE*, Jgg. 1039 of *LNCS*, Seiten 71–82. Springer, 1996.

[DVW09] Y. Dodis, S. Vadhan und D. Wichs. Proofs of Retrievability via Hardness Amplification. In *TCC*, Seiten 109–127. Springer, 2009.

[EJ01] D. Eastlake und P. Jones. RFC RFC 3174: US Secure Hash Algorithm 1 (SHA1), 2001.

[EKPT09] C. Erway, A. Küpçü, C. Papamanthou und R. Tamassia. Dynamic provable data possession. In *CCS*, Seiten 213–222. ACM, 2009.

[Gas04] W. Gasarch. A Survey on Private Information Retrieval. *Bulletin of the EATCS*, 82:72–107, 2004.

[HHPSP11] S. Halevi, D. Harnik, B. Pinkas und A. Shulman-Peleg. Proofs of ownership in remote storage systems. In *CCS*, Seiten 491–500. ACM, 2011.

[JK07] A. Juels und B. Kaliski. PORs: Proofs of Retrievability for Large Files. In *CCS*, Seiten 584–597. ACM, 2007.

[LC11] S. Liu und K. Chen. Homomorphic Linear Authentication Schemes for Proofs of Retrievability. In *INCOS*, Seiten 258–262. IEEE Computer Society, 2011.

[LEB+03] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows und M. Isard. A cooperative internet backup scheme. In *USENIX ATEC*, Seiten 29–41. USENIX Association, 2003.

[Mer89] R. C. Merkle. One Way Hash Functions and DES. In G. Brassard, Hrsg., *CRYPTO*, Jgg. 435 of *LNCS*, Seiten 428–446. Springer, 1989.

[PSU12] M. Paterson, D. Stinson und J. Upadhyay. A coding theory foundation for the analysis of general unconditionally secure proof-of-retrievability schemes for cloud storage. *CoRR*, abs/1210.7756, 2012.

[Ras13] S. Rass. Dynamic Proofs of Retrievability from Chameleon-Hashes. In *SECRYPT '13*, Seiten 296–304. SciTePress, 2013.

[Riv92] R. Rivest. RFC 1321: The MD5 Message-Digest Algorithm, 1992.

[SW08] H. Shacham und B. Waters. Compact Proofs of Retrievability. In *ASIACRYPT*, Jgg. 5350 of *LNCS*, Seiten 90–107. Springer, 2008.

[WWR+11] Q. Wang, C. Wang, K. Ren, W. Lou und J. Li. Enabling Public Auditability and Data Dynamics for Storage Security in Cloud Computing. *IEEE Trans. on Parallel and Distributed Systems*, 22(5):847–859, 2011.

[XC12] J. Xu und E.-C. Chang. Towards efficient proofs of retrievability. In *ASIACCS*, Seiten 79–80. ACM, 2012.

[ZX11] Q. Zheng und S. Xu. Fair and dynamic proofs of retrievability. In *CODASPY*, Seiten 237–248. ACM, 2011.

A Komplexitäten

Wir geben die asymptotischen Komplexitäten der einzelnen Verfahren hier in Abhängigkeit der Dateigröße n, gemessen in Blöcken, an. Hierbei wird eine Hash-Operation (egal ob Chameleon- oder herkömmlich) mit einem konstanten Aufwand $O(1)$ angenommen. Die vorgestellte Konstruktion ergibt die in Tabelle 1 angegebenen Komplexitäten [Ras13], wobei hier der Aufwand für fehlerkorrigierende Codierungen nicht berücksichtigt ist. Eine Implementierung im Rahmen derer Benchmarks durchgeführt und empirische Laufzeiten ermittelt werden ist Gegenstand laufender Aktivitäten.

Operation	berechenmäßige Komplexität Verifier (Client)	Prover (Server)
Encode	$O(n \log n)$	$O(n \log n)$
Challenge	$O(1)$	–
Response	–	$O(\log n)$
Verify	$O(\log n)$	–
Update	$O(1)$	$O(1)$
neue Challenge	$O(\log n)$	–
Extract	$O(n \log n)$	$O(n \log n)$

Tabelle 1: Komplexitäten

B Beweis von Satz 4.1

Wir betrachten die Ereignisse

$$A := \left\{ \mathbf{Succ}^{\mathcal{A}}_{\mathsf{extract}}(F^*, \alpha) < 1 - \zeta \,\middle|\, [(F^*, \alpha) \leftarrow \mathbf{Exp}^{\mathcal{A}}_{\mathsf{setup}}(t)] \wedge [F \leftarrow \mathtt{extract}^{\mathcal{A}_{\mathsf{resp}}}(\alpha)] \right\},$$
$$B := \left\{ \mathbf{Succ}^{\mathcal{A}}_{\mathsf{chal}}(F^*, \alpha) \geq \lambda \,\middle|\, [(F^*, \alpha) \leftarrow \mathbf{Exp}^{\mathcal{A}}_{\mathsf{setup}}(t)] \wedge [F \leftarrow \mathtt{extract}^{\mathcal{A}_{\mathsf{resp}}}(\alpha)] \right\},$$

und zeigen $\Pr[\neg A \cup \neg B] \geq 1 - \zeta$. Hieraus folgt $\Pr[A \cap B] = \rho = \mathrm{negl}(t)$, und damit die Sicherheit des Verfahrens gemäß Definition 2.1.

Das Ereignis $\neg A$ tritt genau dann ein, wenn der Verifizierer $F' = F^*$ mit überwältigender Wahrscheinlichkeit rekonstruiert. Nach Konstruktion überprüft `extract` ob $CH_{pk}(F^*) = r^*$ gilt, wobei r^* der beim Client vorab gespeicherte Wurzel-Hash der Original-Datei ist (welcher über eine beliebige Folge von Updates unverändert bleibt). Somit ist das Ereignis der irrtümlichen Akzeptanz äquivalent zum Ereignis einer Hash-Kollision $CH_{pk}(F') = CH_{pk}(F^*)$, dessen Wahrscheinlichkeit für eine kryptographisch sichere Hashfunktion als vernachlässigbar angenommen werden kann. Hieraus folgt $\Pr[\neg A] \geq 1 - \mathrm{negl}(t)$.

Für das Ereignis $\neg B$ erinnern wir daran, dass der Angreifer mit Wahrscheinlichkeit von $\geq \lambda = 1 - |S| / |F|$ Fällen korrekt antworten kann (in diesen Fällen ist beim Client keine korrekte Antwort gespeichert, sodass der Client immer akzeptieren wird). In diesen Fällen gilt $\Pr[\neg B] = 0$.

Nach dem Satz von Sklår gilt $\Pr[\neg A \cap \neg B] = C(\Pr[\neq A], \Pr[\neg B])$ für eine Copula-Funktion C, welche die obere Frechet-Hoeffding-Ungleichung erfüllt, nämlich $C(x, y) \leq \min\{x, y\}$. Hieraus folgt $\Pr[\neg A \cap \neg B] \leq \min\{\Pr[\neg A], \Pr[\neg B]\} = 0$. Dies schließt den Beweis ab, da $\Pr[\neg A \cup \neg B] \geq 1 - \mathrm{negl}(t) + 0 - 0$ und demzufolge $\Pr[A \cap B] = 1 - \Pr[\neg A \cup \neg B] \leq \mathrm{negl}(t)$.