Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Studienarbeit Nr. 2454

Secure Delete Object Store - Sicheres Löschen auf nicht vertrauenswürdigen Speichersystemen

Mathias Mormul

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. -Ing. habil. Bernhard Mitschang
Betreuer/in:	Dipl.-Inf. Tim Waizenegger
Beginn am:	16. März 2014
Beendet am:	16. September 2014
CR-Nummer:	E.1, E.3

Kurzfassung

Durch neue Technologien wie Solid State Drives und Cloud Computing hat die Datenhaltung in den letzten Jahren einen Paradigmenwechsel erlebt. Beide Technologien führen dazu, dass der Benutzer sich im Unklaren über die Lokalität und die Anzahl von Kopien seiner ist. Herkömmliche Verfahren zur sicheren Datenlöschung wie das Überschreiben der Daten funktionieren daher nicht mehr. Dennoch muss weiterhin garantiert werden können, dass die Daten des Benutzers bei Bedarf sicher gelöscht werden können. Diese Arbeit präsentiert einen Secure Delete Object Store, welcher das sichere Löschen auf nicht vertrauenswürdigen Speichersystemen mittels Verschlüsselung garantiert. Jede Datei des Benutzers wird mit einem exklusiven Schlüssel verschlüsselt, sodass bei einem Löschvorgang nur der entsprechende Schlüssel gelöscht werden muss. Es wird Wert darauf gelegt, eine Datenstruktur zu schaffen, die erweiterbar ist auch bei großen Datenmengen eine effiziente Lösung für die Verwaltung der Schlüssel darstellt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die Datenhaltung hat in den letzten Jahren einen Paradigmenwechsel erlebt. Daten wurden meist in eigenen Rechenzentren auf magnetischen Festplatten gespeichert. Die Daten waren damit unter eigener Kontrolle, die Unternehmen waren sich über die vorhandenen Sicherheitsmaßnahmen bewusst und bei Bedarf konnten sie die Daten löschen. Durch das Auftreten neuer Technologien wie den Solid State Drives (SSD) und des Cloud Computing sind die üblichen Methoden zur Datenlöschung obsolet geworden und können das sichere Löschen nicht mehr garantieren.

SSDs sind eine neue Art von Speichermedien, die, im Gegensatz zu magnetischen Festplatten, ihre Daten in Flashspeichern aufbewahren. Magnetische Festplatten bestehen teilweise aus beweglichen Komponenten (Spindel, Scheiben, Schreib-und Leseköpfe,..) und besitzen dadurch mechanische Limitierungen, was Zugriffszeiten sowie Schock- und Temperaturresistenz betrifft. Weitere Nachteile sind ein höherer Energieverbrauch und die durch die Bewegungen entstandene Lautstärke. Aufgrund dieser Nachteile und den stetig sinkenden Preisen der SSDs werden diese höchstwahrscheinlich nach und nach magnetische Festplatten ersetzen.

Konventionelle Methoden zur sicheren Datenlöschung, die bei magnetischen Festplatten Erfolg hatten (ein - bzw. mehrmaliges Überschreiben der Datenträger), sind aufgrund der internen Architektur von SSDs meist nur bedingt erfolgreich. Dem Benutzer erscheinen die Daten als gelöscht, allerdings werden nur die Metadaten, nicht die Daten selbst, gelöscht. Die Daten (und weitere Kopien) sind weiterhin physikalisch vorhanden und können ausgelesen werden. Deshalb müssen neue Methoden entwickelt werden, die diese neue Architektur von SSDs beachten und auch bei diesen Speichern ein sicheres Löschen ermöglichen.

Cloud Computing stellt die Datenschützer vor weitere Herausforderungen. Cloud Computing beschreibt das Konzept, IT-Infrastruktur zu virtualisieren und sie erst dann dem Benutzer zur Verfügung zu stellen. Wird jedem Benutzer ein eigener PC mit exklusiver Hardware zugeteilt, so ist dieser in den seltensten Fällen voll ausgelastet (einerseits durch die Aufgaben des Benutzers an sich, andererseits durch gelegentliche Pausen etc...). Werden diese Kapazitäten aber auf mehrere Benutzer aufgeteilt, ermöglicht es eine effizientere Nutzung der bestehenden Kapazitäten. Dabei kann entweder ein Zugriff auf die virtualisierte Hardware, bereits funktionsfähige Laufzeitumgeben, oder fertige Software gewährt werden.

Zu unterscheiden sind auch die verschiedenen Organisationsformen des Cloud Computing. Es ist möglich, eine exklusive Cloud für eine bestimmte Gruppe von Benutzern zu erstellen (z.B. ein eigene Cloud innerhalb des Firmennetzwerks). Die Unterschiede zu dem herkömmlichen Rechenzentren liegen dabei einzig und allein auf der effizienteren Nutzung der Ressourcen. Das Problem der Datensicherheit ist dasselbe wie zuvor. Zugleich ist es möglich, die Cloud-Dienste eines externen Anbieters zu nutzen. In diesem Fall verschiebt sich das Problem der Datensicherheit von dem Benutzer auf den Anbieter. Der Benutzer hat keinen Zugriff mehr auf die zugrundeliegende Hardware und übergibt

die Kontrolle über den sicheren Umgang und des sicheren Löschens der Daten komplett dem Cloudanbieter. Spätestens bei sensiblen Daten ist ein sicheres Löschen der Daten unbedingt notwendig. Wo die Daten gespeichert sind (Amazon EC2 [ec2]. besitzt z.B. mehrere Standorte weltweit) ist dabei genauso wenig bekannt wie die Anzahl der Kopien, die in der Cloud gespeichert sind.

1.1 Problembeschreibung

Ob SSD oder Cloud Computing - beide Technologien führen dazu, dass neue Methoden zum sicheren Löschen benötigt werden. Bei SSDs ist dies erforderlich, weil herkömmliche Methoden wie das mehrmalige Überschreiben aufgrund der internen Architektur nicht mehr ausreichend sind [WGSS11]. Bei Cloud Computing besitzt der Benutzer keine Kontrolle mehr über die zugrundeliegende Hardware. Die Skepsis ist vor allem bei Unternehmen hoch, die sich kaum mit Cloud Computing beschäftigt haben. Die Sicherheitsstandards werden infrage gestellt und es besteht Angst vor einem Missbrauch der Daten durch Administratoren oder weiteren Drittparteien [ADKR11].

Um dem Benutzer diese Sicherheit bezüglich seiner Daten zu garantieren, wird eine Methode verlangt, die von der Benutzerseite aus funktioniert. Eine dieser Methoden basiert auf dem Prinzip der Verschlüsselung der Daten. Wird eine Datei verschlüsselt, erhält der Benutzer einen generierten Schlüssel zum Dekodieren der Datei. Nur mit diesem Schlüssel ist ab sofort ein Zugriff auf die Datei möglich. Sollte der Benutzer diese Datei nun löschen wollen, so reicht es auch, den dazugehörigen Schlüssel zu löschen, wodurch die kodierte Datei unbrauchbar wird, unabhängig davon, ob sie noch im Flashspeicher einer SSD oder auf Servern von Cloudanbietern zu finden ist.

Im Rahmen dieser Arbeit wird deshalb ein neues System zum sicheren Löschen auf nicht vertrauenswürdigen Speichersystemen implementiert - der Secure Delete Object Store (SDOS), mit dem das Konzept des sicheren Löschens durch Verschlüsselung verfolgt wird. Vorrangig ist dabei die effiziente Verwaltung der Schlüssel auch bei einer großen Menge von Datenbeständen.

1.2 Übersicht

In den folgenden Kapiteln wird näher auf das in der Einleitung beschriebene Problem eingegangen. Kapitel 2 zeigt den derzeitigen Stand der Technik. SSDs und Cloud Computing werden ausführlicher besprochen, darunter speziell OpenStack, auf welchem SDOS getestet wird. In Kapitel 3 werden verwandte Arbeiten vorgestellt, die sicheres Löschen auf nicht vertrauenswürdigen Speichermedien ermöglichen. In Kapitel 4 wird näher auf SDOS eingegangen. Zuerst wird der konzeptionelle Entwurf vorgestellt. Kapitel 5 beschreibt die Implementierung der vorangegangenen Konzepte. Es werden Probleme, die dieser Entwurf in der Anwendung hervorruft, besprochen und mögliche Lösungsansätze sowie Alternativverfahren vorgestellt. In Kapitel 6 werden offene Punkte angeführt sowie weitere Ideen zur Weiterentwicklung von SDOS. Kapitel 7 enthält die Zusammenfassung dieser Arbeit.

2 Stand der Technik

Im Folgenden werden SSDs und Cloud Computing näher betrachtet. Dabei wird erklärt, weshalb SSDs herkömmliche Löschverfahren nicht unterstützen. Des Weiteren werden die verschiedenen Organisationsformen sowie der Cloud Stack des Cloud Computing vorgestellt. Bezüglich Cloud Computing wird zusätzlich OpenStack vorgestellt - eine Cloud Software, die es ermöglicht, eigene Clouds zu erstellen.

2.1 Solid State Drives

Wie in Kapitel 1 bereits erwähnt, ist das Löschen mit herkömmlichen Mitteln bei Solid State Drives nicht mehr möglich. Grund dafür ist die unterschiedliche Architektur von SSDs im Vergleich zu magnetischen Festplatten. SSD verwenden Flashspeicher, welche in pages und blocks unterteilt sind. Ein block besteht aus 64-256 pages. Operationen, die Daten schreiben, verwenden die pages und ändern dabei 1en zu 0en. Eine Löschoperation dagegen verändert den gesamten block und ändert alle 0en zu einer 1. In einem Flashspeicher müssen zuerst die Daten gelöscht werden, bevor er mit neuen Daten beschrieben werden kann. Aufgrund dieser unterschiedlichen Granularität ist deshalb ein in-place update des Sektors vom *logical block adresses* (LBA) nicht möglich. Der *flash translation layer*(FTL) ist für das Mapping zwischen den LBA), welche für den Benutzer sichtbar sind, und den *physical pages* des Flashspeichers zuständig. Der FTL schreibt beim Verändern eines Sektors die gesamten Daten an eine andere Stelle. Aus diesem Grund ist das (mehrmalige) Überschreiben zum sicheren Löschen nicht ausreichend. Aus der Benutzersicht sind die Daten nicht zugreifbar, obwohl sie noch physisch vorhanden sind. Für diese übriggebliebenen Fragmente werden bei SSDs bis zu 25% mehr Speicherkapazität bereitgestellt. In [WGSS11] wurden 1000 Dateien erstellt und die SSD anschließend untersucht, wobei sich herausgestellt hat, dass von manchen Dateien bis zu 16 veraltete Kopien vorhanden waren. Mittels des in den Festplatten integrierten Löschverfahrens Secure Erase soll das sichere Löschen ermöglicht werden. Allerdings ist dieser von den Herstellern teilweise falsch implementiert und ermöglicht auch nur das Löschen der gesamten Festplatte. Das Löschen individueller Dateien ist weiterhin nicht sicher. Auch bei Secure Erase ist es unter Umständen möglich, anschließend Dateien zu rekonstruieren. Die Alternative ist das Zerstören der gesamten Festplatte, was auch den Verlust der gesamten Daten zu Folge hat, was in den meisten Fällen unerwünscht ist [WGSS11].

2.2 Cloud Computing

Das sich Cloud Computing inzwischen zu einem festen Bestandteil der IT-Landschaft herauskristallisiert hat, ist weitestgehend bekannt. Im Folgenden wird ein grober Überblick über den Cloud Stack, die verschiedenen Organisationsformen und die Vor- und Nachteile des Cloud Computing erstellt. Anschließend wird OpenStack vorgestellt - eine Cloud Software, auf welcher SDOS implementiert wird.

2.2.1 Cloud Stack

Der *Cloud Stack*[CSt] bezeichnet die verschiedenen Dienstleistungen, die das Cloud Computing zu bieten hat. Im Folgenden werden diese kurz vorgestellt, wobei die oberen Schichten auf den unteren aufbauen.

- *Software as a Service (SaaS)*
  SaaS bietet dem Kunden vollständige Produkte zum Verwenden an (z.B. ein Monatsabonnement für Photoshop).
- *Platform as a Service (PaaS)*
  Bei PaaS werden Laufzeitumgebungen angeboten, wobei sich der Benutzer nicht um die zugrundeliegende Infrastruktur kümmern muss (z.B. Microsoft Windows Azure).
- *Infrastructure as a Service (IaaS)*
  IaaS stellt die gesamte zugrundeliegende Infrastruktur zur Verfügung, u.a. Prozessorleistung, Speicherplatz und Netzwerke. Externe Anbieter von IaaS sind z.B. Amazon AWS (EC2 Cloud) oder die Rackspace Cloud [rs].

2.2.2 Organisationsformen

Cloud Computing lässt sich in verschiedene Organisationsformen klassifizieren [MG11]. Dazu zählen:

- *Public Cloud*
  Der angebotene Dienst wird öffentlich zur Verfügung gestellt, sodass jede beliebige Person oder Organisation darauf Zugriff erlangen können.
- *Private Cloud*
  Eine Private Cloud dient dazu, den Zugriff auf die Cloud auf eine bestimmte Gruppe zu beschränken (bspw. ein internes Firmennetzwerk).
- *Hybrid Cloud*
  Eine Hybrid Cloud beschreibt eine Mischung der oben genannten Organisationsformen.

Im Kontext dieser Arbeit wird ab sofort nur noch von Public Clouds gesprochen, da sich die sicherheitskritischen Aspekte, für deren Lösung SDOS konzipiert wurde, größtenteils dort auffinden lassen.

2.2.3 Vor- und Nachteile

Im Folgenden werden kurz die Vor- und Nachteile des Cloud Computing erläutert - welche Probleme es gibt, die mit SDOS gelöst werden können und wieso trotz der derzeitig bestehenden Probleme das Cloud Computing weiter wächst. Das *National Institute of Standards and Technology*(NIST[NI]) beschreibt in [MG11] die Charakteristika einer Cloud wie folgt:

- *On-demand self-service*
  Der Benutzer kann selbst bei Bedarf weitere Kapazitäten hinzufügen bzw. entfernen. Damit ist sichergestellt, dass der Benutzer nur so viel zahlt, wie er tatsächlich benötigt.
- *Broad network access*
  Die Cloud ist über ein Netzwerk (Internet) verfügbar. So kann beispielsweise von verschiedenen Standorten auf der Welt auf die selben Daten zugegriffen werden, ohne dass ein Datentransfer stattfinden muss.
- *Resource Pooling*
  Ein Kunde erhält keinen exklusiven Zugriff auf spezielle Ressourcen wie z.B. Rechenleistung (Prozessorzugriff). Stattdessen werden die gesamten Ressourcen des Anbieters unter den Kunden mit einem Multi-Tenant-Modell geteilt und bei Bedarf den Kunden zur Verfügung gestellt. Dies senkt die Kosten, da ansonsten brachliegende Kapazitäten nicht genutzt werden würden.
- *Rapid elasticity*
  Das Hinzufügen und Entfernen von Kapazitäten soll möglichst schnell erfolgen. Einerseits, damit der Kunde bei fehlenden Kapazitäten rechtzeitig neue erhält, und andererseits, damit bei nicht benötigten Kapazitäten keine unnötigen Kosten entstehen. Dem Benutzer erscheinen die Kapazitäten dabei unendlich.
- *Measured service*
  Die Nutzung der Cloud-Dienste kann überwacht und kontrolliert werden, um anschließend eine optimale Ressourcennutzung zu erzielen.

Die Kombination dieser Eigenschaften ermöglicht eine kostengünstige, bedarfsabhängige Bereitstellung von Ressourcen. Viele Unternehmen bleiben dennoch skeptisch. Hier wird vor allem der Sicherheitsaspekt aufgegriffen. Meistens sind die verwendeten Sicherheitsmaßnahmen der Anbieter nicht bekannt und die Unternehmen müssen sich auf zuverlässige Sicherheitsstandards verlassen. Ein Beispiel für potenzielle Gefahren stellt beispielsweise die Multi-Tenant-Architektur des IaaS-Modells dar. Ein einzelner physikalischer Server kann mehrere virtuelle Maschinen für verschiedene Benutzer bereitstellen. Sollte eine von einem einzelnen Benutzer verwendete Schadsoftware Zugriff auf die zugrundeliegenden Systeme erlangen, so sind die Instanzen anderer Benutzer auch gefährdet. Auch der Missbrauch von Administratoren der Anbieter stellt Gefahren dar, da Daten der Benutzer gelöscht oder gestohlen werden können [ADKR11]. Die Cloud-Anbieter im Gegenzug versichern, dass ihre Sicherheitsmaßnahmen höher sind, als die von durchschnittlichen Unternehmen [CGJ+09]. Die Daten von Unternehmen können darüber hinaus auch ganz legal betrachtet werden. Der sogenannte Patriot Act [McC02] verpflichtet sogar amerikanische Unternehmen dazu, ihre gespeicherten Daten den US-Behörden auf Anfrage auszuhändigen. Da die größten Anbieter, beispielsweise Amazon, amerikanische Unternehmen sind, stellt dies ein enormes Problem für viele Benutzer dar. Im Kontext

dieser Arbeit ist vor allem wichtig, wann und wie gründlich Daten von Benutzern bei Bedarf gelöscht werden. Wie lange dauert es, bis die Daten gelöscht werden? Sind die Daten und alle weiteren Kopien unwiederbringlich gelöscht? Diese Fragen lassen sich nicht pauschal beantworten. 2009 stellt Amazon für die Elastic Compute Cloud bereits etwa 40000 Server bereit [Geh]. 2012 waren es Berechnungen zufolge bereits etwa 450000[1] Server [Liu12]. Bei dieser Größe sind durchgehend mehrere Server nicht verfügbar. Sollte sich auf einem dieser Server eine Datei befinden, die gelöscht werden soll, kann dies nicht sofort erfolgen. Auch ob die Daten überhaupt gelöscht werden, lässt sich von Benutzerseite nicht überprüfen. Obwohl das sichere Löschen bei Cloud-Anbietern durch Service Level Agreements[2] gewährleistet wird, sind klein- und mittelständische Unternehmen weiterhin verunsichert [HMMG10].

2.2.4 OpenStack

OpenStack [OS], von NASA[Na] und Rackspace[RaS] entwickelt, ist eine Open Source Software zur Erstellung von Public, Private und Hybrid Clouds und wird unter der Apache 2.0 Lizenz [ALV] angeboten. Die NASA entwickelte Nova, die jetzige Compute-Komponente von OpenStack. Zu Beginn der Entwicklung einer Storage-Komponente hat NASA erkannt, dass Rackspace bereits eine Storage-Komponente mit einem ähnlichen Konzept verfolgte, woraufhin 2010 in Kooperation der beiden Organisationen OpenStack entstand [Mor10].

Die 2012 gegründete OpenStack Foundation dient der Verbreitung und Weiterentwicklung von OpenStack und besteht derzeit aus über 9500 Mitgliedern und 850 Unternehmen[OSFoun]. Aufgrund dieser Kollaboration entsteht mit OpenStack möglicherweise eine ubiquitäre Cloud Software, die dringend benötigte Standards im Bereich der IaaS-Software bietet, was sowohl IaaS-Anbietern als auch IaaS-Nutzern dient.

OpenStack weist folgende Komponenten auf [OpS]:

- *OpenStack Compute*
  Die Compute Komponente dient der Bereitstellung und dem Management großer Netzwerke von Instanzen.
- *OpenStack Storage*
  OpenStack Storage ist die Speicherkomponente. Diese wird unterteilt in den Object Storage als allgemeiner Speicher und den Block Storage, welcher als persistenter Speicher für Daten von Instanzen zuständig ist.
- *OpenStack Networking*
  Diese Komponente verwaltet die Netzwerke. Es ermöglicht den erstellten Instanzen eine Netzwerkanbindung, indem es ein Local Area Network erstellt und IPs an die virtuellen Maschinen vergibt.

[1] nur eine Schätzung, da Amazon keine genauen Daten preisgibt
[2] vertragliche Vereinbarung zwischen Kunde und Anbieter

- *OpenStack Dashboard*
  Das Dashboard ist der Zugriffspunkt auf die Cloud und bietet eine grafische Oberfläche, um eine vereinfachte Benutzung zu ermöglichen.
- *OpenStack Image Service*
  Beim Start einer Instanz muss diese zuerst eingerichtet werden (Betriebssystem, Software,...). Ist dies einmal erfolgt, kann ein Image dieser Instanz erzeugt werden und auf weiteren Instanzen geladen werden, ohne die Einrichtung neu erstellen zu müssen.
- *OpenStack Identity Service*
  Der Identity Service wird für die Zugriffsveraltung für alle Benutzer verwendet. Hier werden Authentifizierungen angefertigt und vergeben.

2.3 Verschlüsselung

Das zur Verschlüsselung der Daten genutzte Verschlüsselungsverfahren ist der Rijndael-Algorithmus, welcher vom NIST zum *Advanced Encryption Standard* (AES) erklärt wurde [Pub01]. AES existiert bislang in drei verschiedenen Variationen - dem AES-128, AES-192 und dem AES-256, wobei die Zahlen für die verwendete Schlüssellänge stehen. Nach [hathaway2003national] sind Schlüssel der Längen 192 sowie 256 zugelassen, um in den USA Regierungsdokumente der höchsten Geheimhaltungsstufe zu verschlüsseln.

Der Rijndael-Algorithmus zählt zu den symmetrischen Verschlüsselungsverfahren, d.h. derselbe Schlüssel wird zum Kodieren sowie zum Dekodieren verwendet. Zudem handelt es sich um eine Blockchiffre. Der zu verschlüsselnde Klartext muss also eine bestimmte Länge besitzen und durch die Blockgröße geteilt werden können. Die gewählte Schlüssellänge ist mit der Blockgröße identisch (Bsp.: AES-256 - Anzahl der Zeichen der Datei muss durch 256 teilbar sein). Zweifellos entspricht nicht jede Datei diesen Anforderungen, weshalb der Inhalt der Datei künstlich erhöht werden muss.

Die derzeit beste Kryptoanalyse für AES-128, AES-196 und AES-256 ist der *Biclique Attack* und ermöglicht ein Auffinden des Schlüssels innerhalb von $2^{126,1}$, $2^{189,7}$ und $2^{254,4}$ Schritten [BKR11]. Ein handelsüblicher PC (4 GHz Prozessor) benötigt somit für die Entschlüsselung einer 128 Bit Verschlüsselung 700 Trillionen Jahre. Auch mit neuesten Supercomputern ist es derzeit nicht effizient, diese Verschlüsselung zu dekodieren.

3 Verwandte Arbeiten

Das sichere Löschen von Daten ist bereits ein seit Jahrzehnten wichtiges Thema. Dabei ist das Löschen von Daten auf magnetischen Speichermedien gut erforscht. Löscht ein Benutzer eine Datei, landet diese im Papierkorb. Wird der Papierkorb geleert, erscheint die Datei dem Benutzer als vollständig gelöscht. In Wirklichkeit wird nur der Verweis auf diese Datei gelöscht. Die Stelle, an der diese Datei gespeichert ist, wird als frei markiert und darf bei Bedarf überschrieben werden, was unter Umständen niemals passiert. Die naheliegende Methode ist deshalb, alle Daten zu überschreiben. Dabei ist die Frage, wie oft der Datenträger überschrieben werden sollte, allgemein umstritten. In [KSSL06] hat das NIST bestätigt, dass das einmalige Überschreiben in den meisten Fällen ausreichend ist, um die Daten sicher zu löschen. Regierungsbehörden überschreiben Dateien maximal drei Mal. Bis auf wenige Spezialfälle lassen sich die Daten mit bekannten Methoden nicht rekonstruieren[1]. Methoden zum Überschreiben werden meist mit den Betriebssystemen mitgeliefert. UNIX-Systeme besitzen die *wipe*-Funktion, Mac OS X besitzt *Secure Emtpy Trash*. Für Windows existiert die Funktion *shred*. Wird das Speichermedium nicht mehr benötigt, so ist auch das physikalische Zerstören des Laufwerks eine Möglichkeit, um zukünftigen Zugriff auf die Daten zu verhindern[KSSL06].

Auch das Konzept des sicheren Löschens mittels Verschlüsseln ist bereits seit Langem bekannt. Bereits 1992 ist diese Idee patentiert worden. [Kun93] verfolgt den Ansatz, die Dateien erst dann zu verschlüsseln, wenn der Benutzer den Befehl zum Löschen gibt. Ist der Benutzer sich dessen sicher, so wird der Schlüssel, welcher zum Dekodieren notwendig wäre, nicht gespeichert. Dies erspart die Verwaltung der Schlüssel. Dieser Ansatz ist bei neuen Technologien wie SSDs oder Cloud Computing keine Lösung. Wird die Datei erst beim Befehl des Löschens verschlüsselt, bestehen möglicherweise bereits Dutzende Kopien, auf die der Benutzer keinen Zugriff hat.

Obwohl die meisten dieser Verfahren auf Verschlüsselung basieren, müssen je nach Anwendungsfeld verschiedene Kriterien betrachtet werden. Wie bereits erwähnt, ist aufgrund der Architektur von SSDs die genaue Lokalisation von Daten nicht möglich. Gleichzeitig können mehrere Kopien existieren. In [LHC+08] wird ein auf Verschlüsselung basiertes Dateisystem entwickelt, das sicheres Löschen auf SSDs ermöglicht. Jede Datei wird beim Erstellen bzw. Modifizieren verschlüsselt. Der dabei generierte Schlüssel wird in den Metadaten im Object Header gespeichert. Folglich ist es ausreichend, nur die Metadaten einer Datei zu löschen, um diese sicher zu löschen. Da auch die Metadaten an verschiedenen Stellen gespeichert sein könnten, wurde ein Dateisystem entwickelt, welches erzwingt, dass Metadaten und deren Kopien alle innerhalb des selben Blocks gespeichert werden. Bei einer Löschoperation wird der gesamte Block gelöscht, demnach zufolge alle Metadaten und damit auch alle Schlüssel.

[1] Es könnten auch unbekannte Methoden existieren, mit denen es funktioniert

Cloud Computing erfordert weitere Maßnahmen. Der Benutzer erwartet permanenten Zugriff auf seine Dateien. Um dem nachzukommen, müssen mehrere Kopien auf verschiedenen Servern gespeichert werden. Je größer eine Cloud ist, desto wahrscheinlicher wird es, dass mehrere Datenspeicher zeitweilig ausfallen. Die Architektur einer Cloud sieht vor, mehrere Kopien in verschiedenen Datenzentren zu speichern. Dies erhöht einerseits die Verfügbarkeit und andererseits die Ausfallsicherheit der gespeicherten Daten. Will der Benutzer nun eine Datei löschen, kann er sich nicht über die Löschung aller Kopien sicher sein. Zeitweilig ausgefallene Datenträger, welche die Kopien enthalten, können die gewünschte Datei nicht löschen. Mindestens eine Kopie kann auf diesem Weg über einen längeren Zeitraum trotz des Löschbefehls verfügbar bleiben.

Die hat zur Folge, dass die einzigen Verfahren zum sicheren Löschen in der Cloud auf Verschlüsselung basieren. Nicht nur das sichere Löschen wird damit ermöglicht. Jeder nicht authorisierte Zugriff auf die Benutzerdaten wird verweigert. Dazu zählen externe Angriffe auf die Cloud sowie Missbrauch durch Administratoren. Im Unternehmensbereich wird dadurch Wirtschaftsspionage deutlich erschwert. Wichtig hierbei ist die sichere Verwaltung der Schlüssel. Eine Möglichkeit ist die lokale Speicherung der Schlüssel. Zu Bedenken ist, dass bei Verlust dieses Schlüssels der Zugang zu den Daten verloren ist. Deshalb sollte zur sicheren und einfacheren Verwaltung spezielle Hardware oder Software verwendet werden sowie mindestens ein Backup vorhanden sein.

Das Hardware Security Module (HSM) ist ein Peripheriegerät, welches ein höheres Maß an Sicherheit als Software bietet. Evaluiert werden diese durch Drittparteien wie das NIST, oder dem französischen *Direction Centrale de la Sécurité des Systemés d'Information*[DCS], wodurch eine noch höhere Sicherheit gewährleistet wird. Im Folgenden werden die wichtigsten Aufgaben eines HSM aufgezeigt[Sus11]

- Erzeugung randomisierter Zahlen
- Asymmetrische und symmetrische Schlüsselerzeugung
- Speicherung privater Schlüssel

Durch Verwendung von HSMs wird die Verwaltung der Schlüssel vereinfacht und erleichtert die Verbreitung von sicheren Löschverfahren, die auf Verschlüsselung basieren.

Amazon Web Services (AWS)[AW] bieten das CloudHSM[AWC] an. Der Benutzer kann somit Zugriff auf ein HSM innerhalb der Amazon Cloud und die darin enthaltenen Funktionen erhalten. Nur der Benutzer hat Zugriff auf das HSM. Um höchstmögliche Sicherheit zu garantieren, wird jedem Benutzer eine dedizierte HSM zugewiesen. Entsprechend hoch sind die Kosten, die beim Einsatz entstehen. Neben einer Vorabzahlung von 5000 USD pro HSM belaufen sich die monatlichen Kosten je nach Region auf zwischen 1373 USD und 1635 USD - Kosten, die vor allem kleinständische Unternehmen mit weniger sensiblen Daten selten aufwenden werden wollen. Zudem empfiehlt Amazon die Verwendung mehrerer CloudHSMs, welche in verschiedenen Serverstandorten betrieben werden. Im Falle des Ausfalls eines HSMs hat der Benutzer weiterhin Zugriff auf das zusätzliche HSM. Des Weiteren kann es vorkommen, dass ein Angriff auf ein HSM erfolgt. Das HSM löscht in diesem Fall (falls der Angriff erkannt wird) alle Schlüssel und das zweite HSM dient als Backup. Verwendet werden HSMs von SafeNet [SN], welche die Standards Federal Information Processing Standard (FIPS)140-2[Cad11] und Common Criteria EAL4+ [Sha03] unterstützt. Alternativ kann der Benutzer auch eine eigene HSM kaufen und lokal in Betrieb nehmen. Jede sicherheitskritische Operation innerhalb der Cloud muss in

diesem Fall durch die lokale HSM ausgeführt werden. Die Preise rangieren zwischen unter 1000 und mehreren 1000 USD, je nach Funktionalität und Sicherheitslevel.

4 Konzept

Wie zu sehen war, gibt es bereits Methoden zum sicheren Löschen. Ab einer gewissen Datenmenge und folglich einer großen Anzahl von Schlüsseln ist allerdings eine effiziente Datenstruktur zum Verwalten der Schlüssel nötig. Demnach wäre eine einzige Liste mit allen Schlüsseln in Anbetracht des linear ansteigenden Aufwands beim Suchen und Einfügen eine denkbar schlechte Lösung. Da auch die Schlüssel nicht im Klartext vorliegen dürfen, müsste auch diese Liste verschlüsselt werden. Jedes Mal, wenn eine Datei eingefügt (Schlüssel hinzufügen) wird oder eine Datei gelesen wird (Schlüssel auslesen) oder eine Datei gelöscht wird (Schlüssel löschen), müsste diese Liste entschlüsselt und wieder verschlüsselt werden, was bei einer großen Anzahl einen erheblichen Aufwand darstellt.

In SDOS wird der Ansatz verfolgt, mehrere Knoten zu erzeugen, die jeweils eine bestimmte Anzahl von Schlüsseln verwalten, wobei nur eine geringe Anzahl von Knoten bei einer beliebigen Operation durchsucht werden müssen. Diese Knoten liegen verschlüsselt in der Cloud, deren Schlüssel zum Entschlüsseln in jeweils anderen Knoten gespeichert sind. Somit können alle Schlüssel in der Cloud gespeichert werden, die für das Verschlüsseln und Entschlüsseln der Dateien und Knoten zuständig sind.

Am folgenden Beispiel soll die grobe Vorgehensweise vorgestellt werden. Angenommen es befinden sich bereits mehrere Dateien in der Cloud. Diese liegen verschlüsselt vor. Die dazugehörigen Schlüssel befinden sich in Knoten, welche auch verschlüsselt vorliegen. Um nun eine Datei sicher zu löschen, muss der entsprechende Schlüssel innerhalb des Knoten gelöscht werden. Der Benutzer startet SDOS auf einem lokalen Rechner und übergibt in der Kommandozeile den Befehl zum Löschen einer Datei. Mit einem Masterkey, welcher entweder lokal oder innerhalb eines HSM in der Cloud vorliegt, wird der Zugriff auf die Datenstruktur ermöglicht. Alle im Verlauf dieser Operation benötigten Knoten werden aus der Cloud heruntergeladen und lokal entschlüsselt, bis der entsprechende Schlüssel für die zu löschende Datei gefunden wird und gelöscht werden kann. Damit ist sichergestellt, dass zu keinem Zeitpunkt unverschlüsselte Knoten in der Cloud vorhanden sind. Zugleich verfügt ausschließlich der Benutzer über den Masterkey. Nicht authorisierte Zugriffe des Cloud Anbieters bzw. Angriffe auf die Cloud bleiben wirkungslos.

Im Folgenden wird die Datenstruktur vorgestellt. Es wird darauf geachtet, eine möglichst kleine Anzahl von Knoten durchsuchen zu müssen, um den gesuchten Schlüssel zu finden. Des Weiteren werden die Benutzerfunktionen *Daten einfügen*, *Daten suchen* und *Daten löschen* anhand von Flussdiagrammen erläutert.

4.1 Datenstruktur

Wie bereits erwähnt, werden die Schlüssel in mehreren Knoten unterteilt. Jeder Knoten besitzt die folgenden vier Attribute:

- *ID*
  Jeder Knoten besitzt eine ID, die ihn eindeutig referenzierbar macht.
- *refs*
  In der Liste refs werden die Referenzen auf Dateien beziehungsweise Knoten gespeichert. Es wird eine leere Liste der Größe 128 initialisiert.
- *keys*
  In der Liste keys werden die Schlüssel zum Dekodieren von Dateien beziehungsweise Knoten gespeichert. Es wird eine leere Liste der Größe 128 initialisiert.
- *isObj*
  Die Liste isObj enthält die Informationen, ob es sich bei der Referenz an dieser Stelle um eine Datei oder ein Objekt handelt. Dabei steht True für eine Datei, False für einen Knoten. Die Liste der Größe 128 wird mit False initialisiert.

Die *ID* jedes Knoten erhält einen Präfix *Node_*, um ihn als solches zu identifizieren. Ausnahme ist der Knoten *root*. Beim ersten Start von SDOS wird dieser automatisch erstellt. Dieser root-Knoten dient als Einstiegspunkt und wird bei jeder Operation als erstes aufgesucht. Abbildung 4.1. zeigt einen root-Knoten, welcher bereits drei Referenzen auf Dateien besitzt und die dazugehörigen Schlüssel speichert. Erst bei Bedarf werden weitere Knoten hinzugefügt, welche im jeweils vorherigen Knoten referenziert werden. Es entsteht die in Abbildung 4.2 gezeigte Baumstruktur. Der Einfachheit halber wurden in der Abbildung die IDs *Node_0 - Node_127* verwendet, um die maximale Anzahl von Referenzknoten zu verdeutlichen. Vollständige Bäume der Tiefe n besitzen damit 128^n Knoten. Diese Anzahl wird allerdings nicht ausgeschöpft. Grund dafür ist die verwendete Hashfunktion, welche für die Einordnung der Daten innerhalb der Knoten zuständig ist.

Um eindeutige Referenzen auf die Objekte zu erhalten, wird eine Hashfunktion verwendet. SDOS verwendet augenblicklich einen simplen Algorithmus, der die Buchstaben des Dateinamen auf ihre entsprechenden ASCII-Codes hasht (z.B. den Buchstaben 'a' auf ASCII(a) = 97). Dementsprechend beschränkt sich die Anzahl der Dateireferenzen innerhalb eines Knotens auf 128 (Größe der ASCII-Tabelle). Bedenkt man weiterhin, dass Dateinamen zum Großteil aus Klein- und gelegentlich Großbuchstaben sowie einigen Sonderzeichen bestehen, sinkt diese Zahl auf durchschnittlich 40-50 Zeichen herab. Nützlich ist die Tatsache, dass es aufgrund des eindeutigen Namensraums zu keinen Kollisionen beim Hashing kommen kann. Denkbar ist allerdings die Verwendung einer anderen Hashfunktion, um die maximale Anzahl von Dateireferenzen pro Knoten bei Bedarf zu erhöhen oder zu beschränken. Da an diesem Punkt die kryptografischen Aspekte keine Rolle spielen, ist vor allem auf einen effizienten Algorithmus mit einer möglichst guten Gleichverteilung der Hashwerte auf die Eingabewerte zu achten.

In Anbetracht dessen sinkt die Anzahl der Knoten eines Baumes bei einer Tiefe von n auf etwa 50^n Knoten. Ein vollständiger Baum (in diesem Sinne mit jeweils 50 Referenzen) kann Dateireferenzen nur in den Blättern enthalten, da die Knoten höherer Hierarchie Referenzen auf andere Knoten besitzen

root	refs	keys	isObj
0	...	...	...
...	...	...	...
ASCII(M)) 77	Metadata.xml	gfd2lv2nh45ld...	TRUE
...	...	...	...
ASCII(t) = 116	testdata.txt	4bdn3vl3nys8...	TRUE
ACSII(u) = 117	und.txt	fbk234bkdb38...	TRUE
...	...	...	...
127	...	...	...

Abbildung 4.1: root-Knoten mit drei Referenzen

müssen. Daraus lassen sich die mögliche Anzahl von Schlüsseln ermitteln. Ein Baum der Tiefe n besitzt 50^{n}-$\sum_{i=0}^{n-1} 50^{i}$. Ein Baum der Tiefe 4 kann damit circa 6 Millionen Schlüssel speichern - ein Baum der Tiefe 5 bereits über 300 Millionen. Trotz dieser enormen Anzahl von Schlüsseln ist es bei einer Tiefe von 5 sichergestellt, den gesuchten Schlüssel - falls vorhanden - nach maximal 5 durchsuchten Knoten zu finden.

4.2 Operationen

Die Operationen, die SDOS zur Verfügung stellt, beschränken sich zunächst auf die wichtigsten Operationen *insertData*, mit der Daten hinzugefügt werden können, *getData*, womit Daten aus der ausgelesen werden können und *deleteData*, womit Daten sicher gelöscht werden können. Wichtig bei der Erstellung der Operationen war vor allem, dass dem Benutzer durch die Verwendung von SDOS kein zusätzlicher Aufwand entsteht. Die einzige vom Benutzer getätigte Eingabe ist daher der Aufruf der Operationen mit dem Dateinamen als Übergabeparameter.

4.2.1 Daten einfügen

Um Daten einzufügen verwendet der Benutzer die Funktion *InsertData*. Der dabei übergebene Parameter *Dateiname* spezifiziert die Datei, die verschlüsselt werden soll und deren Schlüssel in der Datenstruktur gespeichert werden soll. Diese Funktion ist maßgebend für die Entstehung der in Abbildung 4.2 gezeigten Baumstruktur, da ein neuer Knoten nur dann erstellt wird, wenn für eine neu eingefügte Datei in den bereits bestehenden Knoten keine geeignete Stelle zum Referenzieren gefunden wird. In Abbildung 4.3 wird der Ablauf beim Einfügen einer Datei gezeigt.

Die übergebene Datei wird zuerst verschlüsselt und der dazugehörige Schlüssel erstellt. Jede Operation lädt zunächst den root-Knoten. Mittels einer for-Schleife wird über die Buchstaben des Dateinamens

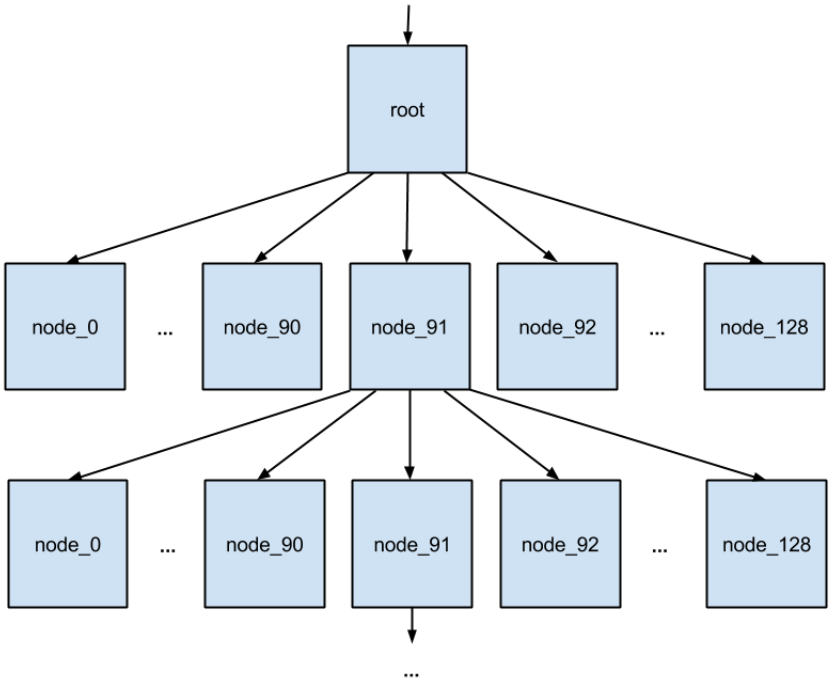

Abbildung 4.2: Baumstruktur der Knoten

iteriert und diese jeweils in ihre ACSII-Zahl übersetzt. In jeder Iteration wird geprüft, ob der Knoten, in dem sich das Programm befindet, an der Stelle refs(ASCII-Zahl) belegt ist oder nicht. Falls die Stelle frei ist, kann die Referenz eingefügt werden und der dazugehörige Schlüssel wird an der gleichen Stelle in der Liste keys eingefügt, woraufhin die Datei erfolgreich in SDOS referenziert ist. Durch diese Vorgehensweise werden Dateien so früh wie möglich in den Baum eingefügt. Dies hat zum Vorteil, dass die Tiefe des Baumes möglichst gering bleibt. Ist die Stelle, an welcher die Datei referenziert werden könnte bereits besetzt, muss überprüft werden, ob sie von einem anderen Knoten oder einer anderen Datei besetzt wird. Ist es ein Knoten, wird eine weitere Iteration durchgeführt und im referenzierten Knoten die Vorgehensweise wiederholt. Falls eine Datei die Stelle besetzt, muss ein neuer Knoten erzeugt werden und an derselben Stelle referenziert werden. Die bereits an dieser Stelle referenzierte Datei sowie die neu eingefügte Datei werden im neuen Knoten referenziert. Spezialfälle beim Einfügen werden im nachfolgenden Kapitel Implementierung genauer betrachtet.

4.2.2 Daten suchen

Da die eingefügten Daten ausschließlich in verschlüsselter Form vorliegen, ist es zum Lesen bzw. Schreiben von Daten notwendig, diese zuerst zu entschlüsseln. Der Ablauf der Funktion *GetData* ist dem der Funktion *InsertData* ähnlich und wird in Abbildung 4.4 gezeigt. Der Benutzer gibt zuerst an, auf welche Datei er Zugriff haben möchte. Wieder wird zuerst der root-Knoten geladen, von dem aus

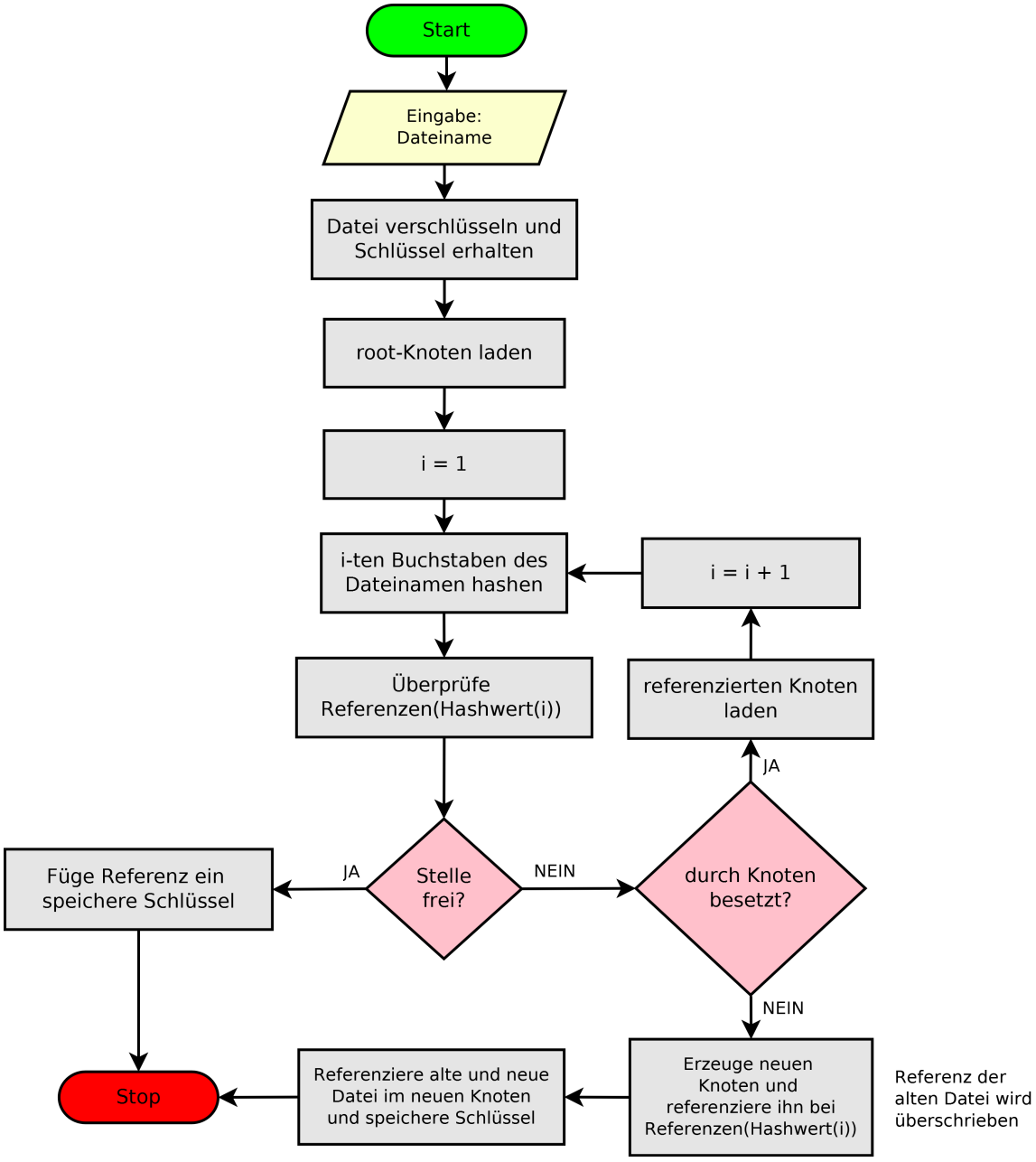

Abbildung 4.3: Flussdiagramm - Datei einfügen

mittels der Hashwerte die Stellen gefunden werden, in denen sich die Datei befinden könnte. Wie beim Einfügen kann es nötig sein, mehrere Iterationen durchzuführen, um den gesuchten Knoten zu finden, welcher die gesuchte Datei referenziert. Wird zu einem beliebigen Zeitpunkt eine leere Stelle vorgefunden, besitzt die Datei in SDOS keine Referenz. Wird die gesuchte Datei gefunden, d.h. der Dateiname stimmt mit einem Wert aus der Liste refs überein, kann der Schlüssel aus der Liste keys ausgelesen werden, womit die Datei entschlüsselt werden kann.

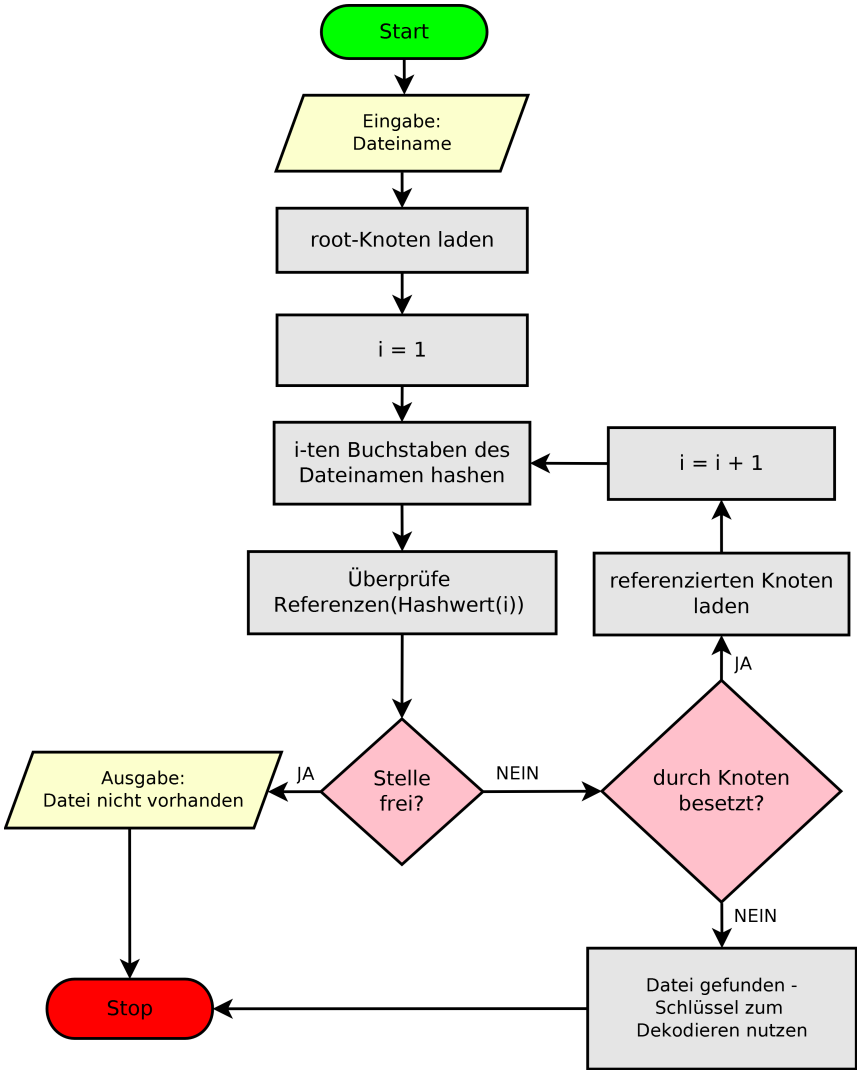

Abbildung 4.4: Flussdiagramm - Datei suchen

4.2.3 Datei löschen

Neben der Erstellung einer geeigneten Datenstruktur zur Schlüsselverwaltung ist vor allem das sichere Löschen die Hauptaufgabe von SDOS. Durch Aufruf der Funktion *DeleteData* wird analog zur Funktion *GetData* zuerst die Datei gesucht. Der einzige Unterschied besteht darin, dass sobald die zu löschende Datei gefunden wird, diese nicht, wie bei *GetData* entschlüsselt wird, sondern die entsprechende Referenz samt Schlüssel gelöscht wird. Daraufhin lässt sich die verschlüsselte Datei nicht mehr entschlüsseln, auch wenn diese noch physikalisch vorhanden ist. Die Datei ist somit sicher gelöscht. Abbildung 4.5 zeigt den entsprechenden Ablauf.

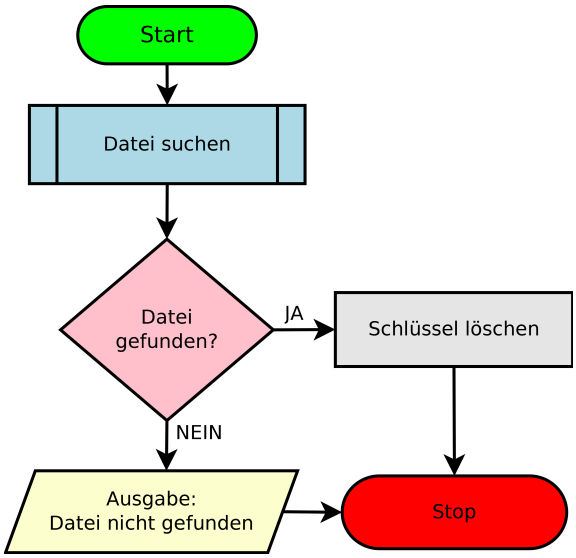

Abbildung 4.5: Flussdiagramm - Datei löschen

5 Implementierung

Dieses Kapitel beschreibt die Implementierung der vorangegangenen Konzepte, sowie die Probleme und mögliche Lösungen, die dabei entstehen. Zuerst wird der Einsatz von OpenStack erläutert. Des Weiteren werden die Module *crypt.py* sowie *SDOS.py* vorgestellt. Anschließend erfolgt eine Evaluation anhand verschiedener Datensätze, bei welcher die entstandene Datenstruktur genauer untersucht wird.

5.1 OpenStack

Als Cloud Plattform zur Implementierung von SDOS wurde OpenStack ausgewählt. OpenStack zählt zu den am meisten verbreiteten Cloud Softwares im Bereich des IaaS. Neben OpenStack seien noch Eucalyptus[Euc] und OpenNebula[ON] zu erwähnen. Eucalyptus dient eher der Erzeugung von Private Clouds und skaliert weniger gut als OpenStack. Der Vorteil liegt in der Unterstützung der AWS-API, da Amazon der größte kommerzielle Anbieter von IaaS ist. OpenNebula erhält im Vergleich zu OpenStack deutlich weniger Unterstützung externer Firmen. Vor allem Rackspace - der zweitgrößte kommerzielle Anbieter von IaaS - verwendet selbst OpenStack. In Anbetracht dessen ist zu erwarten, dass OpenStack weiterhin eine weitverbreitete Cloud Software bleibt und sich möglicherweise als Quasi-Standard im Bereich der IaaS Software etabliert.

Die von SDOS benötigten Komponenten sind der Identity Service namens Keystone und der Object Storage - auch Swift genannt. Keystone wird in SDOS verwendet, um den Benutzer zu authentifizieren. Dies geschieht indirekt über die Swift Komponente, in welcher die verschlüsselten Dateien und Knoten in Containern gespeichert werden. Keystone muss deshalb nicht als eigenständiges Modul integriert werden. OpenStack bietet für Swift eine unter der Apache-Lizenz lizenzierte Python-Bibliothek. Diese enthält die wichtigsten Funktionen von Swift und wird in SDOS integriert (genaueres im Kapitel *Prototyp*).

Die Integration anderer Cloud Software ist ohne großen Aufwand möglich. Zwischen der Swift-Bibliothek und SDOS besteht eine lose Kopplung. Die Authentifizierung findet zu Beginn des Programmstarts unabhängig von der Funktionsweise von SDOS statt. Des Weiteren werden hauptsächlich zwei Funktionen verwendet - Daten in die Cloud hochladen und Daten aus der Cloud herunterladen. Als Parameter ist ausschließlich der Name des spezifizierten Objekts anzugeben.

5.2 Modul - crypt.py

Das Modul *crypt.py* implementiert das Verschlüsselungsverfahren von SDOS. Dieses wird durch das *Python Cryptographic Toolkit) pycrypto*[PyC] bereitgestellt, welches unter anderem neben AES weitere symmetrische sowie asymmetrische Verschlüsselungsverfahren, Hashfunktionen und einen Zufallsgenerator zum Erstellen von Schlüsseln. *crypt.py* importiert die Bibliotheken *AES* sowie *Random*, welches den Zufallsgenerator implementiert. Die in SDOS verwendete Schlüssellänge beträgt 128 Bit, um den Aufwand so gering wie möglich zu halten. Um die Schlüssellänge zu ändern und damit einen erhöhten Sicherheitsgrad zu erreichen, muss in *crypt.py* bei der Funktion

```
def encrypt(message, dataKey, key_size = 128)
```

der Parameter entsprechend angepasst werden. Zulässige Schlüssellängen sind 128, 192 und 256. Die Übergabe eines Schlüssels im Feld *dataKey* ist optional. Wird kein Schlüssel übergeben, wird ein neuer Schlüssel mittels

```
key = Random.OSRNG.posix.new().read(key_size//8)
```

erstellt.

Wie in Kapitel 2.3 bereits beschrieben, entspricht nicht jede Datei den Anforderungen des Verfahrens. Die Anzahl der Zeichen der Datei müssen durch Blöcke der Größe 32 teilbar sein. Deshalb müssen die Anzahl der Zeichen meist künstlich erhöht werden. Dies geschieht mit der folgenden Funktion:

```
pad(message):
      x = AES.block_size - len(s) % AES.block_size
      return message + (chr(x) * x)
```

Mit *x* wird die Anzahl der Zeichen berechnet, um die erforderliche Blocklänge zu erreichen. Zurückgegeben wird die ursprüngliche Nachricht mit den zusätzlichen Zeichen, wobei die Zeichen dem ASCII-Wert von x entsprechen. Ist der Schlüssel erstellt und die Dateilänge korrekt, kann die Datei verschlüsselt werden. Der Rückgabewert der Funktion ist ein Tupel, bestehend aus dem verschlüsselten Inhalt der Datei sowie dem verwendeten Schlüssel.

5.3 Modul - SDOS.py

Die Operationen *InsertData*, *GetData* und *DeleteData* wurden in Kapitel 4.1.2 bereits beschrieben. Im Folgenden wird der genaue Ablauf dieser Funktionen beschrieben und an Beispielen gezeigt, zu welchen Problemen die genannten Abläufe führen können sowie Lösungen dieser Probleme. Des Weiteren wird gezeigt, welche Verfahren nötig sind, um die Anbindung von SDOS an OpenStack Swift zu ermöglichen.

5.3.1 Funktion - InsertData

Wie das Prinzip beim Einfügen von Dateien funktioniert, wurde bereits in Kapitel 4.1.2 beschrieben. Im Folgenden werden allerdings Spezialfälle betrachtet, die nach dem obigen Einfügeverfahren einen

unnötigen Overhead erzeugen würden. Folgendes Beispiel zeigt den Fall von zwei sich stark ähnelnden Dateinamen.

1. Beim ersten Start von SDOS wird zuerst der root-Knoten erstellt, welcher zuerst leer ist. Besteht der root-Knoten bereits, so muss er zuerst entschlüsselt werden, um auf die Datenstruktur zugreifen zu können.
2. Wird nun eine Datei eingefügt, die testdata.txt heißt, wird sie gemäß dem Hashingalgorithmus aus Kapitel 4.1.1 im root-Knoten an der Stelle ASCII(t)=116 eingefügt.
3. Da der root-Knoten an der Stelle 116 in der List refs noch nicht belegt ist, kann die Datei "testdata.txt"dort referenziert werden. Alle Dateien werden an der ersten nicht belegten Stelle referenziert. Der Name der referenzierten Datei wird im Feld refs(116) gespeichert.
4. Die Datei wird mittels des Rijndael-Algorithmus verschlüsselt und der dabei erstellte Schlüssel in der Liste keys an der Stelle 116 abgelegt.
5. In der Liste isObj wird an der 116 der boolesche Wert von FALSE auf TRUE gesetzt.
6. Die Datei ist nun vollständig referenziert und der root-Knoten wird wieder verschlüsselt. Der Schlüssel des root-Knoten kann dabei beispielsweise in einem HSM gehalten werden.
7. Des Weiteren wird eine Datei Metadata.xml sowie und.txt eingefügt. Die Vorgehensweise entspricht derselben wie oben. Zu diesem Zeitpunkt entspricht das Beispiel der Abbildung 4.2.
8. Nun wird nochmals eine Datei mit Namen und.xml eingefügt. Wieder wird die Stelle ASCII(u)=117 betrachtet, allerdings ist diese nun durch und.txt besetzt.
9. Aufgrund dieser Kollision muss ein neuer Knoten erstellt werden. Der Knoten wird an der Stelle referenziert, wo die Kollision aufgetreten ist. Der Name des Knoten ergibt sich durch folgendes Prinzip.
    a) Jeder Knoten erhält zunächst das Präfix node_ um ihn als solchen zu kennzeichnen.
    b) Befindet sich die Kollision im root-Knoten, wird dem Präfix node_ das Zeichen $ASCII^{-1}$(Stelle der Kollision) angefügt.
    c) Wurden dem Präfix bereits Zeichen hinzugefügt (Bsp.: node_uvx), so werden diese beibehalten und gemäß b) das weitere Zeichen hinzugefügt.
10. Es wird ein neuer Knoten namens node_u erstellt ($ASCII^{-1}$(117) = u).
11. Der boolesche Wert in der Liste isObj(117) wird von TRUE auf FALSE gesetzt.

    Weiterhin müssen die Dateien und.xml sowie und.txt referenziert werden. Da sich der Knoten im zweiten Level des Baumes befindet (root-Knoten = Level 1), wird die Datei anhand ihres zweiten Zeichens eingeordnet. Wie zu sehen ist, ist das zweite Zeichen beider Dateinamen n, was wiederum zu einer Kollision führen würde - so auch beim dritten Zeichen (d) und vierten Zeichen (.). Erst ab der fünften Stelle unterscheiden sich die Dateien, wodurch eine entsprechende Referenzierung erst im Knoten *node_und.* möglich wäre. Die Tiefe des Baumes erhöht sich dabei um 4, obwohl nur eine Datei hinzugefügt worden ist. Um diese unnötige Erzeugung von Knoten, welche sehr zu Lasten der Performanz gelten, da jeder von ihnen

verschlüsselt werden muss, zu vermeiden, werden die Dateien und.xml und und.txt nach einem anderen Prinzip im Knoten node_u eingefügt.

12. Die vorher bereits referenzierte Datei (und.txt) wird im neuen Knoten node_n an der exklusiven Stelle 0 in der Liste refs referenziert. Der vorherige Schlüssel wird in keys(0) gespeichert, das Feld isObj(0) wird auf TRUE gesetzt.
13. Die neue Datei und.xml wird gemäß der oberen Vorschriften an der Stelle ASCII(n) = 110 gespeichert, der entsprechende Schlüssel erzeugt und gespeichert sowie der boolesche Wert zu TRUE verändert.
14. Knoten node_u wird verschlüsselt und der dazugehörige Schlüssel wird im root-Knoten an der Stelle 117 in der Liste keys gespeichert.
15. Der root-Knoten wird verschlüsselt.

Dieses Vorgehen wird nochmals in den Abbildungen 5.1-5.3 verdeutlicht.

root	refs	keys	isObj
0	...	...	...
...	...	...	...
ASCII(M)) 77	Metadata.xml	gfd2lv2nh45ld...	TRUE
...	...	...	...
ASCII(t) = 116	testdata.txt	4bdn3vl3nys8...	TRUE
ACSII(u) = 117	und.txt	fbk234bkdb38...	TRUE
...	...	...	...
127	...	...	...

Abbildung 5.1: root-Knoten vor Einfügen von und.xml

Wie bereits erwähnt, wird versucht, alle Dateien so früh wie möglich zu referenzieren, um die Tiefe des Baums möglichst gering zu halten. Aufgrund dieser Methode entsteht allerdings ein weiteres Problem. Angenommen, es werden mindestens acht Dateien gespeichert, deren Dateinamen mit *test* beginnt. Nach obigem Verfahren würden folglich zusätzlich zum root-Knoten die Knoten node_t, node_te, node_tes und node_test entstehen, die jeweils zwei der acht Dateien referenzieren. Wenn jetzt eine Datei gespeichert werden soll, die *test* heißt, so ist, wenn man das obige Verfahren verwendet, kein Platz mehr in der Datenstruktur vorhanden, um diese Datei zu referenzieren, siehe Abbildung 5.4. Wie man bereits erkennen konnte, werden Dateien nur in Knoten referenziert, deren Namen einen Präfix des Dateinamens darstellen. Demnach würde die Datei *test* entweder im root-Knoten, Knoten node_t, node_te, node_tes referenziert werden müssen. Da dies nicht mehr möglich ist, wird jedem Dateinamen vor dem Einfügen das Sonderzeichen $ASCII^{-1}(1)$ hinzugefügt. Demnach kann die Datei im Knoten node_test an Stelle 1 referenziert werden.

root	refs	keys	isObj
0	...	...	...
...	...	...	...
ASCII(M)) 77	Metadata.xml	gfd2lv2nh45ld...	TRUE
...	...	...	...
ASCII(t) = 116	testdata.txt	4bdn3vl3nys8...	TRUE
ACSII(u) = 117	node_u	fbk234bkdb38...	FALSE
...	...	...	...
127	...	...	...

Abbildung 5.2: root-Knoten mit Referenz auf neuen Knoten node_u

node_u	refs	keys	isObj
0	und.txt	fbk234bkdb38...	TRUE
...	...	...	...
ASCII(n) = 110	und.xml	gu3ks32nglFK2...	TRUE
...	...	...	...
127	...	...	...

Abbildung 5.3: node_u mit referenzierten Dateien und.xml sowie und.txt

Gleiche Dateinamen werden in SDOS nicht weiter berücksichtigt, da dieses Problem von zugrundeliegenden Speichersystem selbst verwaltet wird.

Ein ungelöstes Problem bleibt allerdings, wenn man es mit einer großen Menge von sich ähnelnden Dateinamen zu tun hat. Angenommen, es werden 100 Dateien mit dem Namen *sehrLangerDateinameMitGleichemNamen* mit dem Suffix 1-100. Diese Dateinamen unterscheiden sich in den ersten 36 Zeichen nicht. Hier trifft der erste genannte Spezialfall zu. Pro Knoten können dadurch zwei Dateien referenziert werden. Dementsprechend müssen mindestens 18 Knoten erzeugt werden, um alle Dateien zu referenzieren. Wie in Abbildung 5.5 zu sehen ist, entsteht dabei ein listenartiger Baum. Aufgrund der unterschiedlichen Zahlen am Ende des Dateinamens können die Dateien eindeutig referenziert werden. Infolge der derzeitig verwendeten Hashfunktion lässt sich dieses Problem nicht vollständig beheben. Eine Behelfslösung wäre die Verwendung weiterer Felder der Listen innerhalb der Knoten (ASCIIs Zeichenkodierung definiert die ersten 33 Zeichen als nicht druckbar). Allerdings ist bei noch größeren Datensätzen auch diese Lösung nicht optimal. Eine elegantere Lösung wäre dagegen die Verwendung einer anderen Hashfunktion, welche Hashwerte unabhängig des Dateinamens erzeugt. Hierfür müssen folgende Änderungen vorgenommen werden:

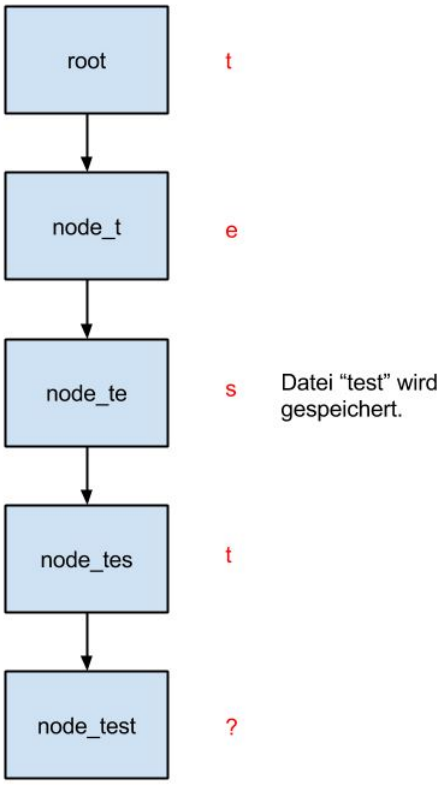

Abbildung 5.4: Datenstruktur besitzt keinen Platz mehr für eingefügte Datei

- *Wahl einer geeigneten Hashfunktion*
  Bei der Wahl ist vor allem auf zwei Eigenschaften zu achten. Zum Einen sollte es ein schneller, effizienter Algorithmus sein. Da diese Hashfunktion ausschließlich dafür zuständig sind, geeignete Stellen in der Datenstruktur zum Referenzieren zu finden, spielen sicherheitskritische Aspekte der Funktionen keine Rolle. Kryptologische Hashfunktionen sind aufgrund des erhöhten Aufwands beim Erzeugen der Hashwerte eher zu vermeiden. Zum Anderen sollten die Hashfunktionen eine möglichst gute Gleichverteilung der Hashwerte garantierten, um den eben besprochenen Problemfall zu vermeiden.
- *Implementierung*
  Selbstverständlich muss zuerst die Hashfunktion implementiert werden. Weitere Änderungen sind an folgenden Stellen nötig. Die Attribute der Knoten müssen angepasst werden. Die drei Listen *refs*, *keys* und *isObj* besitzen derzeit aufgrund der Größe der ASCII-Tabelle eine Größe von 128. Diese müssen entsprechend der neu verwendeten Hashfunktion abgepasst werden. Dazu ist es nötig zu wissen, welche Hashwerte erzeugt werden und wie diese zur Einordnung in den Knoten verwendet werden sollen. Die einzige Änderung im Programmablauf ist, dass SDOS nicht wie bisher über den Dateinamen iteriert, sondern über den Hashwert.

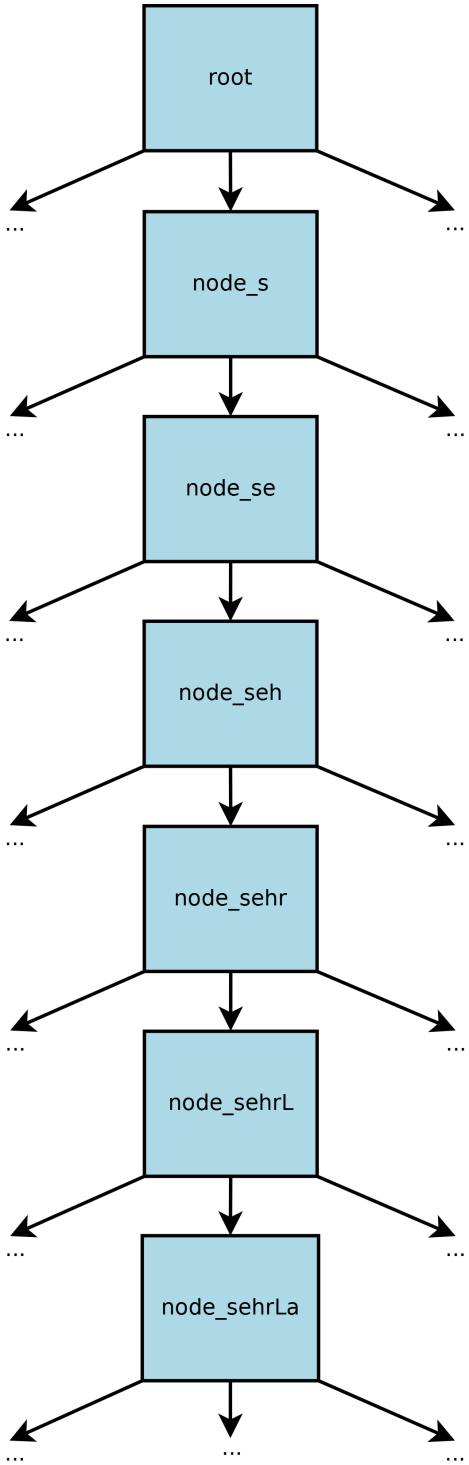

Abbildung 5.5: Datenstruktur nach Einfügen vieler Dateien mit ähnlichen Namen

Im folgenden Beispiel wird eine Hashfunktion aufgezeigt, die aus Dateinamen eine 32 Stellen lange Hexadezimalzahl erzeugt. Des Weiteren werden mehrere Möglichkeiten gezeigt, wie aus diesem Hashwert

eine Einordnung in die Datenstruktur erfolgen kann. Mit oberem Beispielnamen erhält man Hashfunktion(sehrLangerDateinameMitGleichemNamen1)=A158B3EF734C123F1285A21235CAB199. Würde man über dasselbe Verfahren, um die Daten einzufügen (über den Dateinamen iterieren) mit dem neuen Hashwert verwenden (über den Hashwert iterieren), würde pro Iteration eine Hexadezimalziffer betrachtet werden, die zwischen 0-15(F) liegt. Demzufolge würden nur 15 Dateien pro Knoten referenziert werden könnten. Der Aufwand, um den Knoten jedes Mal zu verschlüsseln und zu entschlüsseln ist zu hoch, um nur 15 Dateien zu speichern. Zwei andere Varianten werden in Abbildung 5.6 gezeigt. Wie zu sehen ist, werden mit der neuen Hashfunktion zwei sehr ähnlichen Dateinamen komplett verschiedene Hashwerte zugewiesen. Die zuerst in SDOS verwendete Variante war, aus einer bestimmten Anzahl der Hexadezimalziffern (im Beispiel sind es vier) die Quersumme zu berechnen und diese zur Einordnung zu verwenden. Der Eingabebereich geht somit von 0 (Quersumme von 0000) bis 60 (Quersumme von FFFF). Die Gleichverteilung im Vergleich zur ASCII-Methode auf die Eingabewerte ist verbessert. Aus der im Beispiel gezeigten Hexadezimalzahl können maximal acht Quersummen berechnet werden, woraus folgt, dass der Baum eine maximale Tiefe von 8 hätte, womit, bei perfekter Gleichverteilung, über 100 Billionen Dateien gespeichert werden könnten. Auch wenn es unwahrscheinlich ist, dass eine so große Datenmenge entsteht, bleibt diese allerdings endlich. Das ASCII-Verfahren besitzt diesbezüglich keine Grenzen.

Ein zusätzlicher Vorteil der ASCII-Methode im Vergleich zur oberen Methode ist außerdem die Kollisionsfreiheit. Da die Hashwerte ausschließlich entsprechend des Dateinamens gehasht werden, kann es nur bei zwei gleichen Dateinamen zu einer Kollision kommen. Auch wenn die Kollisionswahrscheinlichkeit der oberen Variante gering wäre, so erhöht sie sich durch die Verwendung der Quersummen, wie in der Abbildung 5.6 zu sehen ist. Zwei Hashwerte, die sich in jeder Ziffer unterscheiden, generieren die gleichen Quersummen. In diesem Fall sind weitere Verfahren notwendig, die mit diesen Kollisionen umgehen.

In der zweiten Variante wird die Hexadezimalzahl in 2er-Blöcke eingeteilt, welche jeweils in Dezimalzahlen umgewandelt werden. Der Eingabebereich ist 0 bis 255 (FF). Kollisionen wie bei der Quersummenberechnung können dabei nicht auftreten (Die Dezimalzahl 161 kann nur durch A1 dargestellt werden). Nachteil dieser und der vorherigen Variante bleibt allerdings der zusätzliche Aufwand, der nach dem Hashing entsteht.

Die gleichzeitige Verwendung mehrerer Hashfunktionen ist kritisch. Dateinamen, die mit einer bestimmten Hashfunktion gehasht und in die Datenstruktur eingefügt werden, müssen bei darauf folgenden Zugriffen dieselbe Hashfunktion verwenden, um den gleichen Hashwert zu erhalten, um die Referenz zu finden. Hierzu wäre ein weiteres Verwaltungsinstrument nötig, welches zu Dateinamen angibt, welche Hashfunktion verwendet wurde.

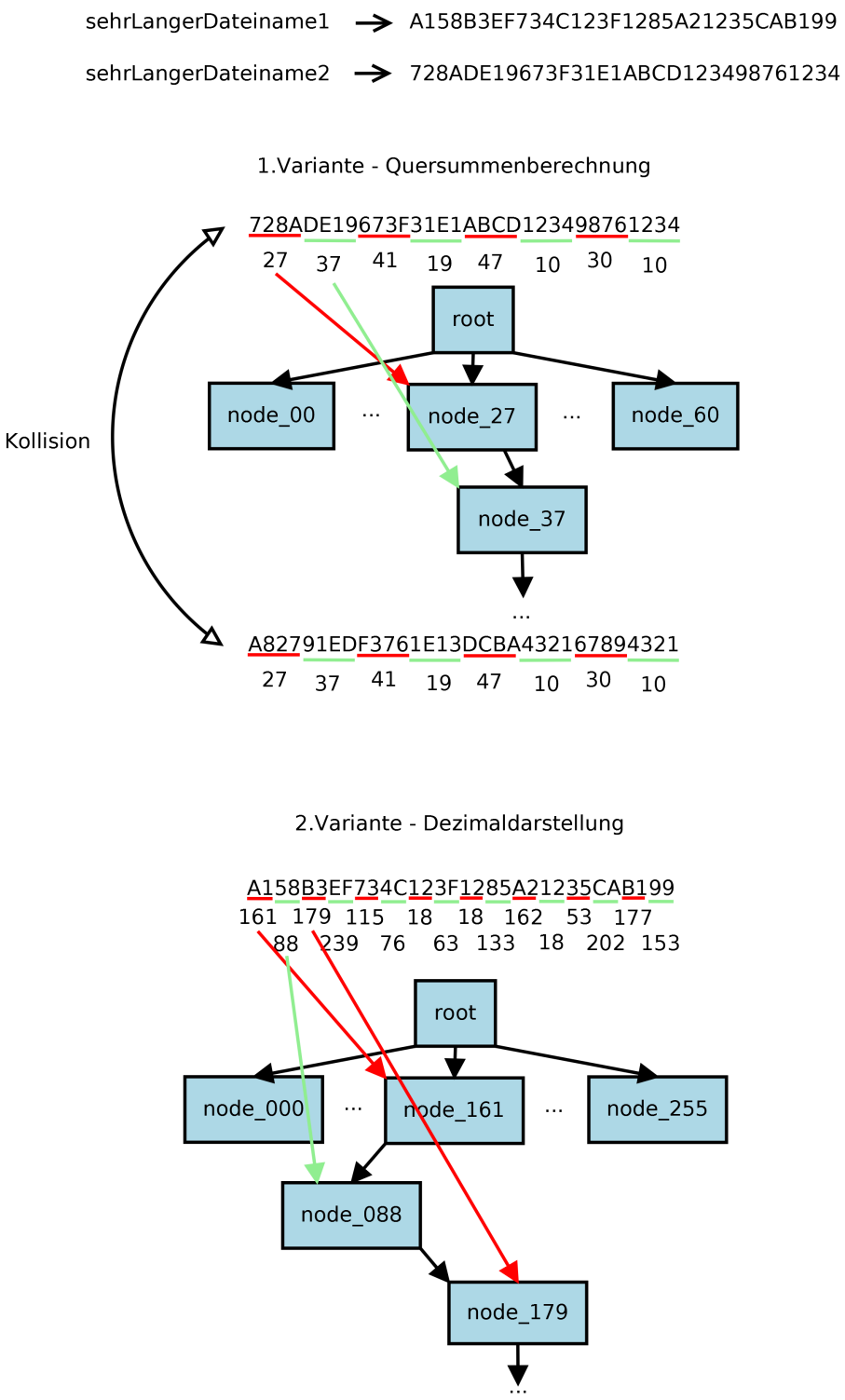

Abbildung 5.6: Weitere Hashfunktionen

5.3.2 Funktion - GetData

Durch die eben angesprochenen Spezialfälle müssen auch in der Funktion *GetData* Änderungen vorgenommen werden, da es nicht ausreichend ist, die Knoten nur anhand de Hashwerte zu durchsuchen. Ausnahme ist der root-Knoten, da die oben genannten Spezialfälle auf diesen nicht zutreffen können. Demnach ist es nötig, bei der Suche jeweils im Knoten die exklusiven Stellen *refs(0)* sowie *refs(1)* zu durchsuchen. Pro Knoten müssen bei jeder Suche drei Stellen untersucht werden, was zwar den Aufwand des Suchens pro Knoten erhöht, allerdings die Anzahl der durchlaufenen Knoten dadurch verringert. Bei Verwendung anderer Hashfunktion sind entsprechende Veränderungen notwendig.

5.3.3 Funktion - DeleteData

Die Löschfunktion ähnelt der Suchfunktion sehr stark. Die entsprechende Referenz und der dazugehörige Schlüssel wird gesucht. Allerdings wird die Datei nicht entschlüsselt und dem Benutzer zur Verfügung gestellt, sondern der dazugehörige Schlüssel gelöscht. Dementsprechende sind die selben Änderungen wie bei *GetData* erforderlich.

5.4 Prototyp

Im Folgenden wird das Design für die prototypische Implementierung vorgestellt. Neben der Implementierung der oben genannten Funktionen sind weitere Verfahren notwendig, um die Verwendung von SDOS zu ermöglichen. Diese werden im folgenden Programmablauf genauer erläutert. Im Rahmen dieser Arbeit wird SDOS in Kombination mit OpenStack verwendet. Das von OpenStack bereitgestellte Modul swiftClient.py ist eine Python-Bibliothek, die unter der Apache Lizenz 2.0 bereitgestellt wird. Sie vereinfacht die Erstellung einer API für die OpenStack Komponente Swift. Für die Verwendung der einzelnen Funktionen wird zuerst eine Instanz der Klasse *Connection* erstellt. Diese dient zugleich der Authentifizierung. Bei Erstellen dieser Instanz werden der Benutzername und das Passwort übergeben, sowie die URL des installierten Swift. Der *Tenantname*, welcher die Zugriffsberechtigung des Benutzers angibt, wird in SDOS für alle Benutzer auf *admin* gesetzt. Aus dieser Instanz heraus können folgende Funktionen verwendet werden, ohne dass sich der Benutzer bei jeder Operation wieder authentifizieren muss:

- put_container *put_container* wird verwendet, um innerhalb von Swift neue Container zu erstellen. Es wird nur der Name des Containers übergeben.
- put_object Mit *put_object* kann der Benutzer Dateien hinzufügen. Neben des Namens des Containers, in welchem die Datei gespeichert werden soll, müssen der Dateiname und der Dateiinhalt übergeben werden.
- get_object *get_object* lädt die spezifizierte Datei lokal herunter. Erforderlich sind Dateiname und Containername.
- delete_object *delete_object* löscht die vom Benutzer angegebene Datei. Erforderlich sind Dateiname und Containername.

Zu Beginn werden zwei Container erstellt. Die Knoten und die Dateien werden separat in den Containern *NodeContainer* und *DataContainer* gespeichert, um mögliche Kollisionen im Namensraum zu vermeiden. Über die Kommandozeile wird der Benutzer aufgefordert, seinen Benutzernamen und Passwort anzugeben, woraufhin eine Verbindung zu Swift aufgebaut wird und SDOS startbereit ist. Der root-Knoten, falls noch nicht vorhanden, wird automatisch erstellt.

Um sicherzustellen, dass zu keinem Zeitpunkt Dateien oder Knoten unverschlüsselt in der Cloud vorhanden sind, werden alle Operationen lokal durchgeführt. Zu Beachten ist deshalb, dass SDOS nicht für die reine Verwendung in Cloud Computing konzipiert ist, sondern für die Kombination von Cloud Computing Speichersystemen und lokalen Rechnern/Rechenzentren. Am Beispiel einer Leseoperation in eine bereits aus mehreren Knoten bestehende Datenstruktur soll dies verdeutlicht werden.

1. Der root-Knoten wird heruntergeladen und lokal gespeichert. Der zum Entschlüsseln verwendete Schlüssel kann dabei aus einer HSM oder SmartCard gelesen werden (nicht implementiert).
2. Durch Zugriff auf den root-Knoten kann die Einfügeoperation erfolgen. Ist ein weiterer Knoten nötig, um das Einfügen zu ermöglichen, wird auch dieser heruntergeladen und lokal gespeichert. Der benötigte Schlüssel zum Entschlüsseln des neuen Knoten ist in dem root-Knoten zu finden. Bei Bedarf müssen weitere Knoten heruntergeladen werden, deren Schlüssel im jeweils vorherigen Knoten zu finden sind.
3. Ist die Datei gefunden, wird diese heruntergeladen und entschlüsselt und kann damit gelesen werden.
4. Die Knoten werden wieder verschlüsselt und anschließend hochgeladen. Die Knoten, an denen Veränderungen vorgenommen worden (z.B. neuer Schlüssel eingefügt) erhalten einen neuen Schlüssel, mit dem sie verschlüsselt werden.

An diesem Punkt ist nochmals zu erkennen, dass es sehr von Vorteil ist, die Tiefe des Baumes, und damit die Anzahl verwendeter Knoten pro Operation gering zu halten. Das Herunter- und Hochladen beansprucht den Großteil der Zeit einer Operation.

Um die Knoten aus dem Arbeitsspeicher auf die Festplatte zu schreiben und umgekehrt, müssen diese zuerst serialisiert bzw. deserialisiert werden. Verwendet wird dafür das Python Modul *Pickle*, welches die Funktionen *dump()*, dass ein Python-Objekt persistiert, und *load()*, welches eine Datei liest und in das entsprechende Python-Objekt übersetzt, enthält. Dementsprechend müssen dem oberen Ablauf folgende Änderungen hinzugefügt werden. Nachdem ein Knoten heruntergeladen wird, muss dieser zuerst durch *pickle.dump()* deserialisiert werden. Nur so hat SDOS Zugriff auf die Listen innerhalb des Knotens und kann diese auslesen. Nachdem alle Veränderungen vorgenommen wurden, werden sie wieder serialisiert, dann verschlüsselt und anschließend hochgeladen.

5.5 Evaluation

5.5.1 Testumgebung

Als Testumgebung wurde die 64 Bit Version von Windows 7 Professional verwendet. Das zugrunde liegende System besteht aus einer Intel Core i5-2400 CPU @ 3,1 GHz und einem 8 GB Arbeitsspeicher. Die verwendete Python Version ist 2.7.6.

5.5.2 Ergebnisse

Im Folgenden werden mehrere Datensätze verschiedener Größe mittels SDOS in die Datenstruktur eingefügt. Als Datensätze werden verschiedene Ordner (und deren Unterordner) innerhalb des Windows Ordners gewählt. Dateien, deren Namen Zeichen außerhalb der ASCII-Tabelle enthalten, werden herausgefiltert. Es wird versucht herauszufinden, wie sich die Anzahl von Dateien auf die Anzahl der Knoten sowie die Tiefe des Baumes auswirkt und wie lange SDOS benötigt, einen Datensatz bestimmter Größe in eine neu erstellte Datenstruktur einzufügen. Die Knoten werden lokal gespeichert. Optimale Datenstrukturen wären Bäume mit einer möglichst geringen Tiefe.

	Anzahl von Dateien	Max. Tiefe	Durchsch. Tiefe	Anzahl von Knoten	Zeit
1	328	7	4,78	147	2,49s
2	246	23	13,75	182	4,8s
3	26	7	3,88	17	0,26s
4	9295	50	12,82	5316	164,77s
5	9203	23	10,2	5074	150,93s
6	32612	33	8,7	19211	479,75s
7	1001622	100	13,93	489348	22887,16s

Tabelle 5.1: Datensätze verschiedener Größe werden mit SDOS eingefügt

Anhand dieser zwei Datensätze lässt sich zeigen, dass sich weder die maximale Tiefe des Baumes noch die durchschnittliche Tiefe der Knoten pauschal von der Anzahl eingefügter Dateien ableiten lässt. Der zweite Datensatz führt trotz geringerer Anzahl von Dateien zu einem tieferen Baum. Die Anzahl erstellter Knoten ist kein Indiz für die gleichmäßige bzw. ungleichmäßige Verteilung der Dateien. Diesbezüglich ist die durchschnittliche Tiefe zu beachten. Je höher diese im Verhältnis zur Gesamtzahl der Dateien ist, desto ähnlicher wird die entstehende Baumstruktur einer Liste. Zugleich bieten die Abbildungen 5.1 und 5.8 einen genaueren Überblick über die Struktur der Dateinamen. Die y-Achse gibt die Stelle innerhalb des Dateinamens an, die x-Achse den entsprechenden ASCII-Wert. Ein Punkt an der Stelle x = 80, y = 5 gibt demnach an, dass der fünfte Buchstabe eines Dateinamens der Buchstabe P war. Die Farbkodierung ist wie folgt: Dunkelblau - Hellblau - Grün - Gelb - Orange - Rot. Von links nach rechts steigt die Häufigkeit, dass Dateinamen an der gleichen Stelle die gleichen Buchstaben besitzen, abhängig von der Gesamtzahl der Dateien.

Die Abbildung von Datensatz 2 verdeutlicht den Zusammenhang zwischen der Verteilung der Dateinamen und der maximalen Tiefe des Baumes bzw. der durchschnittlichen Tiefe der Knoten. Die ersten sieben roten Punkte in der rechten Hälfte der Abbildung bedeuten, dass der Großteil der Dateien dieselben sieben ersten Buchstaben besitzt. In der Tat beginnen die meisten Dateien dieses Datensatzes mit dem Präfix *Windows*.

Datensatz 3 verdeutlicht nochmals, dass trotz einer geringen Menge von Dateien die maximale Tiefe bei stark ähnlichen Dateinamen schnell ansteigen kann. Obwohl der Datensatz nur ein Zwölftel der Dateien von Datensatz 1 beträgt, stimmen die maximale Tiefe beider Datenstrukturen überein.

Das wichtigste Kriterium dieser Ergebnisse ist die durchschnittliche Tiefe der Knoten, da dies angibt, wie viele Knoten im Durchschnitt durchsucht werden müssen, um einen gesuchten Schlüssel zu finden. Vor allem bei großen Datensätzen ist es erforderlich, dass die durchschnittliche Tiefe nicht linear mit der Anzahl an Dateien steigt.

Datensätze 4, 5 und 6 zeigen, dass auch bei großen Datenmengen die durchschnittliche Tiefe geringer sein kann als bei kleinen Datenmengen (vgl. Datensatz 2). Mit zunehmender Datenmenge ist die maximale Tiefe des Baumes weniger ausschlaggebend für die durchschnittliche Tiefe. Nichtsdestotrotz ist es von großem Vorteil, eine niedrige maximale Tiefe zu erreichen. Da die Priorität der Dateien nicht beachtet wird, ist es möglich, dass die am meisten verwendete Datei im tiefsten Knoten gespeichert wird.

Die Laufzeit hängt von der durchschnittlichen Tiefe sowie der Anzahl von Dateien ab. Je höher die durchschnittliche Tiefe der Knoten, desto mehr Knoten werden durchlaufen, um eine Datei an passender Stelle zu referenzieren. Zu beachten ist, dass die Knoten in diesen Testreihen ausschließlich lokal gespeichert werden. Die in der Praxis gegebene Laufzeit ist entsprechend höher, da die Knoten sowie die referenzierten Dateien in die Cloud hochgeladen werden müssen. Der Durchsatz der oberen drei Beispiele liegt zwischen 56,4 Dateien/s (Datensatz 4) und 67,9 Dateien/s (Datensatz 6).

Der letzte Datensatz beträgt über eine Million Dateien. Das Einfügen dieser Dateien beträgt bereits über sechs Stunden. Die durchschnittliche Tiefe ist weiterhin in akzeptablen Maße (im Vergleich zu den anderen Datensätzen). Mit knapp 500000 erstellten Knoten und circa 2-3 KB/Knoten entsteht ein zusätzlicher Datensatz von bis zu 1,5 GB.

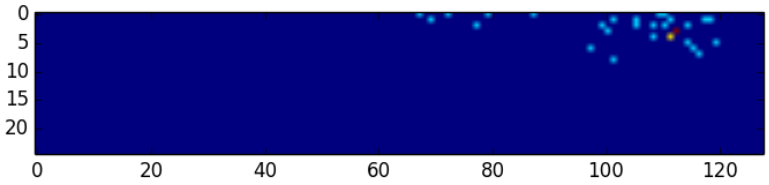

Abbildung 5.7: Datensatz 1 - durchschnittliche Verteilung

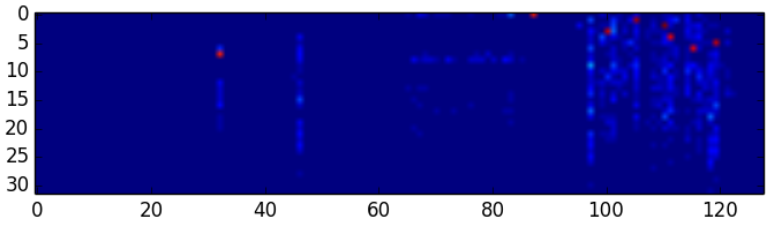

Abbildung 5.8: Datensatz 2 - Ungleichverteilung der ersten Zeichen

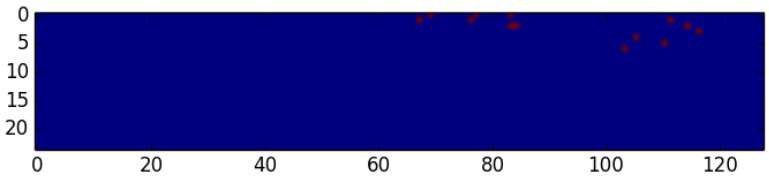

Abbildung 5.9: Datensatz 3 - sehr starke Ungleichverteilung

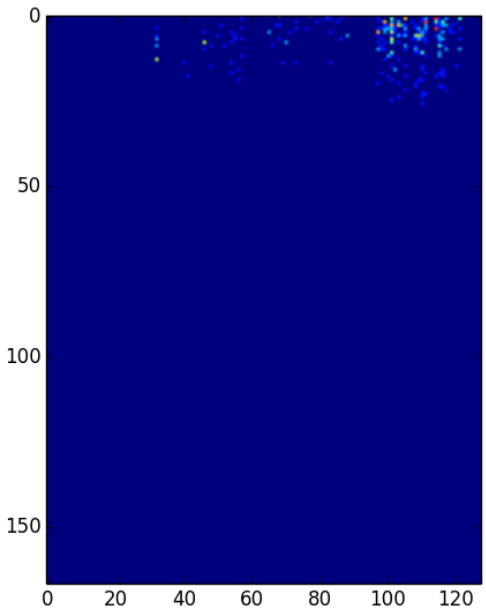

Abbildung 5.10: Datensatz 7 - Verteilung von 1 Mio. Daten

6 Offene Punkte

Im Folgenden werden Erweiterungen vorgeschlagen, die SDOS in Zukunft hinzugefügt werden könnten:

- Speicherung des root-Keys
  Die sichere Verwaltung des root-Keys ist von größter Bedeutung. Geht dieser verloren, besitzt der Benutzer keinen Zugriff auf die Datenstruktur und damit auf die verwalteten Schlüssel. Alle Dateien wären unwiederbringlich verloren. SDOS verwendet derzeit einen manuell erstellten Schlüssel für den root-Knoten. Wie bereits erwähnt, ist es sinnvoll diesen in einem HSM zu speichern. Gleichzeitig sollten weitere Backups dieses Schlüssels existieren.
- Hashing
  SDOS verwendet ausschließlich das ASCII-Verfahren für das Hashing. Es wurden zwei weitere Verfahren vorgestellt, die jeweils Vor- und Nachteile gegenüber dem jetzigen Verfahren besitzen. Obwohl die gleichzeitige Verwendung mehrerer Hashfunktionen nicht empfohlen wird, ist zumindest die Wahl alternativer Hashfunktionen sinnvoll. Abhängig der Dateinamenstruktur kann der Benutzer selbst entscheiden, welche Hashfunktion die beste Lösung ist. Bei gleichzeitiger Verwendung muss ein weiteres Verwaltungslevel hinzugefügt werden, welches angibt, für welche Datensätze welche Hashfunktion verwendet wurde. Für Datensätze, deren Dateinamen Buchstaben außerhalb der ASCII-Tabelle liegen, ist SDOS derzeitig nicht verwendbar - so auch für deutsche Datensätze (Umlaute ä,ü,ö).
- Datenstruktur
  In der jetzigen Datenstruktur ist vorgesehen, dass alle Knoten im Container *NodeContainer* und alle Dateien in *DataContainer* gespeichert werden. Unterordner innerhalb der Container werden nicht unterstützt. Bei großen Datenmengen ist ein komplexeres Dateisystem sinnvoll, wenn nicht sogar erforderlich. Eine Erweiterung der Implementierung, um beim Einfügen von Dateien einen anderen Container bzw. Unterordner zu spezifizieren, wäre denkbar. Zusätzlich müsste jeder Knoten eine weitere Liste (bspw. *location*) besitzen, die den Container/Unterordner angibt. Dies ist erforderlich, da durch das Hinzufügen weiterer Container/Unterordner gleiche Dateinamen vorhanden sein könnten.
- *OpenStack*
  Der Prototyp von SDOS funktioniert nur in Kombination mit OpenStack Swift. Die bereitgestellte Bibliothek vereinfacht das Erstellen einer API. Der größte Anbieter von IaaS ist derzeit allerdings Amazon mit EC2. Das Einfügen der entsprechenden Schnittstellen für EC2 sowie weitere Cloud Software (Eucalyptus, OpenNebula,...) würde den potentiellen Einsatzbereich von SDOS erhöhen.

- Maximale Tiefe Die Evaluation hat gezeigt, dass bei großen Datenmengen maximale Tiefen von bis zu 100 entstehen können. Sollten in dieser Tiefe Dateien referenziert sein, die sehr häufig verwendet werden, entsteht ein erheblicher Aufwand. Der vorgestellte Prototyp fügt die Dateien unabhängig von ihrer Priorität in die Datenstruktur an. Eine simple Lösung wäre es, die am häufigsten verwendeten Dateien als erstes in die Datenstruktur einzufügen.

7 Zusammenfassung

Das sichere Löschen wird in Zukunft mehr und mehr Nachfrage erhalten. Durch die Durchsetzung von Technologien wie SSDs und vor allem Cloud Computing sind ältere Verfahren zum sicheren Löschen nicht mehr ausreichend. Neue Technologien wie SSDs und Cloud Computing haben in diesem Bereich einen Paradigmenwechsel stattfinden lassen. Der Benutzer ist sich nicht über die Anzahl weiterer Kopien bewusst und im Falle von Cloud Computing besitzt der Benutzer nicht einmal die Kontrolle über seine Daten. Deshalb wächst die Anzahl der Verfahren, die Verschlüsselung zum sicheren Löschen verwenden. Bei SSDs wurde ein Verfahren vorgestellt, dass erzwingt, dass alle Object Header, in welcher der Schlüssel gespeichert wird, im selben block gespeichert werden. Beim Löschen wird der gesamte block gelöscht, wodurch alle Schlüssel entfernt werden. Auch Cloud Anbieter sind sich bewusst, dass Benutzer mehr Sicherheit und Integrität ihrer Daten wünschen. Amazons CloudHSM ermöglicht das sichere Löschen von der Anbieterseite aus, allerdings mit zusätzlichen Kosten. Ein weiterer Vorteil bei diesen Verfahren ist allgemein die Tatsache, dass die Daten nur verschlüsselt vorliegen. So wird jede Art unauthorisierter Zugriffe verhindert.

Immer größere Datenmengen verlangen eine sinnvolle Struktur, die ein schnelles und sicheres Löschen auch bei einer großen Anzahl von Schlüsseln ermöglichen. Neben anderen Ansätzen wie [LHC+08] und [Kun93] wurde in SDOS vor allem Wert auf eine Datenstruktur gelegt, die eine enorm hohe Zahl von Schlüsseln effizient verwalten kann. Kapitel 4 zeigte das grobe Konzept und Vorgehensweise bei den einzelnen Operationen. Im darauffolgenden Kapitel wurde die Implementierung dieser Konzepte betrachtet. Zugleich wurde in dieser Arbeit versucht, weitere Verfahren und Erweiterungen vorzustellen, um SDOS weiterzuentwickeln. Zuletzt wurde eine Evaluation erstellt, die aufzeigt, wie schnell die maximale Baumtiefe mit der Anzahl von eingefügten Dateien wächst. Es hat sich gezeigt, dass auch bei sehr großen Datenmengen die durchschnittliche Tiefe der Bäume nicht höher war als bei kleinen Datenmengen. Ausschlaggebend dafür war die Struktur der Dateinamen bzw. die verwendete Hashfunktion von SDOS.

Literaturverzeichnis

[ADKR11] M. Ardelt, F. Dölitzscher, M. Knahl, C. Reich. Sicherheitsprobleme für IT-Outsourcing durch Cloud Computing. *HMD Praxis der Wirtschaftsinformatik*, 48(5):62–70, 2011. (Zitiert auf den Seiten 8 und 11)

[ALV] Apache License, Version 2.0. URL http://www.apache.org/licenses/LICENSE-2.0.html. (Zitiert auf Seite 12)

[AW] Amazon Web Services. URL http://aws.amazon.com/de/. (Zitiert auf Seite 16)

[AWC] AWS CloudHSM. URL http://aws.amazon.com/de/cloudhsm/. (Zitiert auf Seite 16)

[BKR11] A. Bogdanov, D. Khovratovich, C. Rechberger. Biclique cryptanalysis of the full AES. In *Advances in Cryptology–ASIACRYPT 2011*, S. 344–371. Springer, 2011. (Zitiert auf Seite 13)

[Cad11] T. Caddy. FIPS 140-2. *Encyclopedia of Cryptography and Security*, S. 468–471, 2011. (Zitiert auf Seite 16)

[CGJ+09] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, J. Molina. Controlling data in the cloud: outsourcing computation without outsourcing control. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, S. 85–90. ACM, 2009. (Zitiert auf Seite 11)

[CSt] Understanding the Cloud Computing Stack: SaaS, PaaS, IaaS. URL http://www.rackspace.com/knowledge_center/whitepaper/understanding-the-cloud-computing-stack-saas-paas-iaas. (Zitiert auf Seite 10)

[DCS] Direction Centrale de la Sécurité des Systemés. URL http://www.ssi.gouv.fr/. (Zitiert auf Seite 16)

[ec2] URL http://aws.amazon.com/de/ec2/. (Zitiert auf Seite 8)

[Euc] Eucalyptus. URL https://www.eucalyptus.com/. (Zitiert auf Seite 27)

[Geh] P. Gehrt. Amazon Elastic Compute Cloud. (Zitiert auf Seite 12)

[HMMG10] O. Hogan, S. Mohamed, D. McWilliams, R. Greenwood. The Cloud Dividend: Part One, The economic benefits of cloud computing to business and the wider EMEA economy, France, Germany, Italy, Spain, and the UK. *Centre for Economics and Business Research*, 2010. (Zitiert auf Seite 12)

[KSSL06] R. Kissel, M. Scholl, S. Skolochenko, X. Li. Guidelines for media sanitization. *NIST Special Publication*, 800:88, 2006. (Zitiert auf Seite 15)

[Kun93] K. C. Kung. Secure file erasure, 1993. US Patent 5,265,159. (Zitiert auf den Seiten 15 und 45)

[LHC+08] J. Lee, J. Heo, Y. Cho, J. Hong, S. Y. Shin. Secure deletion for NAND flash file system. In *Proceedings of the 2008 ACM symposium on Applied computing*, S. 1710–1714. ACM, 2008. (Zitiert auf den Seiten 15 und 45)

[Liu12] H. Liu. Amazon data center size, 2012. URL http://huanliu.wordpress.com/2012/03/13/amazon-data-center-size/. (Zitiert auf Seite 12)

[McC02] M. T. McCarthy. USA Patriot Act, 2002. (Zitiert auf Seite 11)

[MG11] P. Mell, T. Grance. The NIST definition of cloud computing. 2011. (Zitiert auf den Seiten 10 und 11)

[Mor10] T. P. Morgan. NASA and Rackspace open source cloud fluffer, 2010. URL http://www.theregister.co.uk/2010/07/19/nasa_rackspace_openstack/. (Zitiert auf Seite 12)

[Na] URL http://www.nasa.gov/. (Zitiert auf Seite 12)

[NI] National Institure of Standards and Technology. URL http://www.nist.gov/. (Zitiert auf Seite 11)

[ON] OpenNebula. URL http://opennebula.org/. (Zitiert auf Seite 27)

[OpS] OpenStack: The Open Source Cloud Operation System. URL http://www.openstack.org/software/. (Zitiert auf Seite 12)

[OS] URL http://www.openstack.org/. (Zitiert auf Seite 12)

[Pub01] N. F. Pub. 197: Advanced encryption standard (AES). *Federal Information Processing Standards Publication*, 197:441–0311, 2001. (Zitiert auf Seite 13)

[PyC] PyCrypto - The Python Cryptography Toolkit. URL https://www.dlitz.net/software/pycrypto/. (Zitiert auf Seite 28)

[RaS] URL http://www.rackspace.com/de/. (Zitiert auf Seite 12)

[rs] URL http://www.rackspace.com/cloud/. (Zitiert auf Seite 10)

[Sha03] J. S. Shapiro. Understanding the Windows EAL4 evaluation. *Computer*, 36(2):103–105, 2003. (Zitiert auf Seite 16)

[SN] SafeNet. URL http://www.safenet-inc.de/?LangType=1031. (Zitiert auf Seite 16)

[Sus11] L. Sustek. Hardware Security Module. In *Encyclopedia of Cryptography and Security*, S. 535–538. Springer, 2011. (Zitiert auf Seite 16)

[WGSS11] M. Y. C. Wei, L. M. Grupp, F. E. Spada, S. Swanson. Reliably Erasing Data from Flash-Based Solid State Drives. In *FAST*, Band 11, S. 8–8. 2011. (Zitiert auf den Seiten 8 und 9)

Alle URLs wurden zuletzt am 14. 09. 2014 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift