REDS

reclaim your data

K O N Z E P T

Torben Haase, Flowy Apps

– E N T W U R F –

Stand: 14.02.2016

Das vorliegende Dokument beschreibt den aktuellen Stand der Entwicklung von REDS.

Es dient als Diskussionsgrundlage für die weitere Entwicklung.

Die hier beschriebenen Strukturen und Abläufe können sich in späteren Versionen grundlegend verändern, so dass dieses Dokument nicht als Grundlage für die Entwicklung von konkreten Implementierungen von REDS verwendet werden sollte.

Inhalt

1 Abstract

Das in diesem Dokument beschriebene Konzept REDS ermöglicht die Implementierung von Cloud-Anwendungen, bei dem der Dienstanbieter den Datenzugriff kontrollieren kann, ohne selbst Zugriff auf die Daten der Nutzer zu haben. Die Anwendungsdaten werden dabei nicht im Kontrollbereich des Anwendungsanbieters gespeichert, sondern in einem vom Nutzer frei wählbaren Datenspeicher.

REDS erlaubt die anonyme Nutzung von Cloudanwendungen, ohne ein Vertrauensverhältnis mit dem Anbieter eingehen zu müssen und unterbindet effektiv die Datenakkumulation durch einzelne Anbieter. Möglich wird dies durch das verteilte Speichern der Daten im Kontrollbereich des Nutzers und die Verschlüsselung der Informationen, solange sich diese im Zugriffsbereich des Anbieters befinden. Zu diesem Zweck wird das übliche Modell zur Datenspeicherung in die drei unabhängigen Komponenten Pod (Daten-Server, Speicher), Node (App-Server, Zugriffskontrolle) und Leaf (App-Client, Verarbeitung) aufgebrochen, von denen zur Laufzeit nur der Node durch den Anbieter kontrolliert wird. Der Datenaustausch zwischen App-Client und Daten-Speicher erfolgt verschlüsselt. Dabei ist REDS auf kein bestimmtest Verschlüsselungsverfahren beschränkt und außerdem unabhängig von sogenannten Trusted-Thrid-Parties.

In diesem Dokument werden die für die Kommunikation zwischen den Komponenten notwendigen Schnittstellen und Protokolle beschrieben. Das ursprüngliche Konzept basiert auf der Bachelorthesis von Torben Haase.[a] Im Jahr 2013 wurde REDS durch das Projekt „Flowy Tasks“ an der FH-Flensburg durch Torben Haase und Annika Schulz weiterentwickelt.

a Einsicht auf Anfrage (torben@flowyapps.com)

2 Einführung

2.1 Problem

Als 1962 am MIT mit der Entwicklung des ARPANET, dem Vorläufer des heutigen Internets, begonnen wurde, war eine der zentralen Eigenschaften dieses neuartigen Netzwerkes seine dezentrale Organisationsstruktur. Im Zuge des Web 2.0 hat sich das Internet jedoch immer mehr von einem Präsentations- zu einem Dienstleistungsnetzwerk entwickelt, in dem Informationen nicht nur an die Websitebesucher ausgeliefert, sondern von diesen auch aktiv generiert werden. Generell sammeln alle cloudbasierten Anwendungen (auch „Cloud-Apps" oder kurz „Apps") Daten ihrer Nutzer an einer zentralen Stelle unter der Kontrolle des App-Anbieters.[1] Diese Akkumulation von Daten ist für die Nutzer einer Cloudanwendung in zweierlei Hinsicht problematisch:

1.) Die dezentrale Struktur des Internets fördert die Unabhängigkeit des Netzes, da keine zentrale Stelle existiert, welche den Zugriff auf Informationen kontrollieren könnte. Da heutzutage jedoch ein Großteil der vorhandenen Informationen unter der Kontrolle einzelner App-Anbieter stehen, können diese die Verbreitung von Informationen steuern und gegebenenfalls sogar unterbinden. Dabei muss die Zensur nicht zwangsläufig aus Eigeninteresse erfolgen, sondern kann auch durch externen Druck erzwungen werden.[2]

2.) Die Anbieter einer Cloudanwendung haben vollen Zugriff auf alle Informationen, die in der Anwendung verarbeitet werden. Nutzer müssen also dem Anbieter vertrauen, dass dieser in Ihrem Sinne mit ihren Daten umgeht. Dieses Vertrauen ist jedoch nicht immer angebracht, da nicht nur der Anbieter sondern auch Dritte Nutzerdaten lesen können, sobald diese sich einmal Zugriff auf die Datenspeicher des Anbieters verschafft haben.[3]

Für App-Anbieter hat eine zentrale Datenspeicherung hingegen eine Reihe von Vorteilen. In erster Linie ist hier natürlich die vergleichsweise einfache Implementierung zu nennen. Aber gerade für kostenpflichtige Dienste ist es außerdem essentiell, dass sie den Zugriff auf die Daten kontrollieren können. Nachteile, wie die zusätzlichen Kosten durch die Bereitstellung der Speicherinfrastruktur und die potentielle Fehleranfälligkeit einer zentralen Organisationsstruktur, werden dabei durch die Notwendigkeit der Zugriffskontrolle aufgewogen.

Zusammengefasst lässt sich sagen, dass das derzeit vorherrschende Konzept der zentralen Datenspeicherung eine Reihe von Problemen beinhaltet. Insbesondere ist hier der volle Informationszugriff des Anbieters und der damit einhergehende Kontrollverlust des Benutzers zu nennen. Diese Schwächen können durch eine dezentrale Datenspeicherung in Verbindung mit einer entsprechenden Verschlüsselung umgangen werden.

2.2 Anforderungen

Aus den oben skizzierten Problemen bei der Verarbeitung von Daten in der Cloud ergeben sich eine Reihe von Anforderungen an mögliche Lösungsansätze. Der Fokus liegt dabei vor Allem auf dem Schutz der Informationen vor fremdem Zugriff.

Insbesondere darf der App-Anbieter zu keinem Zeitpunkt Zugriff die Nutzerdaten haben. Dies umfasst neben dem Schutz der Inhalte auch Abwehrmaßnahmen gegen Datenmanipulation. Des weiteren sollte es Nutzern der Anwendung möglich sein, diese anonym zu verwenden. Trotz dieser Sicherheitsanforderungen

darf die Handhabung für den Nutzer jedoch nicht komplizierter als die einer herkömmlichen App sein. Vor allem sollte die Verschlüsselung des Datenverkehrs keine Aktionen auf Nutzerseite erfordern, da die Mehrheit der Nutzer mit derartigen Konzepten nicht vertraut ist und den Mehraufwand scheut.[4]

Auf der anderen Seite muss der Anbieter einer Anwendung die Möglichkeit besitzen, eine Zugriffskontrolle auszuüben, um beispielsweise Dienste auf Basis eines monatlichen Abonnements anbieten zu können. Dazu gehört insbesondere der Schutz vor der unerlaubten Verwendung von Nutzerdaten in anderen Anwendungen.

Konkrete Implementationen sollten sich auf offene und anerkannte Standards stützen und Datenstrukturen verwenden, die sowohl von Mensch als auch von Maschine lesbar sind. Des Weiteren sollten sie als Open-Source umgesetzt werden. Dies ist dabei nicht nur ein kosmetischer Aspekt, sondern verbessert essentiell die Sicherheit der Implementierung, da diese so durch unabhängige Dritte überprüft werden kann, wodurch Fehler schneller gefunden werden und das Vertrauen in die Implementierung insgesamt gesteigert wird.

2.3 Lösungsskizze

Die derzeit dominierende Ansicht ist, dass sich der Wert eines Dienstes vor allem über die von ihm angesammelten Daten definiert. Der Wert eines Dienstes erschließt sich jedoch in den meisten Fällen nicht aus den Informationen in den Daten, sondern vielmehr aus den Relationen zwischen den einzelnen Datensätzen, der Präsentation der Informationen auf der Website und aus den Verknüpfungen zwischen seinen Benutzern.

Herkömmliche Cloudanwendungen verarbeiten viele Informationen auf dem App-Server im Kontrollbereich des Anbieters, wohingegen der App-Client (z.B. die Website im Browser des Nutzers) nur die Visualisierung der Daten übernimmt. Da Informationen jedoch nur auf Clientseite verfügbar sein sollen, ist ein Bruch mit dem Thin-Client/Fat-Server Dogma notwendig, an dessen Stelle eine Fat-Client/Thin-Server Architektur treten muss. Der App-Server stellt hierbei nur die Daten zur Verfügung, welche dann vom App-Client bearbeitet werden.

Des Weiteren ist es notwendig, dass die Daten zwar verschlüsselt übertragen aber unverschlüsselt gespeichert werden. Denn nur so ist es möglich, die angefragten Datensätze nach bestimmten Gesichtspunkten zu filtern (z.B. alle Adressen mit einem bestimmten Nachnamen). Es gibt zwar Lösungsansatze für die Durchsuchbarkeit von verschlüsselten Daten, diese sind derzeit jedoch aufgrund von Performanceproblemen nicht in der Praxis einsetzbar. Da die unverschlüsselte Speicherung der Daten auf dem App-Server, also unter der Kontrolle des Anbieters, keine Option ist, müssen die Inhalte auf einem Daten-Server im Zugriffsbereich des Nutzers gespeichert werden. Dennoch ist es wichtig, dass Relationen über die Grenzen des Daten-Servers auf Nutzerseite erstellt werden können, um globale Verknüpfungen von Datensätzen auf verschiedenen Daten-Servern zu ermöglichen.

Daraus ergibt sich folgendes Szenario: Auf Seiten des Anbieters werden nur die Relationen zwischen verschiedenen Datensätzen gespeichert, wohingegen die Inhalte der Datensätze auf einem Daten-Server auf Nutzerseite gespeichert werden. Um einen Datensatz zu bearbeiten, sendet der App-Client eine verschlüsselte und signierte Anfrage an den App-Server. Dieser leitet die Anfrage anhand der Relationen an den entsprechenden Daten-Server weiter. Auf dem Daten-Server werden die Zugriffsrechte überprüft, die Anfrage ausgeführt und die verschlüsselte und signierte Antwort über den App-Server zurück an den App-Client gesendet. Dort wird die Signatur geprüft sowie der Datensatz entschlüsselt und weiterverarbeitet.

Der App-Server ist demnach als zentraler Knotenpunkt unverzichtbar. Der Anbieter behält so die Kontrolle über die Relationen, wohingegen der Nutzer die Inhalte auf dem Daten-Server kontrolliert. Da der

Datentransfer zwischen App-Client und Daten-Server verschlüsselt erfolgt, hat der Anbieter keinen Zugriff auf die Informationen. Der Nutzer muss also nur dem Besitzer des Daten-Servers, der er im Idealfall selbst ist, vertrauen, um die durch den Anbieter bereitgestellte Anwendung nutzen zu können.

2.4 Stand der Technik

Derzeit ist den Entwicklern von REDS kein System bekannt, welches die im vorherigen Kapitel beschriebenen Anforderungen erfüllt. Dennoch gibt es auch jetzt schon einige Entwicklungen, welche zumindest einen Teil der Anforderungen umsetzen. Drei dieser Technologien, welche jeweils exemplarisch für eine größere Gruppe ähnlicher Lösungen stehen, sollen hier im Einzelnen kurz erläutert werden.

2.4.1 Lokaler Cloud-Speicher: ownCloud

ownCloud[a] ist eine open-source Software, die Dienste zur Datensynchronisation und zum gemeinsamen Verwenden von Daten bereitstellt. ownCloud wurde in PHP und JavaScript geschrieben und muss daher auf einem eigenen Webserver installiert werden. Neben den typischen Diensten, wie Kalender, Adressbuch und File-Sharing, können weitere Funktionen durch Apps von externen Entwicklern ergänzt werden.

Lokale Cloud-Lösungen wie ownCloud lösen zwar das Datenschutzproblem, da die Kontrolle über die Daten im Bereich des Nutzers liegt, allerdings setzt die Administration einer lokalen Cloud Fachwissen voraus, das bei den meisten Nutzern nicht vorhanden ist. Zudem verliert der App-Anbieter nicht nur die Kontrolle über die Informationen, sondern auch über die Zugriffsrechte. Freemium-Dienste oder monatliche Abos sind bei Apps in der lokalen Cloud also nicht möglich.

2.4.2 Verteilte soziale Netzwerke: Friendica Red Matrix

Friendica[b] ist ein verteiltes soziales Netzwerk, bei dem sich mehrere unabhängige Knotenpunkte zu einem sozialen Netzwerk verbinden. Grundsätzlich kann dabei jeder einen Knotenpunkt auf seinem eigenen Webserver aufsetzen und so die Kontrolle über seine Daten behalten. Die Red Matrix stellt die nächste Entwicklungsstufe von Friendica dar. Sie ist jedoch in ihrer grundlegenden Struktur völlig anders als REDS, da Programm und Daten hier nicht getrennt werden.

Aufgrund der verteilten Struktur ergeben sich die gleichen Probleme wie bei lokalen Cloud Lösungen: Webapp-Anbieter geben nicht nur die Daten, sondern die gesamte Anwendung in die Kontrolle der Nutzer und die Nutzer müssten die Anwendung wiederum erst auf ihrem eigenen Server installieren.

2.4.3 Verschlüsselter Cloud-Speicher: SpiderOak Crypton

Crypton[c] ist ein Framework für die Entwicklung von Zero-Knowledge Anwendungen, welches von SpiderOak[d], einem der größten Anbieter für verschlüsselten Cloud-Speicher, entwickelt wurde. Crypton stellt im Kern eine Objekt-Datenbank dar, deren Daten-Objekte verschlüsselt gespeichert werden[e]. Mit diesem Prinzip kommt Crypton von allen vorgestellten Ansätzen am nächsten an das REDS-Konzept heran.

Allerdings werden die Daten bei Crypton zentral gespeichert, so dass der Anbieter weiterhin die Speicher-Infrastruktur bereitstellen. Daraus folgt unmittelbar, dass es es dem Nutzer nicht möglich ist die Daten zu rekonstruieren, sobald die App, z.B. durch Serverausfall oder Passwortverlust, nicht mehr erreichbar ist. Des

a http://owncloud.org
b http://friendica.com
c https://crypton.io
d https://spideroak.com
e https://crypton.io/developer-guide

Weiteren liegen die Daten nur in verschlüsselten Containern vor, welche erst komplett auf den App-Client übertragen werden müssen, bevor mit ihnen gearbeitet werden kann. Die serverseitige Durchsuchbarkeit fällt also weg.

3 REDS Grundlagen

3.1 Ziele

REDS ist eine Implementierung der in Abschnitt 2.2 skizzierten Lösung. REDS steht dabei für „Remotely Encrypted Distributed Storage“ und spiegelt so die zentralen Aspekte des vorgestellten Lösungsansatzes wider. Im einzelnen sind dies:

- **Schutz von Nutzerdaten:** Die Daten der Anwendungsnutzer sind zu keinem Zeitpunkt für den Anwendungsanbieter einsehbar oder manipulierbar. Systembedingt ist der Anbieter jedoch in der Lage den Zugriff auf einzelne Daten zu blockieren.
- **Anonymität des Nutzers:** Der Anwendungsanbieter kann zu keinem Zeitpunkt die tatsächliche Person hinter einem Nutzeraccount ermitteln. Es sei denn der Nutzer hat dem Anbieter ausserhalb von REDS persönliche Informationen übermittelt (z.B. bei einem Bezahlvorgang).
- **Datenhoheit beim Nutzer:** Durch die Speicherung von Daten auf Nutzerseite ist dieser in der Lage die Daten auch ausserhalb der eigentlichen Anwendung weiterzuverwenden. Insbesondere hat er so auch nach Abschaltung der Anwendung weiterhin Zugriff auf die Daten.
- **Zugriffskontrolle durch den Anbieter:** Der Anwendungsanbieter kann einzelnen Accounts den Zugriff auf die gesamte Anwendung oder einzelne Daten gewähren oder verweigern. Allerdings ist er nicht in der Lage Daten zu ändern oder zu löschen.
- **Unabhängigkeit von Dritten:** REDS-Anwendungen benötigen weder auf Nutzer- noch auf Anbieterseite einen vertrauenswürdigen Dritten für die Authentifizierung von Schlüsseln o. Ä.

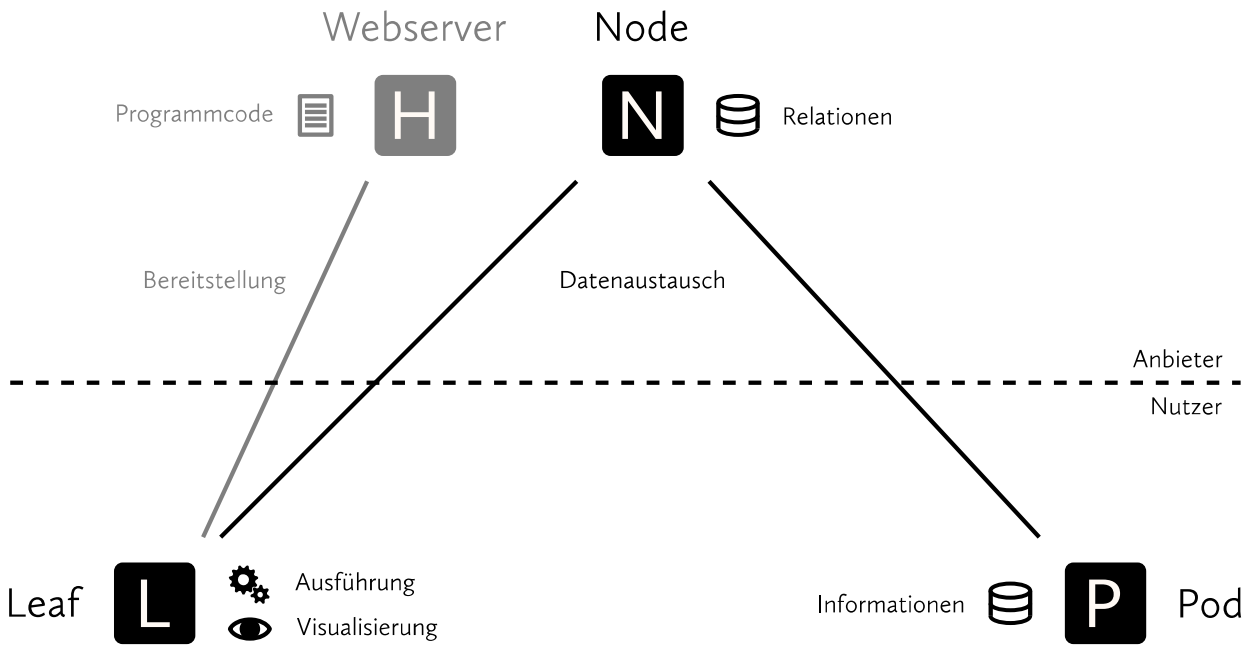
Dazu kommen weitere Aspekte, welche in erster Linie durch die konkreten Implementierungen umgesetzt werden müssen:

- **Verteilte Datenspeicherung:** Der Anwendungsnutzer kann frei wählen wo er seine Daten speichern möchte. Die Speicherung erfolgt idealerweise im Kontrollbereich des Nutzers oder alternativ bei einem vertrauenswürdigen Dritten.
- **Offene Implementierung:** Der Quellcode der REDS Implementierung ist vollständig einsehbar. Eingabe- und Ausgabedaten von Anwendungen, die REDS verwenden, können auf ihre Korrektheit überprüft werden, um korrumpierte REDS Implementierungen zu erkennen.
- **Offene Plattform:** REDS ist kein geschlossenes „Ökosystem“, welches von Einzelnen oder Wenigen kontrolliert wird. REDS Anwendungen können auf allen gängigen Computerplattformen von jedem frei entwickelt und vertrieben werden.

3.2 Terminologie

Da REDS in erster Linie für die Verwendung mit Webapps entwickelt wurde, wird im Folgenden vor allem dieses Szenario berücksichtigt. Das zugrundeliegende Konzept kann aber ausdrücklich auch auf andere Software-Anwendungen mit einer Client-Server Organisation übertragen werden.

3.2.1 Leaf, Node & Pod



Die obige Abbildung gibt eine Gesamtübersicht über die Struktur von REDS. Der Webserver ist kein direkter Teil von REDS, sondern wird nur bei Webapps für die Bereitstellung des Programmcodes gebraucht. Wie bereits in der Lösungsskizze beschrieben, verteilt sich auch REDS auf drei Komponenten:

- **Leaf:** Vereinfacht ist dies das Programm mit der der Nutzer aktuell arbeitet. Genauer betrachtet besteht der Programmcode in der Regel aus einem großen Anwendungsteil, der von REDS unabhängig ist, und einem kleinen Modul, das die Kommunikation mit dem Node übernimmt. In dieser Sichtweise wird der Anwendungsteil als App und das Modul als Leaf bezeichnet. App und Leaf sind weitestgehend unabhängig voneinander und kommunizieren über klar definierte Schnittstellen miteinander. Die Ausführung des Leafs erfolgt auf dem Endgerät des Nutzers.
- **Node:** Der App-Server, welcher Anfragen des Leafs entgegen nimmt. Zur jeder App gehört immer genau ein Node, der als zentrales Verzeichnis für die Accounts, Domains und Entities (→ Kapitel 3.4) dient und die Zugriffskontrolle übernimmt. In der Regel werden die Leaf-Anfragen vom Node an den Pod weitergeleitet, um dort weiterverarbeitet zu werden. Der Node befindet sich komplett in der Kontrolle des App-Anbieters. Die Ausführung erfolgt auf dem Server des App-Anbieters.
- **Pod:** Der Daten-Server. Der Pod empfängt über den Node die Anfragen des Leaf und speichert die Inhalte der Datensätze. Diese werden auf dem Pod für jede App in einer individuellen Sandbox gespeichert. Ein Nutzer kann mehrere Pods nutzen, um dort verschiedene Datensätze zu speichern. Der Pod befindet sich komplett in der Kontrolle des Nutzers oder eines vertrauenswürdigen Dritten. Ausführung erfolgt auf einem Computer des Nutzers (z.B. NAS oder REDS.box).

Vereinzelt werden die Begriffe Leaf, Node und Pod auch verwendet, um die Computersysteme auf denen die jeweilige Software ausgeführt wird zu bezeichnen. Die Aussage „wird auf dem Pod gespeichert“ ist demnach als „wird auf dem Computersystem, auf dem der Pod ausgeführt wird, gespeichert“ zu verstehen.

3.2.2 Account, Domain, Entity & Ticket

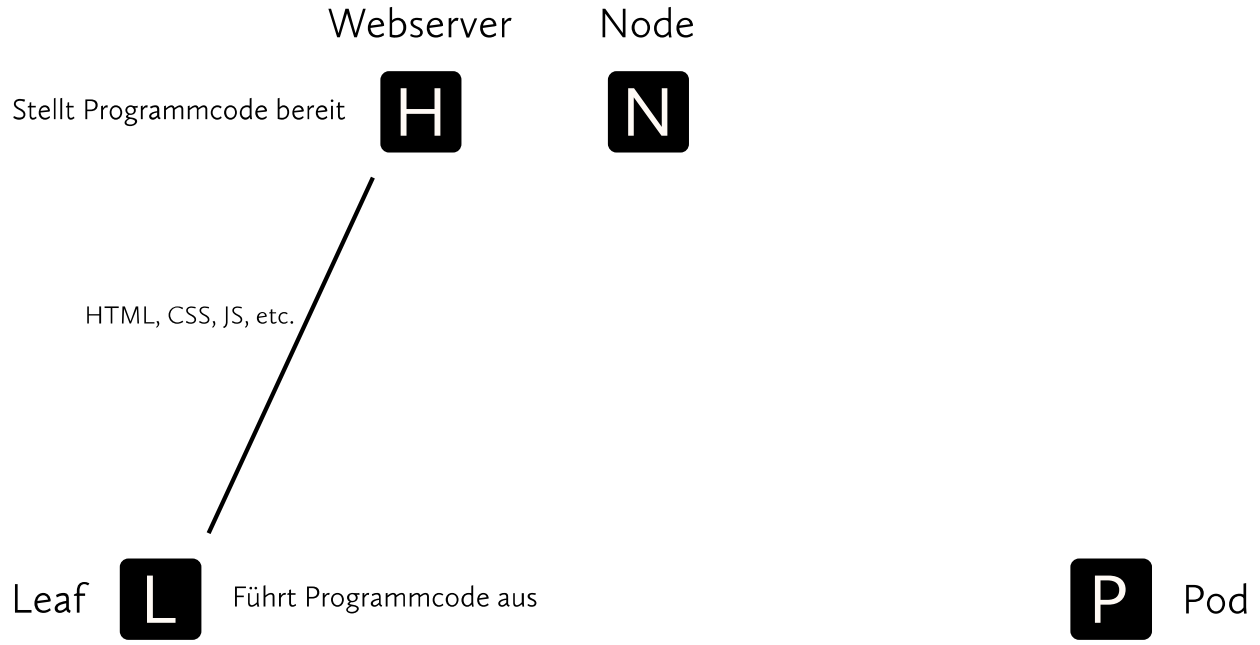
- **Account:** Zu jedem User gehört genau ein Account. Dieser dient zur verschlüsselten Speicherung des Keyrings auf dem Node. Im Klartext werden nur ein Pseudonym, eine ID und der

Authentifizierungsschlüssel gespeichert. Pseudonym und ID sind nodeweit eindeutig. Der Keyring liegt auf dem Node in verschlüsselter Form vor und ist daher nur für den Nutzer selbst zugänglich.

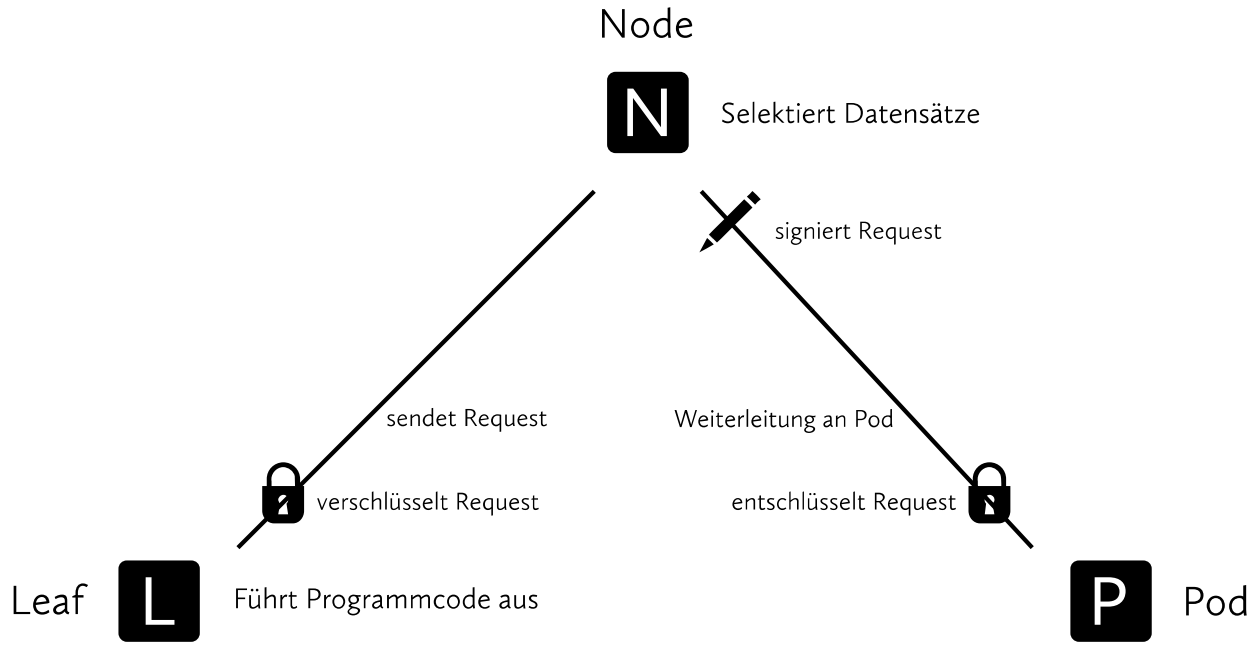
- **Domain:** Datensätze werden in Domains gruppiert, über welche die Zugriffsrechte definiert werden. Eine Domain hat eine nodeweit eindeutige ID und gehört immer zu genau einem Pod. Daher befinden sich alle Datensätze einer Domain auf dem gleichen Pod. Auf dem Node wird nur die ID gespeichert; Die eigentlichen Informationen befinden sich auf dem Pod.
- **Entity:** Ein einzelner Datensatz innerhalb einer Domain wird als Entity bezeichnet. Jede Entity gehört einem bestimmten Typ an und hat eine ID. Die Kombination Typ+ID ist nodeweit eindeutig. Abgesehen von Typ und ID werden alle Entitydaten immer auf dem Pod der überordneten Domain gespeichert, also nicht über mehrere Pods verteilt. Es können Eltern-Kind-Relationen zwischen Entities geknüpft werden, um diese so in einer Graphenstruktur zu organisieren. Relationen zwischen Entities werden auf dem Node gespeichert.
- **Ticket:** Definiert die Zugriffsrechte eines Nutzers auf eine Domain. Rechte können nur domainweit und nicht für einzelne Entities vergeben werden. Einer Domain können mehrere Tickets zugeordnet werden, wobei jeder Nutzer ein eigenes Ticket mit einer domainweit eindeutigen ID besitzt. Mit Hilfe von Einladungen können Tickets für andere Nutzer erstellt werden, so dass auch das Teilen von Domaindaten möglich ist.
- **Stream:** Der eigentliche Datentransfer erfolgt in sogenannten Streams. Dieser kann als temporärer verschlüsselter Datentunnel zwischen einem Pod und einem Leaf verstanden werden. Auf einem Leaf können mehrere Streams existieren, um Daten von verschiedenen Domains gleichzeitig anfragen zu können. Ein Stream wird bei Nichtbenutzung nach einem festgelegten Zeitraum geschlossen und die verwendeten Schlüssel zerstört.

3.3 Kommunikation

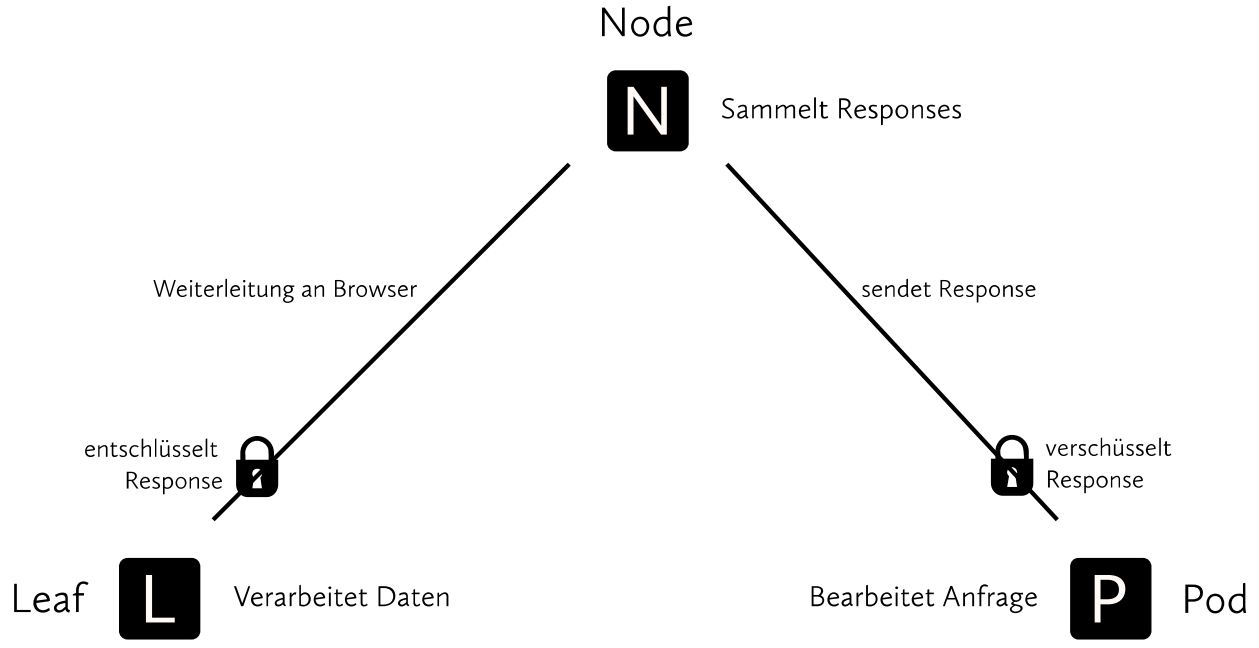
Die folgenden Diagramme beschreiben der Kommunikation zwischen den verschiedenen REDS-Komponenten. Auf Details der Verschlüsselung und Authentifizierung wird bewusst verzichtet, um den allgemeinen Ablauf besser verdeutlichen zu können.



Bei Webapps fordert der Browser zunächst den Programmcode für den Leaf beim Webserver des Anbieters an. Bei nativen Anwendungen entfällt die Anforderung des Programmcodes, da sich dieser bereits auf dem Endgerät des Nutzers befindet. Sobald der Code verfügbar ist, kann der Leaf-Programmcode ausgeführt werden und die Kommunikation mit dem Node beginnen.



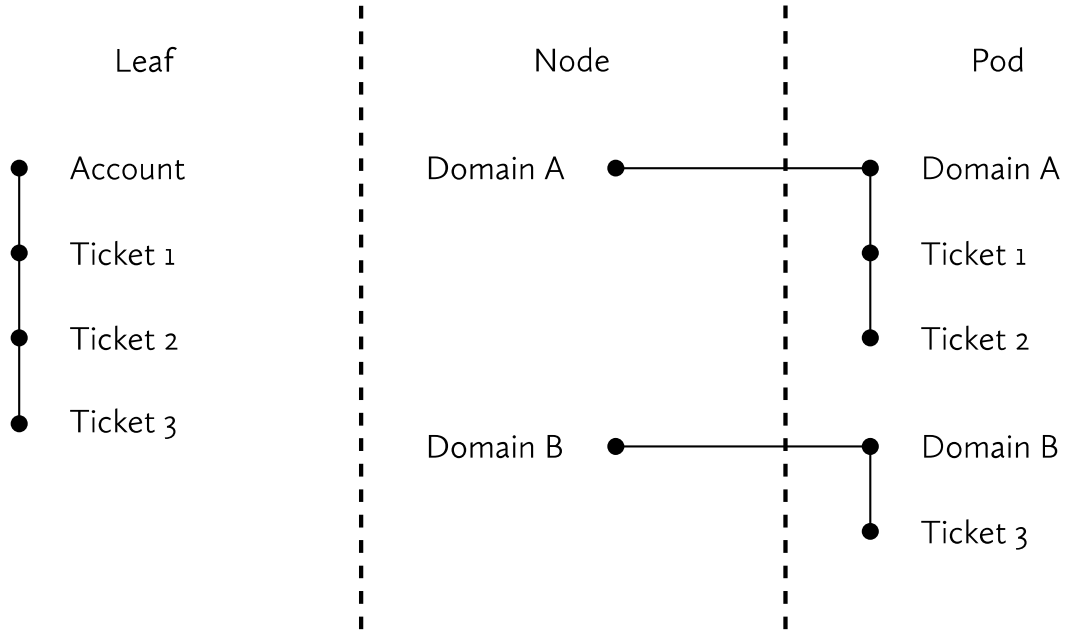
Sobald der Leaf Entities aus einer Domain benötigt, wird eine Anfrage formuliert, und an den Node gesendet. Der Node kann anhand der Domain-ID den Pod, auf dem die Domain liegt, ermitteln und die Anfrage and diesen weiterleiten.



Der Pod verarbeitet die Anfrage und sendet die Antwort zurück an den Node. Dieser sammelt gegebenenfalls die Antworten von mehreren Pods und leitet diese an den Leaf weiter. Dort können die Entities dann in der Anwendung weiterverarbeitet werden.

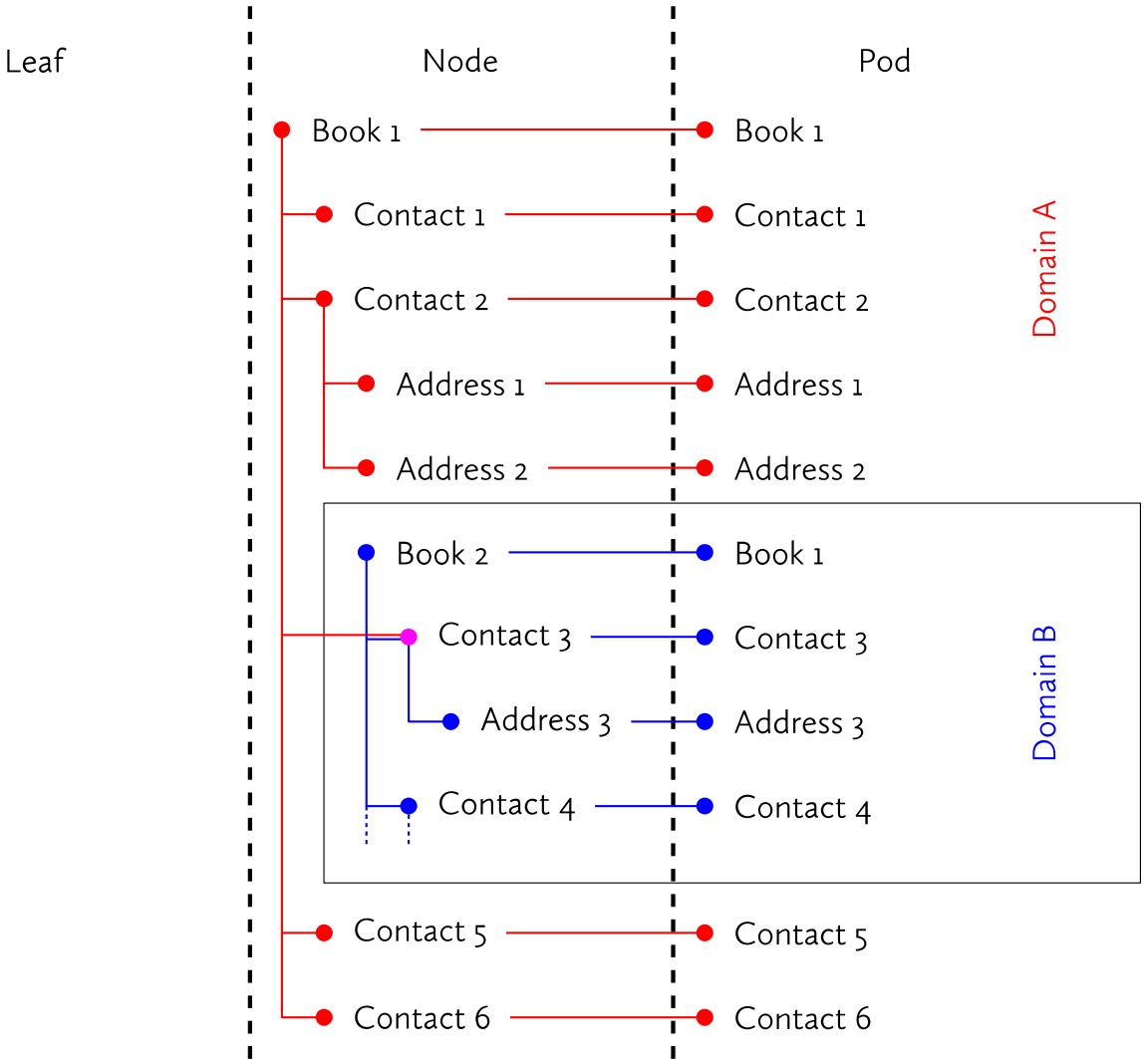
3.4 Datenstruktur

Zentrales Element für die Strukturierung von Daten sind die Domains. Domains fassen Entities zu Gruppen zusammen und definieren auf welchem Pod die Entities gespeichert werden. Die Verwaltung von Domains obliegt dem Node, da Domain-IDs nodeweit eindeutig sein müssen.



Alle relevanten Domaindaten, insbesondere die Entities, werden jedoch auf den Pod gespeichert. Zu jeder Domain gehören außerdem ein oder mehrere Tickets, die ebenfalls auf dem Pod gespeichert werden. Dem Pod ist nur die Verbindung zwischen Domain und Ticket bekannt. Welche Tickets zu welchem Account gehören ist nur für den Accountbesitzer direkt ersichtlich.

Innerhalb einer Domain können beliebig viele Entities beliebigen Typs gespeichert werden. Entities werden auf dem Node verwaltet, das heißt der Node definiert Entitytypen, generiert Enitity-IDs und speichert die Domainzugehörigkeit. Die eigentlichen Entitydaten werden auf dem Pod gespeichert.



Entities können entweder frei innerhalb der Domain platziert werden oder als Kindentity einer bereits vorhandenen Entity in die Datenstruktur eingehängt werden. Eine Entity kann auch durch mehrere Eltern-Kind-Relationen mit anderen Entities verknüpft sein (siehe Abbildung: Contact 3). Relationen zwischen Entities werden auf dem Node gespeichert und verwaltet, um auch Verknüpfungen über Podgrenzen hinweg zu ermöglichen.

3.5 Zugriffskontrolle

Die Zugriffskontrolle erfolgt sowohl auf dem Node, als auch auf dem Pod, wobei beide Instanzen verschiedene Ziele verfolgen. Die Kontrolle auf dem Node erfolgt auf Basis des Nutzeraccounts und ermöglicht es beispielsweise dem App-Anbieter seinen Dienst als Abonnement anzubieten oder bestimmte Funktionen nur nach Kauf einer Premium-Option freizuschalten. Auf dem Pod erfolgt die Kontrolle anhand von Domaintickets. Diese stellt sicher, dass nur berechtigte Nutzer Zugriff auf die Domaindaten haben.

3.5.1 Node

Die Zugriffskontrolle auf dem Node kann je nach Bedarf des App-Anbieters in verschiedenen Abstufungen erfolgen. Da mit der Dichte des Authentifizierungsrasters auch die Informationsdichte beim App-Anbieter steigt, sollte das gewählte Raster immer nur die Authentifizierungen erfordern, welche für das Geschäftsmodell des Anbieters zwingend notwendig sind.

- Das REDS die Authentifizierung des Nutzers beim Zugriff auf Accountdaten erfordert, eigent es sich nicht nur für komplett freie Dienste oder für Anwendungen, bei denen die Bezahlung einmalig bei Erwerb erfolgt. Da der Zugriff auf die Keyringdaten reguliert werden kann, sind auch Abonnements möglich. Der Anbieter gelangt hier nicht an zusätzliche Informationen, da durch die Authentifizierung nur die ohnehin notwendige Account-ID offenbart wird.
- Eine feinere Zugriffskontrolle erlangt der App-Anbieter, wenn das Erstellen von Domains und Einladungen eine Authentifizierung erfordert. Auf diese Weise ist es dem Anbieter möglich das Teilen von Domaindaten mit Anderen für sein Geschäftsmodell zu nutzen. Allerdings ist es dem Anbieter so auch möglich zu erkennen, welche Accounts welche Domains besitzen.
- Die Authentifizierung jeglicher Anfragen (abgesehen von der Nutzeranmeldung) ist die maximale Rasterdichte. Hier hat der Anbieter die gleiche Zugriffskontrolle wie bei einer herkömmlichen Cloudanwendung. Da es so jedoch auch möglich ist zu ermitteln welche Tickets zu welchem Account gehören, ist diese Konfiguration nicht empfehlenswert.

Neben den hier exemplarisch vorgestellten Authentifizierungsrastern sind noch feinere Abstufungen oder Abwandlungen möglich, die hier nicht beschrieben wurden.

3.5.2 Pod

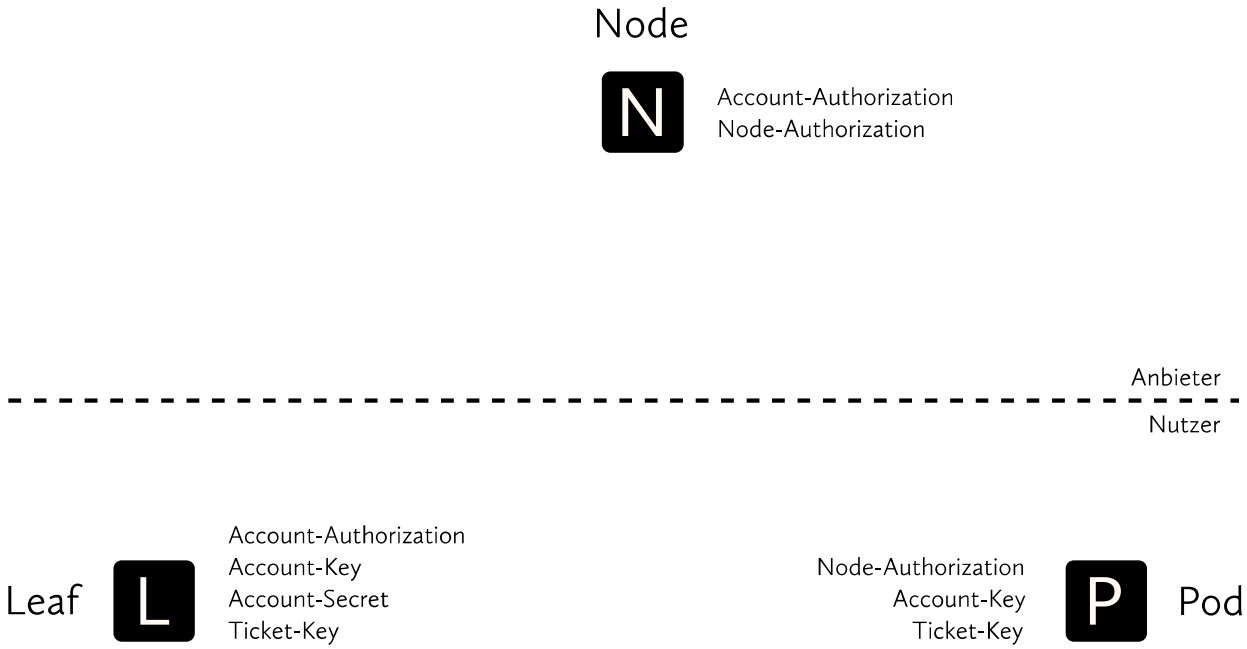
Wie bereits erwähnt, erfolgt die Zugriffskontrolle auf dem Pod auf Domainbasis. Das heißt, es können mit Hilfe von Tickets Zugriffsrechte für alle Entities innerhalb einer Domain definiert werden. Eine feinere Kontrolle für einzelne Entities ist nicht vorgesehen.

Jedem Ticket werden Zugriffsrechte nach dem CRUD-Prinzip zugeordnet. CRUD steht dabei für **c**reate, **r**ead, **u**pdate und **d**elete. Diese Zugriffsrechte entsprechen den Operationen, welche der jeweilige Ticketbesitzer auf die Domaindaten anwenden kann.

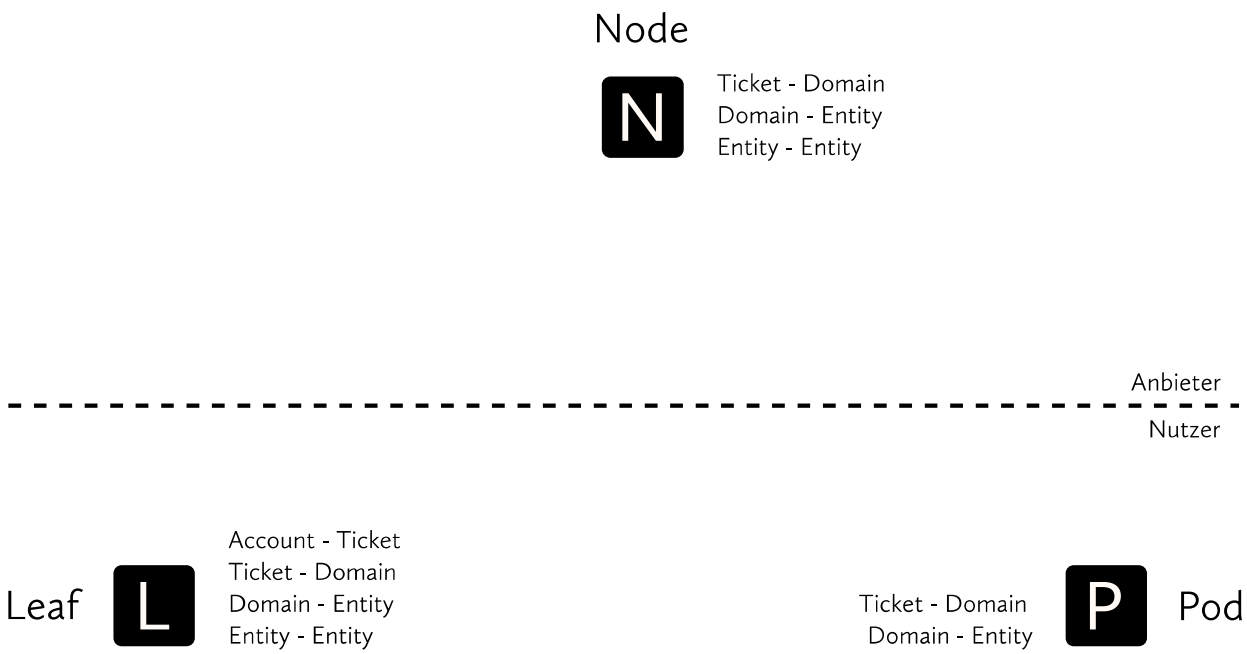
Zusätzlich existieren noch die Rechte **a**dministrate und **o**wn. Das Recht own kennzeichnet das Ticket des Domainbesitzers, welcher immer alle Zugriffsrechte hat. Außerdem kann nur der Besitzer einer Domain diese vollständig löschen. Das Besitzerrecht muss immer genau einmal innerhalb einer Domain vergeben sein, kann jedoch von einem Ticket auf ein anderes übertragen werden. Tickets mit dem Recht administrate dürfen Zugriffsrechte verändern, andere Nutzer in die Domain einladen oder aus der Domain entfernen.

Zu jedem Ticket gehört ein individueller Schlüssel, der für den Aufbau eines verschlüsselten Streams zwischen Endgerät und Pod benötigt wird. So sind selbst andere Domainmitglieder nicht in der Lage, Inhalte während des Transfers abzufangen und zu lesen. So ist sichergestellt, dass vor jeder Anfrage die Zugriffsrechte überprüft werden, was unter anderem die Erstellung von Tickets mit reinem Schreibzugriff auf einer Domain erlaubt.

3.6 Informationsverteilung



Die Schlüsselverteilung zeigt, dass dem Node nur die Authentifizierungsschlüssel bekannt sind. Alle Schlüssel, die für die Verschlüsselung von Daten verwendet werden, sind nur dem Pod und dem Leaf bekannt. Darüber hinaus sind die Stream-Schlüssel für den Datenaustausch zwischen Leaf und Pod nur temporär (→ Kapitel 4.4). Die Kommunikation zwischen Leaf und Pod entspricht also dem Prinzip der Perfect-Forward-Secrecy.



Das Diagramm zeigt, an welche Informationen die einzelnen Komponenten anhand von gespeicherten Daten und der Analyse der ausgetauschten Daten gelangen können. Wichtigstes Merkmal ist dabei, dass es dem Node nicht möglich ist, Verbindungen zwischen Tickets und Accounts oder Domains und Accounts aus den Anfragen abzuleiten. Somit ist es einem Angreifer mit Zugriff auf den Node und eine Domain nicht möglich, Rückschlüsse auf andere Domains oder die Accounts der Domainmitglieder zu ziehen.

3.7 Kryptographie

REDS ist im Wesentlichen eine neue Kombination von bekannten kryptographischen Verfahren. Das heißt, dass keine neuen kryptographischen Algorithmen implementiert werden müssen. Stattdessen kann auf bekannte und erprobte Standardalgorithmen zurückgegriffen werden.

Die Protokollbeschreibung setzt die Existenz einiger kryptographischer Funktionen, insbesondere zum sicheren Schlüsselaustausch und zur symmetrischen Verschlüsselung voraus. Eine wichtige Kenngröße ist dabei die Bit-Länge *n* des für die symmetrische Verschlüsselung notwendigen Schlüssels *key*:

n = length(key)

3.7.1 Schlüssel

key `key()` – Generiert einen zufälligen Schlüssel für die symmetrische Verschlüsselung und gibt diesen zurück.

3.7.2 Privater Schlüsselteil

prikey `private()` – Gibt einen zufälligen privaten Schlüsselteil für die asymmetrische Verschlüsselung zurück.

3.7.3 Öffentlicher Schlüsselteil

pubkey `public(`*prikey* `key)` – Erwartet einen privaten Schlüsselteil als Parameter und generiert daraus einen öffentlichen Schlüsselteil für die asymmetrische Verschlüsselung.

3.7.4 Schlüsselkombination

key `shared(`*prikey* `private,` *pubkey* `public)` – Erwartet einen privaten und einen öffentlichen Schlüsselteil als Parameter und generiert daraus einen Schlüssel für die symmetrische Verschlüsselung. Es gilt:

shared(pri_A, pub_B) == shared(pri_B, pub_A)

Geeignete Implementierungen der Funktionen private(), public() und shared() könnten beispielsweise auf dem Diffie-Hellman Schlüsselaustausch[5] basieren.

3.7.5 Sicherer Hash

key `shash(`*string* `data,` *key* `salt)` – Erwartet einen String von beliebiger Länge und ein Schlüssel als Salt. Der Rückgabewert ist ein sicherer Hash derart, dass dieser auch als Schlüssel verwendet werden kann.

Geeignete Hashfunktionen wären beispielsweise: PBKDF2[6], bcrypt[7] oder scrypt[8].

3.7.6 Authentifizierung

key `hmac(`*string* `data,` *key* `key)` – Erwartet einen String von beliebiger Länge und einen Schlüssel als Parameter. Daraus wird ein hashbasierter Message Authentification Code generiert, der auch als Schlüssel verwendet werden kann.

Geeignete Hashfunktionen wären beispielsweise SHA2[9] oder SHA3[10].

3.7.7 Timestamp

key `time()` – Gibt einen von der aktuellen Zeit *timestamp* abhängigen Schlüssel zurück. Sollte *length(timestamp)* < *n* sein, werden die verbleibenden Bits mit Zufallswerten aufgefüllt. Dies verringert gleichzeitig die Vorhersagbarkeit des Schlüssels und ist daher empfehlenswert. Es muss sichergestellt sein, dass bei einem Vergleich von zwei mittels *time()* generierten Schlüsseln immer festgestellt werden kann, welcher von beiden der jüngere und welcher der ältere ist.

3.7.8 Konkatenation

string `sconc(`*string* `str1,` *string* `str2, …)` – Anders als die übliche String-Konkatenation verknüpft diese Funktion mindestens zwei Strings von beliebiger Länge in einer Weise, so dass gilt:

sconc(a, bc) != scon(ab, c) ∧ size(sconc(a, b)) >= n

Auf diese Weise ist sichergestellt, dass unterschiedliche Eingabestrings immer auch unterschiedliche Ausgabestrings ergeben. Der Rückgabewert ist ein String von mindestens der Länge *n*. Eine einfache Implementierung wäre:

sconc(a, b) = a+x+b

Wobei + die normale Konkatenation ist und das Zeichen *x* weder in String *a* noch in String *b* vorkommt.

3.7.9 Verschlüsselung

string `encrypt(`*string* `data,` *key* `key,` *key* `vector)` – Erwartet einen String von beliebiger Länge und einen Schlüssel und einen Intialisierungsvektor. Der String wird mit einem symmetrischen Verschlüsselungsverfahren verschlüsselt und als String von beliebiger Länge zurückgegeben.

Geeignete Verfahren wären beispielsweise AES[11] oder Salsa20[12].

3.7.10 Entschlüsselung

string `decrypt(`*string* `cipher,` *key* `key,` *key* `vector)` – Erwartet einen String von beliebiger Länge und zwei Schlüssel. Der String wird mit einem symmetrischen Verschlüsselungsverfahren entschlüsselt und als String von beliebiger Länge zurückgegeben.

Geeignete Verfahren wären beispielsweise AES[13] oder Salsa20[14].

4 REDS Single-User Protokoll

Der folgende Abschnitt beschreibt schematisch die Kommunikation zwischen Leaf, Node und Pod. Dabei wird eine dreispaltige Notation verwendet, welche sequentiell von oben nach unten alle Aktionen zeigt, die auf den jeweiligen Komponenten ausgeführt werden. Die Möglichkeit, dass einzelne Aktionen gegebenenfalls parallel ausgeführt werden können, wird nicht berücksichtigt.

Leaf	**Node**	**Pod**
APP → a		
a →	→ a	
	b = a * 5	
	a, b →	→ a, b
		c = a * b * 7
		c → DB(a)
c, b ←	← c, b ←	← c, b
d = c * b		

Ein Datentransfer zwischen den Komponenten wird durch einen Pfeil → oder ← in Verbindung mit einem Spaltenwechsel dargestellt. Die übertragenen Daten werden vor bzw. hinter den Pfeil geschrieben.

Die Pfeilnotation wird außerdem verwendet, wenn Variablenwerte aus dem Speicher gelesen oder in den Speicher geschrieben werden. Eine Variable in Klammern hinter dem Speichernamen benennt eine Referenzvariable über die der Wert adressiert wird. In den folgenden Schemata werden mehrere benannte Speicher verwendet:

- **APP:** Die Anwendung in die das Node-Client-Modul integriert ist.
- **MEM:** Der temporäre Speicher auf dem Leaf, Node oder Pod.
- **DB:** Die permanente Datenbank auf dem Node oder Pod.
- **KEYS:** Der Keyring im temporären Speicher des Leaf und verschlüsselt in der Datenbank des Node.
- **CONF:** Die Konfigurationsdaten von Node und Pod.

4.1 Authentifizierung

Die Authentifizierung des Leaf am Node oder des Node am Pod erfolgt durch die Signierung mit einem Authentifizierungsschlüssel, welcher bei der Registrierung ausgetauscht wird und danach beiden Seiten bekannt ist. Authentifizierungsschlüssel werden auf dem Node und Pod permanent gespeichert und auf dem Leaf aus Nutzernamen und Passwort generiert.

4.1.1 Registrierung eines Nutzers

Der erste Schritt bei der Anmeldung eines neuen Nutzers ist die Registrierung am Node. Dazu wird zunächst aus dem Nutzernamen *name* und dem dazugehörigen Passwort *pw* das Nutzerpseudonym *alias* generiert. Der Node erzeugt durch die Funktion *id()* eine nodeweit eindeutige die Account-ID *aid* und generiert den Authentifizierungsschlüssel *auth*. Diese werden zusammen mit dem Accountsalt *asalt* in der Datenbank gespeichert.

Nach der Registrierung sind dem Leaf *aid*, *auth* und das Accountsecret *asec* bekannt. Der Authentifizierungsschlüssel wird im Keyring gespeichert, um diesen bei zukünftigen Anmeldungen (→ Kapitel 4.2.1) verfügbar zu haben.

Leaf	**Node**
APP → name, pw	
alias = shash(name, pw) asalt = key() asec = shash(sconc(name, pw), asalt)	
$auth_L$ = private() $auth_L'$ = public($auth_L$)	
alias, asalt, $auth_L'$ →	→ alias, asalt, $auth_L'$
	aid = id() $auth_N$ = private() $auth_N'$ = public($auth_N$) auth = shared($auth_N$, $auth_L'$) aid, alias, asalt, auth → DB(aid)
aid, $auth_N'$ ←	← aid, $auth_N'$
auth = shared($auth_L$, $auth_N'$) auth → KEYS aid, asec → MEM	

Ein mögliches Angriffsszenario wäre eine Man-in-the-Middle Attacke, bei dem der Angreifer sich zwischen Leaf und Node schaltet. Alternativ könnte sich der Angreifer auch selbst die Aufgaben des Node übernehmen. Beide Szenarien wären jedoch unproblematisch, der Node im Allgemeinen sowieso nicht als vertrauenswürdige Instanz eingestuft wird.

Die Authentifizierung ist vor allem aus Nodeperspektive relevant, da so eine Zugriffskontrolle ausgeübt werden kann, was verschiedene Monetarisierungsmodelle ermöglicht (→ Kapitel 3.5.1).

4.1.2 Registrierung eines Pods

Nachdem der Leaf dem Node eine bisher unbekannte Pod-URL *purl* mitgeteilt hat, kontaktiert dieser den Pod, um einen Authentifizierungsschlüssel *auth* auszutauschen. Dabei wird dem Pod auch die weltweit eindeutige Node-ID *nid* genannt und dem Pod auf dem Node eine eindeutige Pod-ID *pid* zugeteilt. Nach

der Registrierung werden *auth* sowie *pid* bzw. *nid* in den Datenbanken von Node und Pod gespeichert. Außerdem wird *pid* an den Leaf gesendet.

Wie genau die weltweite Eindeutigkeit der Node-ID sichergestellt wird, ist der konkreten Implementation überlassen, so könnte zum Beispiel die URL des Nodes auch als Node-ID verwendet werden. Die Node-ID definiert auf dem Pod einen unabhängigen Namespace in dem alle Domains des Node gespeichert werden. Auf diese Weise können auf einem Pod Daten von mehreren Nodes gespeichert werden.

Leaf	**Node**	**Pod**
APP → purl		
purl →	→ purl	
	CONF → nid	
	pid = id();	
	$auth_N$ = private()	
	$auth_N'$ = public($auth_N$)	
	nid, pid, $auth_N'$ →	→ nid, pid, $auth_N'$
		$auth_P$ = private()
		$auth_P'$ = public($auth_P$)
		auth = shared($auth_P$, $auth_N'$)
		nid, pid, auth → DB(nid)
	$auth_P'$ ←	← $auth_P'$
	auth = shared($auth_N$, $auth_P'$)	
	pid, auth, purl → DB(pid)	
pid ←	← pid	

Wie bei der Registrierung eines Nutzers ist auch hier eine Man-in-the-Middle Attacke, bei der sich der Angreifer zwischen Node und Pod schaltet oder selbst vorgibt, der Pod zu sein, denkbar. Diese Szenarien sind unproblematisch, da für den Node die korrekte Identität des Pods irrelevant ist und diese bei Zugriffen durch den Leaf entweder über des Pod-Password (→ Kapitel 4.3.1) oder über die auf dem Pod vorhandenen Schlüssel (→ z.B. Kapitel 4.4.1) verifiziert wird.

4.1.3 Signierung

Einige Anfragen vom Leaf an den Node und alle Anfragen vom Node an den Pod werden mit Hilfe des Authentifizierungsschlüssels signiert. Sobald der Empfänger die Signaturen erfolgreich verglichen hat, gilt die Anfrage als authentifiziert. Sofern eine Anfrage signiert wurde, muss auch die entsprechende Antwort mit dem gleichen Authentifizierungsschlüssel signiert werden.

Welche Anfragen vom Leaf an den Node signiert werden müssen, wird durch das Authentifizierungsraster des Nodes definiert (→ Kapitel 3.5.1). Die Bandbreite reicht hier von der ausschließlichen Authentifizierung bei der Accounterstellung bis hin zur Authentifizierung jeder Anfrage.

Bei einer Anfrage werden nicht nur die Daten $data_A$, sondern auch der Befehl mit den dazugehörigen Parametern[a] *cmd*, die zugehörige Account- beziehungsweise Node-ID *id* und der aktuellen Timestamp $time_A$ signiert. Bei einer Antwort werden die Daten $data_B$ zusammen mit der Timestamp $time_B$ signiert.

Leaf *oder* Node	**Node *oder* Pod**
MEM *oder* DB(pid) → id, auth	
$time_A$ = time()	
req = sconc(id, cmd, $data_A$, $time_A$)	
sig_A = hmac(req, auth)	
id, cmd, $data_A$, $time_A$, sig_A →	→ id, cmd, $data_A$, $time_A$, sig_A
	DB(id) → auth
	req' = sconc(cmd, $data_A$, $time_A$)
	sig_A' = hmac(req', auth)
	sig_A' == sig_A ?
	$data_A$ → *Ausführung von cmd* → $data_B$
	$time_B$ = time()
	res = sconc($data_B$, $time_B$)
	sig_B = hmac(res, auth)
$data_B$, $time_B$, sig ←	← $data_B$, $time_B$, sig_B
res' = sconc(data', time')	
sig_B' = hmac(res', auth)	
sig_B' == sig_B ?	

Um Replay-Attacken zu vermeiden, sollte die Timestamp auf ihre Aktualität geprüft werden. Idealerweise ist die soeben empfangene Timestamp immer jünger als die zuletzt gesendete Timestamp sein. Eine derart strikte Kontrolle kann jedoch problematisch sein, wenn nicht davon ausgegangen werden kann, dass die Uhren der Kommunikationspartner synchron laufen.

4.1.4 Kurzschreibweise

Im Folgenden werden die Authentifizierungsschritte zur besseren Lesbarkeit meist vollständig weggelassen. Im Allgemeinen gilt, dass eine Authentifizierung bei der Kommunikation zwischen Leaf und Node nur notwendig ist, wenn eine Zugriffskontrolle durch den Node erfolgen soll. Die Kommunikation zwischen Node und Pod muss, bis auf die Registrierung eines Pods, immer authentifiziert sein.

Sollte die Authentifizierung besonders relevant sein, so wird diese durch folgende Kurzschreibweise gekennzeichnet. Dabei werden die signierten Daten in eckige Klammern gefasst, wobei der verwendete Schlüssel am Anfang der Klammerung steht:

a Ein Befehl mit Parametern wäre zum Beispiel „DELETE /address/5“, um die Entity mit der ID 5 vom Typ „address“ zu löschen.

Leaf	Pod
[pkey: data] →	→ [pkey: data]
[pkey: data] ←	← [pkey: data]

4.2 Keyring

Ein Keyring enthält neben dem Authorisierungsschlüssel des jeweiligen Nutzers in der Regel außerdem noch die jeweiligen Domaintickets hat (→ Kapitel Error: Reference source not found und Error: Reference source not found). Der Keyring wird für den Zeitraum der Anmeldung im temporären Speicher des Leaf gehalten und außerdem verschlüsselt in der Datenbank des Node gespeichert. Die Verschlüsselung erfolgt mit dem Accountsecret des jeweiligen Nutzers, so dass nur dieser Zugriff auf den Keyring hat.

Der Keyring wird auf dem Node als monolithischer Datensatz gespeichert, in dem alle Keyringdaten zusammengefasst sind. So ist sichergestellt, dass der Node keine Aussagen über den Inhalt des Keyrings machen kann. Im Umkehrschluss bedeutet dies jedoch auch, dass der Leaf keine einzelnen Keyringdaten gezielt anfragen kann, sondern immer den gesamten Keyring anfragen oder aktualisieren muss.

4.2.1 Anmeldung eines Nutzers

Die Anmeldung erfolgt mit dem Nutzernamen *name* und dem dazugehörigen Passwort *pw* aus denen das Nutzerpseudonym *alias* generiert wird. Der Leaf fordert darauf die öffentlichen Accountdaten für *alias*, das heißt die Account-ID *aid*, das Accountsalt *asalt* und die verschlüsselten Keyringdaten *keys'* mit dem dazugehörigen Vector *vec*, beim Node an.

Nach dem Empfang dieser Daten berechnet der Leaf aus *name*, *pw* und *asalt* das Accountsecret *asec* mit dem die Keyringdaten entschlüsselt werden können. Die entschlüsselten Daten *keys* werden als neuer Keyring verwendet und *asec* sowie *aid* für den Zeitraum der Anmeldung im temporären Speicher des Leaf gespeichert.

Leaf	Node
APP → name, pw	
alias = shash(name, pw)	
alias →	→ alias
	DB(alias) → aid, asalt, keys', vec
aid, asalt, auth', keys', vec ←	← aid, asalt, auth', keys', vec
asec = shash(sconc(name, pw), asalt)	
keys = decrypt(asec, vec, keys')	
keys → KEYS	
aid, asec → MEM	

4.2.2 Aktualisieren des Keyrings

Der Leaf generiert zunächst einen neuen Vektor *vec* und nutzt diesen, um die gesammelten Keyringdaten *key* zu verschlüsseln. Danach werden die verschlüsselten Keyringdaten *key'* sowie *vec* an den Node übermittelt und dort gespeichert. Abschließend sendet der Node eine Erfolgsmeldung an den Leaf.

Leaf	Node	Pod
MEM → aid, asec KEYS → keys vec = time() keys' = encrypt(auth, vec, keys) keys', vec [aid] → *ok* ←	 → [aid] keys', vec keys', vec → DB(aid) ← *ok*	

4.3 Domains

Sobald der Nutzer einen Account erstellt hat, kann er Domains anlegen, um darin Daten in Form von Entities zu speichern. Da die Zugriffsrechte innerhalb einer Domain über sogenannte Tickets definiert werden, benötigt jeder Nutzer einer Domain ein eigenes Ticket.

Mit dem Ticketschlüssel werden Zugriffe auf Domaindaten signiert und verschlüsselt, daher muss bereits bei der Erstellung einer Domain ein entsprechendes Ticket für den Domainbesitzer angelegt werden. Über die Erstellung weiterer Tickets kann mehreren Nutzern Zugriff auf eine Domain gewährt werden (→ Kapitel 5).

4.3.1 Erstellung einer Domain

Die Erstellung einer Domain erfolgt genaugenommen in zwei Schritten. Zunächst wird die neuen Domain auf dem Pod erstellt und eine Einladung zwischen Ersteller und Pod ausgetauscht. Diese Einladung wird dann vom Domainersteller in einem zweiten Schritt dazu verwendet, um das erste Ticket für die neue Domain anzulegen.

Dem Ersteller müssen die URL *purl* unter der Pod zu erreichen ist und das Pod-Passwort *ppw* bekannt sein. Der Leaf generiert zunächst den Einladungsschlüsselteil $ikey_L$ und sendet diesen an den Node, welcher mittels *id()* eine nodeweit eindeutige Domain-ID *did* erzeugt. Die zur Domain gehörende Pod-ID *pid* wird zusammen mit *did* gespeichert und *did* sowie $ikey_L'$ an den Pod weitergeleitet.

Auf dem Pod wird zunächst gegebenenfalls pkey auf ppw und psal berechnet, wobei die Berechnung von pkey auch bereits vor dem Empfang (z.B. einmalig beim Starten des Pods) geschehen kann. Dann werden der zweite Einladungsschlüsselteil $ikey_P$ und die Einladungsschlüsselvorstufe *ikey°* generiert, worauf der finale Einladungsschlüssel *ikey* aus *ikey°* und *pkey* berechnet werden kann. Nun werden die Ticketflags *tflags* (→ Kapitel 4.3.2) für das Besitzerticket erstellt und die Einladung sowie die Domain in der Datenbank des Pods gespeichert. Abschließend werden die Einladungsdaten *iid*, $ikey_L$ und $ikey_P$ sowie *psalt* an den Node gesendet. Dort wird die Einladung registriert und die Antwort an den Leaf weitergeleitet.

Auf dem Leaf wird zunächst *pkey* aus *ppw* und *psalt* berechnet. Anschließend kann die Signatur der Antwort überprüft und *ikey* generiert werden. Abschließend werden *iid* und *ikey* für die nun folgende Erstellung des Besitzertickets im Keyring gespeichert.

Leaf	Node	Pod
APP → purl, ppw		

iid = key() $ikey_L$ = private() $ikey_L'$ = public($dkey_L$)		
purl, iid, $ikey_L'$ →	→ purl, iid, $ikey_L'$ DB(purl) → pid did = id() iid, did → DB(iid) did, pid → DB(did)	
	did, iid, $ikey_L'$ →	→ did, iid, $ikey_L'$ CONF → psalt, ppw pkey = shash(ppw, psalt) $ikey_P$ = private() $ikey_P'$ = public($dkey_P$) ikey° = shared($dkey_P$, $dkey_L'$) ikey = hmac(ikey°, pkey) tflags = *owner* did → DB(did) iid, ikey, did, tflags → DB(iid)
[pkey: iid, $ikey_L'$, $ikey_P'$, psalt] ← pkey = shash(ppw, psalt) ikey° = shared($ikey_L$, $ikey_P'$) ikey = hmac(ikey°, pkey) iid, ikey → KEYS(iid) iid → APP	← [pkey: iid, $ikey_L'$, $ikey_P'$, psalt] ←	← [pkey: iid, $ikey_L'$, $ikey_P'$, psalt]

Da nur ein Kommunikationspartner mit Kenntnis von *ppw* und *psalt* den korrekten Podschlüssel *pkey* generieren kann, kann *pkey* für die Authentifizierung der Kommunikation verwendet werden. Notwendige Vorraussetzung ist allerdings, dass die Generierung von *pkey* mittels *shash()* so aufwendig ist, dass Brute-Force-Angriffe de facto unmöglich sind.

Der Leaf kann nach der Generierung von *pkey* die Signatur der Antwort überprüfen und so sicherstellen, dass die Antwort tatsächlich von dem gewünschten Pod kommt und dass der Pod den korrekten Einladungsschlüsselteil des Leafs empfangen hat. Umgekehrt ist durch die Verknüpfung von *ikey°* mit *pkey* sichergestellt, dass nur ein Nutzer mit Kenntnis von *pkey* einen korrekten Einladungsschlüssel generieren kann. Somit ist ein Man-in-the-Middle Angreifer nicht in der Lage, sich zwischen die Kommunikation von Leaf und Pod zu schalten.

Allerdings kann der Pod erst nach dem erfolgreichen Erstellen eines Besitzertickets davon ausgehen, dass die Domainerstellung nicht durch einen Angreifer korrumpiert wurde. Daher sollten Pod und Node die Domain wieder entfernen, falls innerhalb eines angemessenen Zeitraumes kein Besitzerticket erstellt wurde.

4.3.2 Erstellen eines Tickets

Nach dem Empfang der Einladungsdaten kann der Empfänger ein Ticket erstellen, um Zugriff auf die Domain zu erlangen. Zunächst wird dafür auf dem Leaf ein Ticketschlüsselteil $tkey_L$ generiert. Dieser wird signiert mit dem Einladungsschlüssel *ikey* an den Pod übertragen.

Nach der erfolgreichen Überprüfung der Signatur erzeugt der Pod die Ticket-ID *tid* und den zweiten Ticketschlüsselteil $tkey_P$ und kann so den Ticketschlüssel *tkey* berechnen. Die Domain-ID *did* und die Zugriffsrechte *tflags* werden zusammen mit *tid* und *tkey* in der Datenbank des Pods gespeichert.

Außerdem werden *tid*, $tkey_P'$, *tflags* und *did* an den Leaf gesendet. Dieser kann nun ebenfalls *tkey* berechnen und *tid*, *tkey*, *did* und eventuell *data* im Keyring speichern.

Leaf	**Node**	**Pod**
KEYS(iid) → iid, ikey		
$tkey_L$= key()		
$tkey_L'$ = public($tkey_L$)		
[ikey: iid, $tkey_L'$] →	→ [ikey: iid, $tkey_L'$] →	→ [ikey: iid, $tkey_L'$]
		DB(iid) → ikey, tflags, did
		tid = id()
		$tkey_P$ = private()
		$tkey_P'$ = public($tkey_P$)
		tkey = shared($tkey_P$, $tkey_L'$)
		tid, tkey, tflags, did → DB(tid)
[ikey: tid, $tkey_P'$, tflags, did] ←	← [ikey: tid, $tkey_P'$, tflags, did] ←	← [ikey: tid, $tkey_P'$, tflags, did]
tkey = shared($tkey_L$, $tkey_P'$)		
tid, tkey, tflags, did → KEYS(tid)		

Durch die Signierung der Anfrage mit geheimen Einladungsschlüssel *ikey* kann der Pod sicherstellen, dass der Leaf eine gültige Einladung besitzt. Umgekehrt kann der Leaf feststellen, dass die empfangene Antwort tatsächlich von dem Pod stammt, auf dem die Einladung abgelegt wurde.

Der Einladungsschlüssel wird hier bewusst nur für die Signierung und nicht für die Verschlüsselung des Schlüsselaustausches verwendet. Würde andernfalls der Ticketschlüssel auf dem Leaf (oder Pod) generiert und verschlüsselt zum Pod (oder Leaf) übertragen werden, wäre es einem Angreifer möglich an den Ticketschlüssel zu gelangen, falls der Einladungsschlüssel nachträglich publik werden sollte.

4.4 Streams

Domaindaten werden zwischen Leaf und Pod **verschlüsselt übertragen**, aber auf dem Pod **unverschlüsselt gespeichert**. Zugriff ist daher sowohl durch den Leaf als auch durch den Pod möglich. Ein typisches Beispiel sind zusätzliche Informationen über die Mitglieder einer Domain. Insbesondere werden aber auch Entities (→ Kapitel Error: Reference source not found) wie Domaindaten ausgetauscht.

Der Austausch von Domaindaten zwischen Leaf und einem Pod erfolgt über sogenannte Streams. Dies sind temporare Datentunnel, die mit Hilfe eines Tickets zwischen einem Leaf und einer Domain auf einem Pod aufgebaut werden können. Ein Stream wird durch den Leaf geöffnet und wird sowohl von Leaf als auch Pod automatisch geschlossen, wenn der Stream für eine vereinbarte Zeit lang nicht genutzt wurde. Für den Zeitraum seiner Existenz hat jeder Stream eine auf Leaf und Pod eindeutige Stream-ID.

Die Verschlüsselung erfolgt symmetrisch mit einem temporaren Schlüssel, der zerstört wird, sobald der Stream geschlossen wird. So können Daten, die über einen Stream gesendet wurden, später nicht mehr entschlüsselt werden. Auf diese Weise wird die Perfect-Forward-Secrecy der übertragenen Domaindaten sichergestellt.

Es wird immer ein Schlüsselteil durch den Leaf und der andere Teil durch den Pod generiert und die öffentlichen Schlüsselteile ausgetauscht. Um Man-in-the-Middle Angriffe zu vermeiden, wird zwischen Leaf und Pod mit dem Ticketschlüssel signiert. So können dem Stream auch die Zugriffsrechte des jeweiligen Tickets zugeordnet werden, so dass der Ticketschlüssel nach dem Öffnen des Streams nicht mehr benötigt wird.

4.4.1 Öffnen eines Streams

TODO: Beschreibung

Leaf	**Node**	**Pod**
APP → did		
$skey_L$ = private() $skey_L'$ = public($skey_L$)		
[tkey: tid, did, $skey_L'$] →	→ [tkey: tid, did, $skey_L'$] →	→ [tkey: tid, did, $skey_L'$]
		DB(tid) → tkey, tflags, did *verify tkey signature; compare did* $skey_P$ = private() $skey_P'$ = public($skey_P$) skey = shared($skey_P$, $skey_L'$) ssalt = key() sid = hmac(ssalt, skey) sid, skey, did, tflags → MEM(sid) *repeat until sid id unique*
[tkey: tid, tflags, did, $skey_P'$, ssalt] ←	← [tkey: tid, tflags did, $skey_P'$, ssalt] ←	← [tkey: tid, tflags, did, $skey_P'$, ssalt]
skey = shared($skey_L$, $skey_P'$) sid = hmac(ssalt, skey) sid, ssalt, tflags → MEM(did) *ok* → APP		

4.4.2 Datenaustausch über einen Stream

TODO: Beschreibung

Leaf	**Node**	**Pod**
APP → did, req		
MEM(did) → sid, skey		
$svec_L$ = timestamp()		
req' = encrypt(skey, $svec_L$, req)		
[skey: sid, did, $svec_L$, req'] →	→ [skey: sid, did, $svec_L$, req']	
	DB(did) → purl	
	[skey: sid, did, $svec_L$, req'] →	→ [skey: sid, did, $svec_L$, req']
		MEM(sid) → skey, did, tflags
		check skey signature, did & tflags
		req = decrypt(skey, $svec_L$, req')
		req → *process request* → res
		$svec_P$ = timestamp()
		res' = encrypt(skey, $svec_P$, res)
[skey: sid, did, $svec_P$, res'] ←	← [skey: sid, did, $svec_P$, res'] ←	← [skey: sid, did, $svec_P$, res']
check skey signature & did		
res = decrypt(skey, $svec_P$, res')		
res → APP		

Kurzschreibweise

Die soeben beschriebenen Verschlüsselungsschritte werden zur besseren Lesbarkeit in allen weiteren Diagrammen weitestgehend weggelassen. Stattdessen wird folgende Kurzschreibweise verwendet, wenn Daten verschlüsselt übertragen werden.

Das Beispiel zeigt, wie eine Anfrage *req* und die entsprechende Antwort *res* mit einem Streamschlüssel *skey* für die Domain mit der ID *did* verschlüsselt und signiert werden. *sid* und *did* werden unverschlüsselt übertragen, sind jedoch Teil der signierten Nachricht. Es wird vorausgesetzt, dass die entsprechenden Schlüssel und Vektoren auf dem Leaf und auf dem Pod existieren.

Leaf	**Node**	**Pod**
APP → did, req		
[skey: sid, did, {skey: req}] →	→ [skey: sid, did, {skey: req}] →	→ [skey: sid, did, {skey: req}]
		req → *process request* → res
[skey: sid, did, {skey: res}] ←	← [skey: sid, did, {skey: res}] ←	← [skey: sid, did {skey: res}]
res → APP		

4.4.3 Aktualisieren des Streamschlüssels

TODO: Beschreibung

Leaf	**Node**	**Pod**
APP → did MEM(did) → sid, skey $skey_L$ = private() $skey_L'$ = public($skey_L$) [skey: sid, $skey_L'$] →	→ [skey: sid, $skey_L'$] DB(did) → purl [skey: sid, $skey_L'$] →	→ [skey: sid, $skey_L'$] MEM(sid) → skey *check skey signature* $skey_P$ = private() $skey_P'$ = public($skey_P$) skey° = skey skey = shared($skey_P$, $skey_L'$) skey → MEM(sid)
[skey°: sid, $skey_P'$] ← *check skey° signature* skey = shared($skey_L$, $skey_P'$) skey → MEM(sid) *ok* → APP	← [skey°: sid, $skey_P'$] ←	← [skey°: sid, $skey_P'$]

4.5 Entities

Entities gehören immer einer bestimmten Domain und einem bestimmten Typ an. Die Entity-ID wird beim Erstellen durch den Node vergeben. Da durch die Kombination von Typ und ID jede Entity nodeweit eindeutig identifizierbar sein muss, wird die Node-ID durch den Node vergeben. Außerdem werden die Verknüpfungen zwischen Entities, sowie die Domainzugehörigkeiten auf dem Node verwaltet. Die eigentlichen Entitydaten werden auf dem Pod als Domaindaten gespeichert und beim Transport entsprechend signiert und verschlüsselt.

4.5.1 Erstellen einer Entity

Das Erstellen von Entities in der Domain mit der ID *did* erfolgt ähnlich wie das Erstellen von anderen Domaindaten (→ Kapitel 4.4). Allerdings wird die Entity-ID *eid* auf dem Node erzeugt und nach dem Erstellen der Entity auf dem Pod zusammen mit dem Entity-Typ *etype* und der Domain-ID *did* in der Datenbank des Node gespeichert.

Leaf	**Node**	**Pod**

APP → did, etype, data [skey: {skey: data}, cmd, did, etype] →	→ [skey: {skey: data}, cmd, did, etype] eid = id() [skey: {skey: data}, cmd, did, etype], eid →	→ [skey: {skey: data}, cmd, did, etype], eid did, eid, etype, data → DB(etype, eid) DB(etype, eid) → data'
	[skey: {skey: data'}, did, etype, eid] ← did, etype → DB(eid)	← [skey: {skey: data'}, did, etype, eid]
[skey: {skey: data'}, did, etype, eid] ← data' → APP	← [skey: {skey: data'}, did, etype, eid]	

Durch senden einer bereits existierenden *eid*, könnte ein Angreifer mit Kontrolle über den Node Daten auf dem Pod zerstören. Daher muss der Pod die Verarbeitung abbrechen, wenn die Kombination *etype+eid* bereits existiert.

4.5.2 Zugriff auf Entitydaten

Der Zugriff auf Daten in bereits bestehenden Entities erfolgt wie bei normalen Domaindaten (→ Kapitel 4.4).

Leaf	**Node**	**Pod**
APP → cmd, etype, eid, data [tid: {tid: data}, cmd, etype, eid] →	→ [tid: {tid: data}, cmd, etype, eid] →	→ [tid: {tid: data}, cmd, etype, eid] data → *Ausführen von cmd* → data'
	[tid: {tid: data'}] ← *Ausführen von cmd*	← [tid: {tid: data'}]
[tid: {tid: data'}] ← data' → APP	← [tid: {tid: data'}]	

4.5.3 Zugriff auf Entityverknüpfungen

Verknüpfungen werden immer zwischen einer Elternentity *ptype*, *pid* und einer Kindentity *ctype*, *cid* definiert und auf dem Node gespeichert. Entscheidend für die Zugriffskontrolle sind die Rechte des Tickets für die Domain der Elternentity. Daher wird die Anfrage auf dem Leaf mit dem Ticket für die Elterndomain signiert. Der Node leitet die Anfrage zunächst an den Pod weiter, um sich die Zugriffsrechte bestätigen zu lassen. Erst danach wird das Kommando *cmd* (C = create, D = delete) auf dem Node ausgeführt.

Leaf	**Node**	**Pod**
APP → cmd, ptype, pid, ctype, cid [tid: cmd, ptype, pid, ctype, cid] →	→ [tid: cmd, ptype, pid, ctype, cid] → *ok* ←	→ [tid: cmd, ptype, pid, ctype, cid] ← *ok*

ok ←	*Ausführen von cmd* *← ok*	

5 REDS Multi-User Protokoll

Durch den Versand von Tickets ist es möglich, anderen Personen Zugriff auf Domaindaten zu gewähren. Tickets werden von einem Nutzer, der bereits Zugriff auf die Domain hat, erstellt und an den Empfänger des Tickets gesendet.

Schlüssel und ID des neuen Tickets werden dabei jedoch nicht direkt vom Ticket-Ersteller an den Ticket-Empfänger übergeben. Vielmehr platziert der Ersteller zunächst eine Einladung auf dem Pod und sendet dann die Einladungsdaten an den Empfänger. Dieser kann dann die Einladung Nutzen, um ein Ticket für die entsprechende Domain auf dem Pod zu erstellen (→ Kapitel 4.3.2).

Nur Domainadministratoren, also Mitglieder bei deren Tickets die Administration-Flag gesetzt ist, können Tickets vergeben und die Zugriffsrechte innerhalb der Domain setzen. Ein Domainadministrator kann immer nur die Rechte gewähren oder verweigern, die er selbst besitzt. Ein Domainbesitzer hat immer alle Zugriffsrechte auf die Domaindaten und kann als einziger die Domain als Ganzes löschen. Der Besitz einer Domain kann übertragen werden, es kann jedoch immer nur genau einen Domainbesitzer geben.

5.1 Einladung über einen sicheren Kanal

Sofern die Einladungsdaten über einen sicheren Kanal an den Empfänger gesendet werden können, folgt die Erstellung einem einfachen Schema. Voraussetzungen an den sicheren Kanal sind, dass die Daten sowohl verschlüsselt übertragen werden als auch zumindest der Empfänger authentifiziert werden kann. Ein typisches Beispiel für einen sicheren Kanal, wäre der Versand einer mit dem Public-Key des Empfängers verschlüsselten Email nach dem OpenPGP Standard[a].

5.1.1 Platzieren einer Einladung

Nachdem die Empfängeradresse *rcvr* sowie die Domain-ID *did und* die Zugriffsrechte *tflags* von der Anwendung übergeben wurden, werden durch den Leaf die Einladings-ID *iid* und der Einladungsschlüssel *ikey* generiert. Diese werden zusammen mit *did* und *tflags* an den Node gesendet, wobei *iid* und *did* sowohl verschlüsselt als auch unverschlüsselt gesendet werden.

Der Node speichert *did* zusammen mit *iid*, um bei späteren Anfragen *iid* an den richtigen Pod weiterleiten zu können. Sollte *iid* bereits vergeben sein, wird die Anfrage abgebrochen und ein Fehler zurück an den Leaf gesendet. Andernfalls werden die verschlüsselten Daten an den Pod weitergeleitet. Dort wird die Einladungs-ID *iid* zusammen *ikey*, *did* und *tflags* gespeichert. Die Antwort des Pods enthält *iid* in verschlüsselter Form.

Nachdem der Leaf die Antwort empfangen und überprüft hat, werden abschließend *iid* und *ikey* über den sicheren Kanal an die Empfängeradresse *rcvr* gesendet.

Leaf	**Node**	**Pod**
APP → rcvr, did, tflags iid = key() ikey = key() [tid: iid,did, {tid: iid,ikey,did,tflags}] →	→[tid: iid,did, {tid: iid,ikey,did,tflags}]	

a RFC 4880: https://tools.ietf.org/html/rfc4880

	iid, did → DB(iid) [tid: iid,did, {tid: iid,ikey,did,tflags}] →	→ [tid: iid,did, {tid: iid,ikey,did,tflags}] iid, ikey, did, tflags → DB(iid)
[{tid: iid}] ← iid, ikey → MSG(rcvr) APP ← iid	← [{tid: iid}] ←	← [{tid: iid}]

Die Generierung von *iid* auf dem Leaf ist eine Angleichung in Hinblick auf den Versand über einen unsicheren Kanal (→ Kapitel 5.2). Dieses Vorgehen kann zwar zu Kollisionen führen, diese sind aber bei Schlüssellängen größer 128bit hinreichend unwahrscheinlich, so dass Verfahren hier akzeptabel ist.

5.2 Einladung über einen unsicheren Kanal

Der Versand einer Einladung über einen unsicheren Kanal muss in drei Schritten erfolgen, da zunächst die Einladungs-ID und der Einladungschlüssel ausgetauscht und die Identität des Empfängers bestätigt werden müssen.

Nachrichten, die über den unsicheren Kanal versendet werden, können abgehört werden und sind darüber hinaus auch noch mit der Sender- und Empfängeradresse verknüpft. Daher sind die folgenden Protokolle so konzipiert, dass keine Informationen, welche Rückschlüsse auf involvierte Accounts oder Domains zulassen, zusammen mit den Einladungsdaten über den unsicheren Kanal versendet werden. Insbesondere gehören dazu Account-, Domain-, Ticket- und Einladungs-ID sowie der Einladungsschlüssel.

Grundsätzlich ist es Dritten (insbesondere den Nodebetreiber) durch das Abfangen von Nachrichten möglich, Verknüpfungen zwischen Sender und Empfänger herzustellen. Diese sind allerdings nicht verlässlich, da derartige Nachrichten problemlos von Dritten, denen die Sender- und Empfängeradresse bekannt sind, gefälscht werden können. Demnach kann der tatsächliche Austausch von Einladungsdaten sowohl vom Sender als auch vom Empfänger plausibel verleugnet werden.

5.2.1 Senden einer Einladung

Bei dem Versand über einen unsicheren Kanal, werden auf dem Leaf des Einladenden zunächst die Austausch-ID *xid* sowie der senderseitige Teil des Austauschschlüssels $xkey_S$ und des Austauschsalts $xsalt_S$ generiert. Alle Werte werden zum einen für die spätere Verwendung im Keyring gespeichert und zum anderen mit der Absenderadresse *sndr* versehen und an den Empfänger der Einladung *rcvr* gesendet.

Leaf

APP → did, tflags, sndr, rcvr,

xid =id()

$xkey_S$ = private()

$xkey_S'$ = public($xkey_S$)

$xsalt_S$ = key()

did, tflags, $xkey_S$, $xsalt_S$ → KEYS(xid)

xid, $xkey_S'$, $xsalt_S$, sndr → MSG(rcvr)

5.2.2 Annehmen einer Einladung

Sobald die Einladungsdaten den Empfänger erreicht haben, kann dieser die empfängerseitigen Teile des Austauschschlüssels $xkey_R$ und des Austauschsalts $xsalt_R$ generieren. Danach ist es dem Leaf möglich den finalen Austauschschlüssel *xkey* berechnen. Dann können die die Einladungs-ID *iid* und der Einladungsschlüssel *ikey* sowie die Einladungsprüfsumme *isig* berechnet werden. Abschließend werden *iid*, *ikey* und *isig* für die spätere Verwendung im Keyring gespeichert und *xid*, $xkey_R'$ und $xsalt_R$ an die Senderadresse *sndr* gesendet.

Leaf
MSG(rcvr) → xid, $xkey_S'$, $xsalt_S$, sndr
$xkey_R$ = private()
$xkey_R'$ = public($ikey_R$)
$xsalt_R$ = key()
xkey = shared($ikey_R$, $ikey_S'$)
xstr = sconc(xid, $xsalt_S$, $xsalt_R$)
isig = hmac(xstr, xkey)
iid = hmac($xsalt_S$, xkey)
ikey = hmac($xsalt_R$, xkey)
iid, ikey, isig → KEYS(iid)
xid, $xkey_R'$, $xsalt_R$ → MSG(sndr)

5.2.3 Bestätigen einer Einladung

Nach dem Empfang der Antwortnachricht ist es auch dem Absender der Einladung möglich, den Austauschschlüssel *xkey* und die Einladungsprüfsumme *isig'* zu berechnen. Nun kann dieser durch den Vergleich der Prüfsummen sowohl die Identität des Empfängers als auch die Korrektheit der ausgetauschten Schlüssel überprüfen. Für die Überprüfiug der Identität ist es essentiell, dass *isig* über einen Kanal empfangen wird, über den der Absender der Einladung die Identität des Empfängers sicher feststellen kann. Ob dieser Kanal privat oder öffentlich ist, spielt keine Rolle.

Danach wird auf auch dem Leaf des Absenders die Einladungs-ID *iid* und der Einladungsschlüssel *ikey* berechnet. Nun kann die eigentliche Einladung auf dem Pod platziert werden, wobei auf die beim Versand gespeicherte Domain-ID *did* und Ticket-Flags *tflags* (→ Kapitel 5.2.1) zurückgegriffen wird. Dies geschieht analog zum Platzieren einer Einladung über einen sicheren Kanal (→ Kapitel 5.1.1).

Leaf	**Node**	**Pod**
MSG(sndr) → xid, $xkey_R'$, $xsalt_R$		
KEYS(xid) → did, tflags, $xkey_S$, $xsalt_S$		
RCVR → isig		
xkey = shared($xkey_S$, $xkey_R'$)		
xstr = sconc(xid, $xsalt_S$, $xsalt_R$)		

isig' = hmac(xstr, xkey) isig' == isig ? iid = hmac($xsalt_S$, xkey) ikey = hmac($xsalt_R$, xkey)		
[tid: iid,did, {tid: iid,ikey,did,tflags}] →	→[tid: iid,did, {tid: iid,ikey,did,tflags}] iid, did → DB(iid)	
	[tid: iid,did, {tid: iid,ikey,did,tflags}] →	→ [tid: iid,did, {tid: iid,ikey,did,tflags}] iid, ikey, did, tflags → DB(iid)
[{tid: iid}] ←	← [{tid: iid}] ←	← [{tid: iid}]
APP ← iid		

6 Implementierung

Dieses Paper beschreibt die Kernfunktionalität von REDS auf einer abstrakten Ebene. Daher wird hier nicht auf die konkrete Implementierung eingegangen, sondern auf das REDS.io Framework[a] verwiesen. Diese auf JavaScript und HTTP basierende Implementierung des REDS Konzeptes stellt Webapp-Entwicklern eine RESTful API für das Verwalten von Usern, Accounts, Domains und Entities zur Verfügung. Implementierungen des REDS Konzeptes für andere Plattformen, zum Beispiel für native Desktopanwendungen oder Apps für mobile Geräte, sind aber ebenfalls möglich.

a http://getreds.org

1 Referenzen

1 O'Reilly (2005): Design Patterns and Business Models for the Next Generation of Software
http://oreilly.com/web2/archive/what-is-web-20.html

2 Index on Censorship (24. August 2012): Web 2.0: Don't shoot the messenger
http://www.indexoncensorship.org/2012/08/internet-intermediary-liability/

3 The Guardian (7. Juni 2013): NSA Prism program taps in to user data of Apple, Google and others
http://www.guardian.co.uk/world/2013/jun/06/us-tech-giants-nsa-data

4 Whitten, Tygar (2005): Why Jonny can't encrypt - A Usability Evaluation of PGP 5.0
Designing Secure Systems that People Can Use (Seite 679ff), O'Reilly Media, ISBN: 978-0596008277

5 Diffie,Hellman (1976): "New directions in cryptography". IEEE Transactions on Information Theory Vol. IT-22
https://ee.stanford.edu/~hellman/publications/24.pdf

6 RSA Laboratories (1999): PKCS #5 v2.0: Password-Based Cryptography Standard
ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf

7 Provos, Mazières (1999): "A Future-Adaptable Password Scheme". Proceedings of 1999 USENIX Annual Technical Conference: 81–92
https://www.usenix.org/legacy/events/usenix99/provos/provos_html/

8 Percival (2012): Stronger Key Derivation via Sequential Memory-Hard Functions
http://www.tarsnap.com/scrypt/scrypt.pdf

9 NIST (2012): Secure Hash Standard (SHS), FIPS PUB 180-4
http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

10 NIST (2014): SHA-3 Standard: Permutation Based Hash and Extendable Output Functions, DRAFT FIPS PUB 202
http://csrc.nist.gov/publications/drafts/fips-202/fips_202_draft.pdf

11 NIST (2001): Announcing the Advanced Encryption Standard (AES), FIPS-197
http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

12 Bernstein (2008): Salsa20 Specification
https://cr.yp.to/snuffle/spec.pdf

13 NIST (2001): Announcing the Advanced Encryption Standard (AES), FIPS-197
http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

14 Bernstein (2008): Salsa20 Specification
https://cr.yp.to/snuffle/spec.pdf