UNIVERSITÄT DUISBURG-ESSEN
FAKULTÄT FÜR INGENIEURWISSENSCHAFTEN
ABTEILUNG INFORMATIK UND ANGEWANDTE KOGNITIONSWISSENSCHAFT

Bachelorarbeit

Entwicklung und Evaluation einer Dateiverschlüsselung im Webbrowser

Carlo Rissmöller
Matrikelnummer: 3004317

UNIVERSITÄT DUISBURG ESSEN

Offen im Denken

Abteilung Informatik und Angewandte Kognitionswissenschaft
Fakultät für Ingenieurwissenschaften
Universität Duisburg-Essen

19. März 2017

Betreuer:
Dr.-Ing. Matthäus Wander
Prof. Dr.-Ing. Torben Weis

Zusammenfassung

Das Thema dieser Arbeit ist die Entwicklung eines Prototypen zur Dateiverschlüsselung, um Daten vor dem Ausspähen durch Fremde zu schützen. Die Dateiverschlüsselung findet bereits schon im Browser statt, um eine große Kontrolle über die Daten gewährleisten zu können.
Die Grundidee einer Dateiverschlüsselung im Webbrowser wurde bereits schon von dem Webtool *Whisply* realisiert. Somit hat diese Arbeit sich zur Aufgabe gemacht, den Stand der Technik noch weiter auszubauen. Ein paar nennenswerte Errungenschaften dieser Arbeit sind die hybride Verschlüsselung, die zweistufige Implementierung der Kryptoverfahren und die statische Eigenschaft der Anwendung.
Die hybride Verschlüsselung wurde mit den Verfahren *AES-GCM-128* und *RSA-OAEP-2048* durchgeführt. Der Vorteil einer hybriden Verschlüsselung ist, dass der Dateischlüssel mit dem öffentlichen Schlüssel verschlüsselt wird. Der private Schlüssel für die Entschlüsselung hingegen kann an einer sicheren Stelle aufbewahrt werden und muss nicht erst über einen sicheren Kanal ausgetauscht werden. Praktischerweise wird zusätzlich zur Chiffrierung über *AES-GCM-128* auch noch die Integrität der Daten gesichert.
Für die verwendeten Kryptoverfahren wurde zudem eine zweistufige Implementierung umgesetzt. Es wurde die schnelle *Web Cryptography API* als Standardimplementierung und das JavaScript-Framework *forge* als Ausweichmöglichkeit genommen, da es sich bei der *Web Cryptography API* um einen offenen Standard handelt, der noch nicht von allen Browsern unterstützt wird. Eine Verschlüsselung kann unter jedem Browser garantiert werden. Der Geschwindigkeitsverlust, der bei einem Fallback entsteht, ist vertretbar.
Whisply macht sich einen Server zu Nutze, der das System beim Verschlüsselungsprozess unterstützt. Wenn der Server abstürtzt, kann es unter Umständen passieren, dass keine Verschlüsselung durchgeführt wird. Die hier vorgestellte Webanwendung arbeitet statisch. Statisch bedeutet, das die Hauptfunktionalität, also die

Verschlüsselung, nicht von der Bereitschaft anderer technischer Systeme abhängt. Zum Schluss dieser Arbeit wurde das System evaluiert. Es wurde eine Laufzeitanalyse und eine Sicherheitsanalyse durchgeführt.
Es zeigte sich, dass der Prototyp für die Plattformen *Safari*, *Firefox*, *Chrome*, *iOS Safari* und *Android Chrome* tauglich ist und für die Verschlüsselung Ergebnisse liefert, die im mittleren Feld anzusiedeln sind und für eine Nutzung im Sicherheitskontext durchaus angemessen sind. Desweiteren hat sich die *Web Cryptography API* als schnelle Kryptoimplementierung erwiesen und begründet somit die Vorgehensweise der zweistufigen Implementierung. Erst der Fallback auf ein JavaScript-Framework führt zu Geschwindigkeitsverlusten.
Die Sicherheitsanalyse ergab, dass die Sicherheitsanforderungen zufriedenstellend umgesetzt wurden, es aber auch noch einige Schwachstellen gibt, die man in zukünftigen Arbeiten untersuchen könnte. Zugewinn an Sicherheit bekam das System durch die hybride Verschlüsselung und die Statik der Anwendung. Schwachstellen wurden bezüglich der hochgeladenen Datei gefunden. So ist es beispielsweise möglich, den Inhalt einer Datei abschätzen zu können.

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Duisburg, 19. März 2017 _____

Unterschrift

Inhaltsverzeichnis

1 Einleitung

Schon Anfang der 1990er Jahre prophezeite die IT-Branche ein Entstehen des Cloud-Computing. [2]

Das Cloud-Computing bietet ein breitgefächertes Spektrum von IT-Infrastrukturen an, die über ein Netz aus der Ferne genutzt werden können. Die IT-Infrastrukturen umfassen Rechenkapazität, Datenspeicher, Datensicherheit, Netzkapazitäten und auch fertige Software. Somit kann der lokale Rechner entlastet werden und die nötigen Ressourcen werden einem zur Verfügung gestellt. [3]

Ein Beispiel für Cloud-Computing ist *Dropbox*. Mit Hilfe von *Dropbox* können Daten extern gespeichert und verwaltet werden.

Cloud-Computing ist praktisch, aber es gibt auch Nachteile. Spätestens nach der NSA-Affäre um Edward Snowden dürfte klar sein, das Daten im Netz vor dem Ausspähen nicht mehr sicher sind.

Zwar bieten manche Cloud-Betreiber eine verschlüsselte Speicherung der Daten an, aber das bedeutet dann auch, das sie im Besitz des Schlüssels sind. Sie haben also die Möglichkeit jederzeit auf die Daten zuzugreifen und sie zu missbrauchen. Auch staatliche Institutionen, besonders im Ausland, können den Zugriff auf Daten aufgrund der Rechtsgrundlage fordern. [1]

Der Verlust der Kontrolle über die eigenen Daten ist ein Problem geworden. Daher ist das Thema Datenschutz für Cloud-Computing sehr bedeutend geworden und wird auch in Zukunft nicht wegzudenken sein.

Um einen unberechtigten Zugriff auf die Daten zu verhindern, muss der Anwender diese bereits vor dem Hochladen verschlüsseln.
Die Motivation dieser Arbeit ist sich mit dem Thema zu beschäftigen und eine informationstechnische Lösung zu finden.

1.1 Aufgabenstellung

Im Rahmen dieser Bachelorarbeit soll ein Framework entwickelt, dass ermöglicht Dateien im Browser zu verschlüsseln und dann anschließend in einen Cloud-Storage hochzuladen. Eine zu verschlüsselnde Datei besteht aus den eigentlichen Nutzdaten.

Es soll angenommen werden, das Cloud-System ist für mehrere Nutzer skalierbar und besitzt einen unendlich großen Speicherplatz.

Der Anwender soll in der Lage sein, über eine statische Webanwendung mit einem Cloud-Server zu kommunizieren. Alle kryptographischen Operationen sollen clientseitig passieren. Die Funktionalität des Programms soll nicht von dem potenziellen Ausfall eines anderen Servers beeinträchtigt werden. Ausschließlich die Speicherung der Daten soll auf dem Cloud-Server stattfinden.

Die Webanwendung soll für die Verschlüsselung unter Verwendung der *Web Cryptography API* und eines *JavaScript Crypto-Frameworks* arbeiten. Das in dieser Arbeit angesprochene *JavaScript Crypto-Framework* ist *forge*. Es soll eine zweistufige Implementierung angestrebt werden. Zunächst soll versucht werden, die *Web Cryptography API* als native und schnellere Implementierung zu nehmen. Falls diese nicht unterstützt wird, soll alternativ ein *JavaScript Crypto-Framework* verwendet werden. Somit passt die Webanwendung sich den Gegebenheiten aller gängigen Browser an, da sie von diesen unterstützt werden soll. Auch mobile Browser sollen in der Lage sein, die Webanwendung auszuführen.

Für die Enkodierung der Daten soll eine hybride Verschlüsselung genommen werden. Dabei werden die eigentlichen Nutzdaten symmetrisch verschlüsselt. Der Dateischlüssel hingegen wird mit einem asymmetrischen Verfahren chiffriert.
Eine hybride Verschlüsselung hat den Vorteil, dass sie keinen sicheren Schlüsseltausch fordert. Der private Schlüssel der asymmetrischen Verschlüsselung bleibt beim Admin des Verschlüsselungsdienstes und der öffentliche Schlüssel wird im clientseitigen Code festkodiert.

Zusätzlich soll vor dem Hochladen die Integrität der Daten gesichert werden. Der Prüfwert zusammen mit der Datei soll dann auf dem Cloud-Speicher hinterlegt werden.

Die Schnittstelle des Cloud-Speichers in dieser Arbeit ist eine HTTP-Schnittstelle. Diese kann als hinreichend generisch angesehen werden, da jeder Cloud-Betreiber ohne großen Aufwand eine HTTP-Schnittstelle zur Verfügung stellen kann.

Ein Hochladen von sehr großen Dateien (1GByte oder mehr) soll durch *chunked Upload* unterstützt werden. *Chunked Upload* bedeutet, eine Datei in mehrere Blöcke zu zerlegen.

Diese Datei-Blöcke werden dann einzeln verschlüsselt und direkt hochgeladen.

Eine Entschlüsselung der Daten soll durch den *Admin* des *Encryption Service* eingeleitet werden. Nur er hat berechtigten Zugriff auf die privaten Schlüssel. Die privaten Schlüssel werden lokal auf seinem Rechner aufbewahrt. Eine Definition der Rolle als *Admin* ist im Kapitel 2 zu den Grundlagen zu finden.

Das Endergebnis dieser Arbeit soll im Hinblick auf die Laufzeit der kryptographischen Operationen und des Gesamtsystems evaluiert werden, die auf den verschiedenen aktuellen Plattformen erreicht wird. Zudem soll die Sicherheitsgüte des Systems bewertet werden.

1.2 Stand der Technik

Der aktuelle Forschungsstand zeigt, dass es bereits Realisierungen einer Verschlüsselung für das Hochladen von Dateien in einen Cloud-Storage gibt. Eine dieser Realisierungen ist *Whisply*.

Whisply wurde Ende des Jahres 2015 in einer Early-Access-Version vorgestellt. Es arbeitet über den Browser und nutzt die *Web Cryptography API* als Verschlüsselungswerkzeug. Die Chiffrierung funktioniert über eine doppelte AES-Verschlüsselung. Die Datei wird mit einem *FileKey* verschlüsselt. Der *FileKey* wiederum wird mit einem *UnlockingKey* verschlüsselt. Dieser *UnlockingKey* wird nun über einen sicheren Kanal ausgetauscht. Nach dem Verschlüsselungsprozess bekommt der Nutzer einen Link für den Zugang zur Datei bereitgestellt. Dieser Link kann dann auch verwendet werden, um eine Datei zwischen mehreren berechtigten Nutzern zu teilen. *Whisply* ist kein File-Hosting-Dienst, aber es wird eine kleinere Liste von Cloud-Betreibern unterstützt. Darunter *Dropbox*, *Google Drive* und *OneDrive*. [15]

Auch gibt es Realisierungen wie *Cryptomator*, die über eine eigenständige Software arbeiten. Diese Software muss dann zuvor auf dem Rechner installiert werden.

Cryptomator ist Anfang des Jahres 2016 in der finalen Version 1.0 freigegeben worden. Es handelt sich um eine Mischung aus Java-Anwendung mit JavaFX-Frontend und WebDAV. Ein wichtiges Kriterium bei der Entwicklung von *Cryptomator* war, die Software vertrauenswürdig zu gestalten. Anwender könnten sich selbst davon überzeugen, dass die Software keine Backdoors besitzt, da sie OpenSource ist. *Cryptomator* nutzt für die Schlüsselableitung *Scrypt* und für die Verschlüsselung *AES* mit einem 256 Bit langen Schlüssel. Der Anwender gibt ein eigenes Passwort an, das zu Generierung des

eigentlichen Schlüssels verwendet wird. Dateien oder Ordner können jetzt mit Hilfe des Schlüssels und *AES* verschlüsselt werden. Für die Entschlüsselung der Daten wird ein virtuelles Laufwerk über WebDAV in das Dateisystem eingehangen. *Cryptomator* unterstützt auch die Cloud-Anbieter *Dropbox*, *Google Drive* und *OneDrive* und noch weitere. [7]

1.3 Vorgehensweise und Struktur der Arbeit

Als erstes werden die zum Verständnis der Arbeit notwendigen Grundlagen geschaffen. Eine Einführung in Thematik Datensicherheit soll den Leser über das Konzept und die Umsetzung von Sicherheit in informationstechnischen Systemen informieren. Weiterhin werden kryptographische Verfahren vorgestellt, die für das Konzept und die Implementierung des Frameworks erforderlich sind.

Dann wird die Problemstellung auf Grundlage der Aufgabenstellung ermittelt. Es wird ein Sicherheitskonzept für die Umsetzung des Datenschutzes und kryptographischer Verfahren erabeitet. Aufbauend auf dem Sicherheitskonzept wird das Framework konzipiert, sodass es die Problemstellung löst.

Anschließend werden die zur Implementierung verwendeten Technologien vorgestellt. Es wird ein Gesamtüberblick über die Architektur des Systems gegeben und interessante Stellen in der Umsetzung der Problemstellung werden referenziert und erläutert.

Im vorletzten Kapitel wird die Laufzeit des Systems evaluiert. Dazu wird die Gesamtlaufzeit auf den verschiedenen Plattformen gemessen und ihre Güte bestimmt. Darüber hinaus wird der Einfluss der einzelnen kryptographischen Implementierungen auf die Gesamtlaufzeit gewichtet. Zuletzt wird eine Sicherheitsanalyse des Systems gemacht, die überprüft, ob die Sicherheitsziele aus dem Sicherheitskonzept ausreichend umgesetzt wurden.

Im letzten Kapitel wird die Arbeit zusammengefasst und ein Ausblick gegeben.

2 Grundlagen

In diesem Kapitel werden die zum Verständnis der Arbeit notwendigen Grundlagen geschaffen. Zunächst wird der Begriff *Datensicherheit* in informationstechnischen Systemen definiert. Danach werden kryptographische Verfahren vorgestellt, die für die Implementierung benötigt werden. Am Ende wird schließlich die Rolle des Administrators und des Nutzers im entworfenen System beschrieben.

2.1 Datensicherheit

Die Datensicherheit in der Informationstechnik beschäftigt sich mit dem ausreichenden Schutz von Daten vor Verlust, Manipulationen und anderen Bedrohungen. Der Begriff umfasst nach Definition der *Common Criteria for Information Technology Security Evaluation* die Sicherheitsziele *Verfügbarkeit*, *Vertraulichkeit* und *Integrität*. [12]

Verfügbarkeit ist gegeben, wenn ein legitimer Benutzer jederzeit Zugriff auf Ressourcen oder Dienste hat. [12]

Vertraulichkeit meint Daten und Informationen vor dem Zugriff von unbefugten Dritten zu schützen. [12]

Integrität bedeutet den Inhalt einer Information sicher zu stellen. Eine Information darf nicht durch Unbefugte verändert werden. [12]

Sicherheitsmaßnahmen, um die Sicherheitsziele zu erfüllen, sind Kryptographieverfahren zur Verschlüsselung und zur Integritätsprüfung, aber auch Firewalls oder Backups um die Verfügbarkeit eines Systems zu garantieren.

2.2 Kryptographische Hashfunktion

Eine Hashfunktion ist eine Funktion $h : X \to Y$, die durch zwei Eigenschaften definiert werden kann. Zum einen bildet die Hashfunktion h eine beliebige Bitlänge $x \in X$ auf eine feste Bitlänge $h(x) \in Y$ ab. Das Bild $h(x)$ wird auch *Fingerabdruck* genannt. Zum anderen soll $h(x)$ in polynomialer Zeit zu berechnen sein, wenn x und h bekannt sind. [17]

Eine kollisionfreie Hashfunktion erweitert jetzt die Definition einer Hashfunktion, um folgende Eigenschaften:

Eine kollisionsfreie Hashfunktion ist schwach kollisionresistent für eine Nachricht $x \in X$, wenn es praktisch unmöglich ist eine Nachricht $x' \in X$ mit $x' \neq x$ zu finden, sodass $h(x) = h(x') \in Y$ gilt. [17]

Eine kollisionsfreie Hashfunktion wird als stark kollisionsfrei bezeichnet, wenn es praktisch unmöglich ist, zwei Nachrichten $x \in X$ und $x' \in X$ mit $x' \neq x$ und $h(x') = h(x) \in Y$ zu finden. [17]

Weiterhin wird für die Hashfunktion die Eigenschaft einer *Ein-Weg-Funktion* verlangt, die es praktisch unmöglich macht zu einem gegebenen Hashwert $z \in Y$ eine Nachricht $x \in X$ zu finden für die gilt $z = h(x) \in Y$. [17]

Sind alle Eigenschaften erfüllt, so handelt es sich um eine kryptographische Hashfunktion.

Kryptographische Hashfunktionen werden verwendet, um die Integrität einer Nachricht zu sichern. Ändert sich also die Nachricht, so ändert sich auch der Hashwert. Auf diese Weise können Modifikationen an einer Nachricht erkannt werden. Digitale Signaturen arbeiten aus Effizienzgründen mit kryptographischen Hashfunktionen. Es wird nicht die ganze Nachricht signiert. Ausreichend ist die Bestimmung und Signierung des Hashwertes. Auch der *Message Authentication Code* macht Verwendung von kryptographischen Hashfunktionen, die zusätzlich durch einen bestimmten geheimen Schlüssel parametrisiert werden. Er wird als Alternative zur digitalen Signatur verwendet. [17]

Es gibt einige Implementierungen für kryptographische Hashfunktionen, die aber alle unterschiedlich sicher sind. Ein Beispiel für eine kryptologische Hashfunktion ist der *Secure Hash Algoritthm (SHA)*. Der *Secure Hash Algorithm (SHA)* wurde im Jahr 1993 vom NIST vorgestellt und in den Folgejahren weiterentwickelt. Die Hauptschleife des Algorithmus besteht aus vier Runden mit jeweils 20 Schritten. [17] Der BSI empfiehlt die Nutzung der kryptographischen Hashfunktionen *SHA-256*, *SHA-512/256*, *SHA-384*

und *SHA-512*, sowie *SHA3-256*, *SHA3-384* und *SHA3-512*. Diese gelten nach heutigem Kenntnisstand als sicher. [6]

Der bekannteste Angriff auf Hashfunktionen beruht auf dem *Geburtstagsparadoxon*. Mit einer Wahrscheinlichkeit ϵ lässt sich die Anzahl der Berechnungen durch $k = c_\epsilon \cdot \sqrt[2]{n}$ (n ist die Anzahl der verschiedenen Bilder der Hashfunktion) mit $c_\epsilon = \sqrt[2]{2 \cdot \ln \frac{1}{1-\epsilon}}$ abschätzen, die benötigt werden, um mindestens eine Kollision der Hashfunktion zu finden. Somit kann die Komplexität der Berechnungen reduziert werden. Wird mindestens eine Kollision gefunden, so ist die Eigenschaft der Kollisionsresistenz verletzt und die Hashfunktion ist angreifbar. Im Folgenden wird angenommen die Wahrscheinlichkeit ist $\epsilon = 0,5$. Bei einer Hashfunktion, die einen 128 Bit Hashwert erzeugt, ist mit einer Wahrscheinlichkeit von 50% bei $k = 1,17 \cdot \sqrt[2]{2^{64}}$ Berechnungen mindestens eine Kollision dabei. [17]

2.3 Symmetrische Kryptographie

Bei symmetrischer Kryptographie wird zur Ver-und Entschlüsselung der gleiche Schlüssel verwendet. Dieser geheime Schlüssel muss allen an dieser Verschlüsselung Beteiligten bekannt sein. Der Austausch des Schlüssels findet daher über einen sicheren Kanal statt. Es gab bereits schon viele symmetrische Chiffren in der Antike, z.B. *Vignere* oder *Caesar*. Aber auch besonders in der heutigen Technik werden bevorzugt symmetrische Verfahren gewählt, da der Rechenaufwand einer Ver-und Entschlüsselung nicht sehr hoch ist. [14]

Bei symmetrischer Kryptographie wird zwischen zwei verschiedenen Verschlüsselungsarten unterschieden. Es gibt die blockweise Verschlüsselung und eine Stromverschlüsselung.

Für die blockweise Verschlüsselung wird die Nachricht zunächst in Blöcke aufgeteilt. Blöcke, die zu klein sind, werden anschließend mit einem *Padding* aufgefüllt. Die Blöcke einer *AES*-Verschlüsselung sind beispielsweise 128 Bit groß. Die einzelnen Blöcke werden dann zusammen mit dem symmetrischen Schlüssel und der Verschlüsselungsfunktion chiffriert. [14]

Bei einer Stromverschlüsselung wird die Nachricht als ein Strom von Bits, Bytes oder Zeichen angesehen. Dieser Nachrichtenstrom wird dann zusammen mit dem Schlüsselstrom und einer Operation bitweise, byteweise oder zeichenweise verrechnet. Das Ergebnis der Operation ist dann der Chiffrestrom. Die Operation ist üblicherweise eine Addition modulo 2 oder modulo 26.
Ein wichtiges Kriterium ist, dass die Verschlüsselungsfunktion bijektiv ist. Verschlüsselung und Entschlüsselung müssen also einander umkehrbar sein.

Der Schlüsselstrom sollte eine echte Zufallsfolge ohne Wiederholungen sein. Nur dann kann eine perfekte Sicherheit erreicht werden.
Beispiel für eine Stromverschlüsselung, die früher im zweiten Weltkrieg genutzt wurde ist das *One-Time-Pad*. Beim *One-Time-Pad* wird der Nachrichtenstrom bitweise mit dem Schlüsselstrom verxodert. Eigentlich handelt es sich um eine perfekte Cipher, sofern der Schlüsselstrom wirklich zufällig ist. Falls es Wiederholungen gibt, so lässt sich das *Pad* duch ein paar Operationen im Gleichungssystem herauskürzen und die Verschlüsselungsfunkion ist angreifbar.
Besonders in modernen aktuellen Systemen wird eine Stromchiffre benutzt. So wird beispielsweise in der Telekommunikation die Funkübertragung durch die Stromchiffre *A5* verschlüsselt. [14]

Eine symmetrische Verschlüsselung lässt sich in verschiedenen Modi betreiben. Es gibt den *Electronic Code Book (ECB)*-Modus, der die einfachste Betriebsart darstellt. Hier wird jeder Klartextblock unabhängig von anderen Klartextblöcken verschlüsselt. Ein großer Nachteil ist, dass gleiche Klartextblöcke auf den gleichen Ciffretextblock abgebildet werden. Ein Angreifer kann dadurch unerwünschte Hinweise erhalten.
Andere sicherere Betriebsarten sind *Ciper Block Chaining (CBC)*, *Cipher Feedback (CBF)*, *Output Feedback (OFB)* und *Galois-Counter-Mode (GCM)*. Bei diesen Betriebsarten gibt es eine Abhängigkeit zwischen einzelnen Blöcken. In dieser Arbeit wird die symmetrische Verschlüsselung im *Galois-Counter-Mode* stattfinden. [8]

Ein Beispiel für ein symmetrisches Verschlüsselungsverfahren ist *AES*. *AES* arbeitet auf Grundlage einer symmetrischen Blockchiffre. Die Nachrichtenblöcke werden zu einer Länge von 128 Bit aufgeteilt. Der Schlüssel kann eine Länge von 128, 192 oder 256 Bit haben. Der Algorithmus *AES* ist ein rundenbasiertes Verfahren. In jeder Runde, bis auf die Initialrunde, werden Transformationsoperationen durchgeführt und das Ergebnis mit Rundenschlüsseln verrechnet. In der letzten Runde des Verfahrens entfällt der Byte-Intermix. Die Anzahl der Runden hängt von der Schlüssellänge ab. Bei 128 Bit sind es 10 Runden, bei 192 Bit 12 Runden und bei 256 Bit 14 Runden.
Im Folgenden wird die AES-Verschlüsselung anhand einer Runde etwas genauer erklärt. Ganz zu Anfang hat man den Datenblock d_{r-1} von 128 Bit. Der Datenblock wird in einer 4x4-Matrix dargestellt, also steht in einer Zelle der Matrix ein Byte. Auf dieser Daten-Matrix werden zunächst einige Byte-Operationen ausgeführt. Diese Operationen beinhalten eine Substitution, eine Permutation und einen Intermix.
Die Substitution besteht im Wesentlichen aus zwei Teilschritten.
Als erstes wird jedes Byte der Matrix durch sein inverses multiplikatives Element ersetzt. Die Ersetzung ist eine nicht lineare Transformation. Die Arithmetik wird dabei durch einen Galois-Körper GF(2^8) hinsichtlich des irreduziblen Modularpolynoms $M(x) = x^8 + x^4 + x^3 + x + 1 = 100011011$ definiert.

Im einem zweiten Teilschritt wird eine lineare Transformation durchgeführt. Es findet eine Multiplikation mit einer festen 8x8-Matrix und eine anschließende Addition mit einer Konstante aus GF(2^8) statt.
In der Praxis wird eine Substitutiontabelle genutzt, die alle Ersetzungen durch Vorberechnungen enthält. Diese Substitutionstabelle wird auch oft *S-Box* genannt.
Bei der Permutation werden die Zeilen der Daten-Matrix um ein paar Byte-Positionen nach links verschoben, also um 0, 1, 2 oder 3 Bytes.
Der Intermix bildet eine Spalte der 4x4-Daten-Matrix auf eine Andere in gleicher Position ab. Dazu wird jede Spalte linear mit einer festen 4x4-Matrix multipliziert. Auch hier sind alle Elemente der Transformation bezüglich des irreduziblen Polynoms $M(x) = x^8 + x^4 + x^3 + x + 1 = 100011011$ im Galois-Körper GF(2^8) definiert.
Nach den Transformationsoperationen wird zum Ergebnis ein Teilschlüssel bitweise addiert. Die Teilschlüssel werden aus dem *AES*-Schlüssel abgeleitet. Als Ergebnis der Addition auf Bitebene entsteht der Datenblock d_r. Dieser Datenblock wandert jetzt wieder als Eingangswert in die nächste Runde. In der letzten Runde ist der Ausgangsdatenblock d_{Nr} (wobei Nr die Anzahl der Runden ist) der resultierende Chiffretext des *AES*-Algorithmus.
Zur Entschlüsselung wird das oben beschriebene Verfahren in der umgekehrten Reihenfolge durchlaufen. Die Teilschlüssel werden in der umgekehrten Reihenfolge bereitgestellt und im Transformationsblock werden die Umkehrfunktionen benutzt.[14]

2.4 Asymmetrische Kryptographie

Asymmetrische Kryptographie ist ein kryptographisches Verfahren zur Verschlüsselung von Daten. Das Verschlüsselungssystem ist asymmetrisch, da es ein Schlüsselpaar bestehend aus zwei verschiedenen Schlüsseln gibt. Dabei wird der öffentliche Schlüssel für die Verschlüsselung der Daten und der private Schlüssel für die Entschlüsselung der Daten verwendet.
Im Gegensatz zur symmetrischen Kryptographie bietet ein asymmetrisches Verfahren einige Vorteile. Bei einer Kommunkation muss kein geheimer Schlüssel übertragen werden. Der private Schlüssel bleibt auf der Seite des Erzeugers. Zusätzlich kann mit dem privaten Schlüssel eine digitale Signatur erstellt werden. Die digitale Signatur ist genau einer Person zuordenbar. Somit kann die Verbindlichkeit einer Nachricht gewährleistet werden. [14]

In einer Beispielsituation soll die Kommunikation zwischen zwei Kommunikationspartnern, hier *Alice* und *Bob*, erläutert werden (siehe auch Abbildung: 2.1). *Alice* möchte *Bob* eine geheime Nachricht senden.

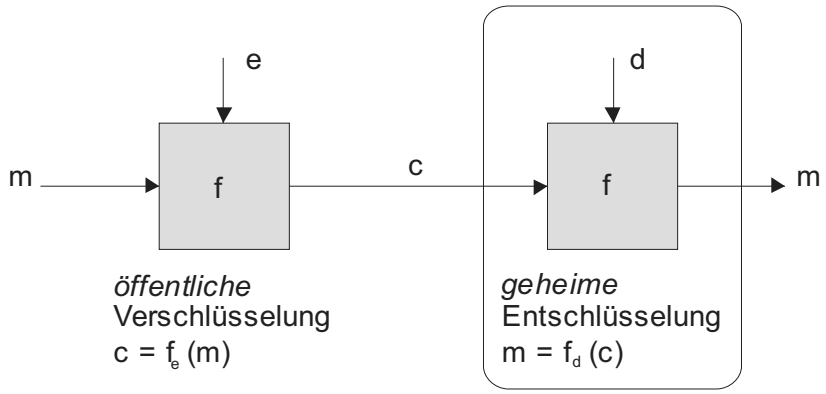


Abbildung 2.1: Schema einer asymmetrischen Verschlüsselung.[14]

Bob erzeugt dazu auf seiner Seite ein Schlüsselpaar (e, d). Den privaten Schlüssel d behält *Bob* selber und der Schlüssel e wird veröffentlicht.
Alice muss sich zunächst davon überzeugen, dass der öffentliche Schlüssel e_{Bob} auch wirklich von *Bob* stammt. Über ein Zertifikat kann die Herkunft verifiziert werden.
In einem nächsten Schritt verschlüsselt *Alice* mit Hilfe von e_{Bob} und einer öffentlichen Verschlüsselungsfunktion f ihre Nachricht. Diese verschlüsselte Nachricht wird jetzt an *Bob* gesendet.
Bob kann nun in einem geschützten Bereich mit seinem privaten Schlüssel d_{Bob} und der gleichen Verschlüsselungsfunktion f, die Nachricht wieder entschlüsseln. [14]

Auch bei der asymmetrischen Verschlüsselung gilt: Je größer der Schlüssel, desto sicherer ist die Verschlüsselung und desto praktisch aufwendiger wäre ein Angriff auf diese Verschlüsselung.
Eine zu verschlüsselnde Nachricht kann maximal nur so lang wie der Schlüssel sein. Ist die Nachricht länger, so wird sie blockweise verschlüsselt. [14]

Ein asymmetrisches Verfahren wird heutzutage in vielen modernen Kommunikationsprotokollen zum Verschlüsseln und zum digitalen Signieren eingesetzt. Das Schlüsselpaar wird vorher generiert und über ein Zertifikat ausgeliefert.
Der defacto-Standard für ein asymmetrisches Verfahren in Protokollen ist *RSA*. [14]

Verwendet man *RSA*, so müssen zunächst der öffentliche und der private Schlüssel erzeugt werden.
Es werden zwei unabhängige Primzahlen p und q gewählt, die weit auseinander liegen. Aus dem Produkt der beiden Primzahlen wird dann das *RSA*-Modul $N = p \cdot q$ gebildet. Mit Hilfe der *Eulerschen Funktion* ϕ und einem Hilfssatz kann jetzt $\phi(N) = (p-1) \cdot (q-1)$ berechnet werden.
Als nächstes wird eine Zahl e gesucht, die teilerfremd zu $\phi(N)$ ist. Diese Zahl e ist der

öffentliche Schlüssel.
Jetzt fehlt nur der private Schlüssel, der als multiplikatives Inverses von e bezüglich des Moduls $\phi(N)$ bestimmt wird. Mit $e \cdot d + k \cdot \phi(N) = 1$ und dem erweiterten euklidischen Algorithmus kann der private Schlüssel d ausgerechnet werden.
Nach den Berechnungen müssen p und q sicher vernichtet werden, damit ein Angreifer nicht in der Lage ist $\phi(N)$ zu berechnen. Ohne die Kenntnis von p und q ist $\phi(N)$ nicht in absehbarer Zeit zu berechnen, da das Faktorisierungsproblem zweier Primzahlen in der Komplexitätsklasse *NP-vollständig* liegt. [10]

Mit dem öffentlichen Schlüssel e, dem privaten Schlüssel d und dem Modul N können jetzt Nachrichten verschüsselt und wieder entschlüsselt werden. [10]

Über $c \equiv m^e \pmod{N}$ kann eine Nachricht m verschlüsselt werden. Es entsteht das Chiffrat c, das übertragen werden kann. [10]

Die Entschlüsselung des Chiffrat c kann über $m \equiv c^d \pmod{N}$ erfolgen. Man erhält die Nachricht m zurück. [10]

Auch für die Verwendung von *RSA* gibt es verschiedene Modi. In dieser Arbeit wurde sich für *RSA-OAEP* als asymmetrischer Verschlüsselungsmodus entschieden. Durch *OAEP* werden die zu verschlüsselnden Daten zunächst mit einem Padding aufgefüllt. Anschließend werden sie über die Falltürpermutation *RSA* verschlüsselt. [13]

2.5 Hybride Kryptographie

Hybride Kryptographie ist eine Zusammensetzung von symmetrischer und asymmetrischer Kryptographie und nutzt so die Vorteile beider Verfahren. [14].

Asymmetrische Verfahren sind viel sicherer als symmetrische Verfahren, da die hergeleiteten Schlüssel auf dem NP-vollständigen *Faktorisierungsproblem* beruhen. Allerdings wäre ein Verschlüsselung mit einem asymmetrischen Verfahren sehr komplex und etwa um den Faktor 1000 langsamer als symmetrische Verfahren. [14]

Daher wird die eigentliche Datei mit symmetrischer Kryptographie verschlüsselt und nur der symmetrische Dateischlüssel wird mit dem öffentlichen asymmetrischen Schlüssel gesichert. Jetzt können die verschlüsselte Datei und der Dateischlüssel sicher übertragen werden. Der private Schlüssel kann dann zur Entschlüsselung des symmetrischen Schlüssel verwendet werden. [14]

Hybride Kryptographie wird heute in vielen Bereichen eingesetzt. Zum Beispiel in Sicherheitsprotokollen wird zu Anfang in einer Authentisierungsphase ein symmetrischer

Schlüssel ausgehandelt. Dieser wird dann von beiden Seiten für die Übertragung der Daten verwendet. [14]

Der defacto-Standard einer hybriden Verschlüsselung ist *AES* als symmetrische Verschlüsselung und *RSA* für die asymmetrische Verschlüsselung.

2.6 Rolle des Administrators

Jedes Sicherheitssytem braucht eine Entität, der vertraut werden kann. Das in dieser Arbeit beschriebene System nutzt den Administrator als vertrauenswürdige Entität.

Der Administrator ist nach allgemeiner Definition ein spezieller Benutzer mit erweiterten Rechten. Er ist berechtigt ein IT-System aufzusetzen, es während des Betriebs zu verwalten, es zu überwachen und Wartungen durchzuführen. [5]

Im Folgenden wird die Abkürzung *Admin* für *Administrator* benutzt.

Im zu entwerfenen System kann ein Admin selbstverständlich wie ein Standardnutzer auch Dateien verschlüsseln und sie dann hochladen.

Zusätzlich ist er für die Erzeugung des asymmetrischen Schlüsselpaars zuständig. Der private Schlüssel wird sicher lokal auf seinem Rechner gespeichert. Somit ist nur er in der Lage, verschlüsselte Dateien auch wieder zu entschlüsseln. Über eine Entschlüsselungs-Webseite kann der Admin Dateien herunterladen und dechiffrieren.

2.7 Rolle des Nutzers

Ein Nutzer zeichnet sich durch die intendierte Verwendung der Funktion des Systems aus.

Jedermann hat Zugang zum System und kann es verwenden. Die einzige Voraussetzung ist eine Anbindung an das Internet, um die Webschnittstelle aufrufen zu können. Zu den Nutzern können Privatnutzer gehören, aber auch Beschäftigte in Unternehmen.

Der Nutzer des Systems hat im Gegensatz zum Admin eingeschränkte Rechte.
Er kann Dateien im Browser verschlüsseln und sie hochladen. Zudem kann ausgewählt werden, in welchen Cloud-Speicher die Dateien hochgeladen werden sollen.
Der Nutzer ist nicht berechtigt, Dateien zu entschlüsseln.

3 Konzept

Dieses Kapitel liefert einen Lösungsansatz für die am Anfang formulierte Aufgabenstellung der Arbeit. Zunächst werden in der Problemstellung die Anforderungen an die Entwicklung des Frameworks genannt, die entscheidend für die Lösung der Aufgabenstellung sind. Anschließend wird das Sicherheitskonzept beschrieben. Im Sicherheitskonzept werden die Realisierungen des Datenschutzes durch kryptographische Operationen besprochen. Aufbauend auf der Problemstellung und dem Sicherheitskonzept wird ein System konzipiert, das den Sicherheitsanforderungen genügt und die Problemstellung löst. Dazu gibt es einen kurzen Überblick über den Aufbau des Systems. Die Anbindung an den Cloud-Server wird vorgestellt und die einzelnen Komponenten zur Entwicklung der Client-Bibliothek werden im Detail erläutert. Auch wird die Funktion des Webservers für das Konzept des Systems erläutert. Am Ende des Kapitels werden noch denkbare Anwendungszenarien innerhalb der Client-GUI aufgeführt und die Plattformunabhängigkeit des Clients wird begründet.

3.1 Problemstellung

Die Anforderungen, die für die Entwicklung des Frameworks unter Berücksichtigung der Aufgabenstellung aus Kapitel 1.1 erfüllt werden müssen, sind:

1. Clientseitige hybride Verschlüsselung von Daten durch eine Webanwendung zur Gewährleistung des Schutzes der Daten vor fremdem Zugriff.
2. Hochladen von Daten in den Cloud-Storage und Unterstützung von *chunked Upload* für größere Dateien.
3. Herunterladen von Daten aus der Cloud.
4. Clientseitige Entschlüsselung von Daten durch den Admin.
5. Gewährleistung der Integrität der Daten durch kryptographische Hashfunktionen.
6. Plattformunabhängigkeit.

7. generische Cloud-Schnittstelle

3.2 Sicherheitskonzept

Das Kapitel Sicherheitskonzept stellt die Anforderungen, die zur Gewährleistung der Sicherheit des Frameworks notwendig sind, dar und beschreibt, wie diese erfüllt werden können. Dazu gehört der Entwurf des Datenschutzes durch die Verschlüsselung und Integrität der Daten.

3.2.1 Entwurf des Datenschutzes

Hauptzweck der Entwicklung des Frameworks ist die Umsetzung des Schutzziels *Vertraulichkeit*. Die Daten sollten vor Zugriff Dritter, insbesondere vor Zugriff des Cloud-Betreibers geschützt sein.

Der benötigte Datenschutz wird durch die Verschlüsselung und die Integrität der Daten erreicht.

Eine Verschlüsselung sorgt für den sicheren Transfer der Daten in die Cloud und für eine anschließend sichere persistente Speicherung der Daten.

Die Integrität der Daten stellt sicher, dass die Daten, während sie gespeichert auf der Cloud sind und während der Übertragung zwischen Client und Server, keiner Modifikation durch Fremdeinwirkungen unterliegen.

Die hier verwendeten kryptographischen Verfahren zur Verschlüsselung und Integrität der Daten müssen nach aktuellem Kenntnisstand als sicher gelten.

3.2.1.1 Verschlüsselung von Daten

Bei der hier gewählten Verschlüsselungsfunktion sollte es sich um eine *hybride Verschlüsselung* (Kapitel 2.5) handeln.
Für jede Datei sollte ein neuer individueller Schlüssel generiert werden. Mit diesem Schlüssel sollen die Nutzdaten einer Datei über ein symmetrisches Verschlüsselungsverfahren chiffriert werden. Anschließend ist ein asymmetrisches Kryptographieverfahren zu wählen, um den Dateischlüssel mit dem öffentlichen Schlüssel zu sichern.

Das asymmetrische Schlüsselpaar bestehend aus dem öffentlichen und dem privaten Schlüssel soll nur genau einmal auf dem Rechner des Admin generiert und dann anschließend für die Verschlüsselung aller Dateien verwendet werden. Die Verwaltung der Schlüssel sollte nur durch den Admin erfolgen. Der öffentliche Schlüssel ist an eine geeignete Stelle auszulagern, auf die jeder Client Zugriff hat. Der private Schlüssel wird lokal auf dem Rechner des Admin gesichert.

Selbstverständlich muss die gewählte hybride Verschlüsselungsfunktion auch wieder umkehrbar sein, um die Daten entschlüsseln zu können. Eine Entschlüsselung darf nur durch den Admin des Systems erfolgen.

3.2.1.2 Integrität der Daten

Zusätzlich zur Verschlüsselung soll die Integrität der Daten sicher gestellt werden. Hier muss ein geeignetes Verfahren gewählt werden. Nur der *Admin* soll in der Lage sein, neben der Entschlüsselung die Integrität der Daten zu kontrollieren.

3.3 Aufbau des Systems

Das in dieser Arbeit zu entwickelnde System besteht aus drei Teilen, nämlich dem Client, Speicher-Server und dem Webserver und behandelt die clientseitige Umsetzung einer Webanwendung zur Dateiverschlüsselung (siehe Abbildung 3.1).
Das Framework wird von überall aus im Internet über eine Webschnittstelle erreichbar sein. Die Webseite wird von einem Webserver ausgeliefert. Es wird eine Ver- und Entschlüsselung durch den Browser unterstützt. Dabei darf die Entschlüsselung nur durch den Admin eingeleitet werden.
Die Speicherung der Daten zusammen mit dem Metadaten, wie zum Beispiel dem Dateischlüssel, findet auf einem *Cloud*-Speicher statt.
Dank einer generischen Schnittstelle kann das Framework mit verschiedenen Speicher-Servern arbeiten.

3.4 Konzeption des Frameworks

In diesem Abschnitt findet eine genauere Betrachtung des Systems statt. Es wird unterschieden zwischen Client, Storage-Server und Webserver. Zunächst wird das Konzept einer generischen Server-Schnittstelle erklärt. Danach wird die Zusammensetzung des

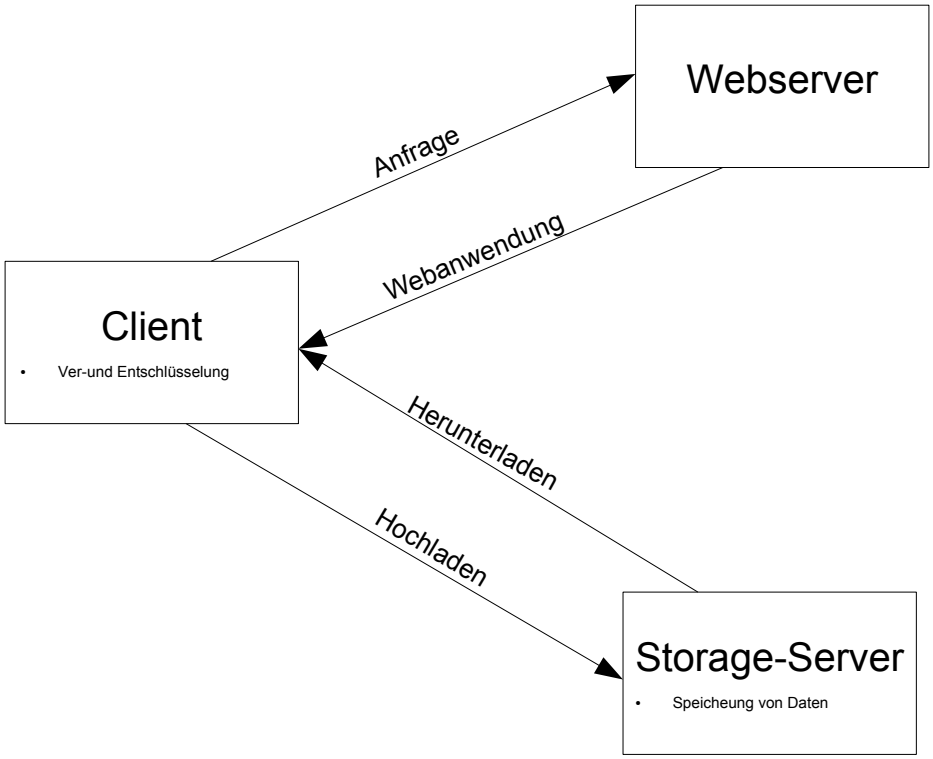


Abbildung 3.1: Der Aufbau des Systems.

Clients durch einzelne Komponenten beschrieben und es gibt eine kurze Übersicht zur Navigation innerhalb der graphischen Benutzeroberfläche des Client.
Zuletzt wird der Webserver vorgestellt und die Plattformunabhängigkeit des Clients wird angesprochen.

3.4.1 Generische Server-Schnittstelle

Eine generische Schnittstelle hat den Vorteil, dass sie von vielen Cloud-Anbietern zur Verfügung gestellt wird oder gestellt werden kann und somit einen einheitlichen Zugriff auf verschiedene Cloud-Speicher bietet.

Sie muss so gewählt werden, dass sie die nötigen Grundfunktionen zur Umsetzung der Webapplikation zur Verfügung stellt. Es müssen Daten hochgeladen und gespeichert werden können. Zudem sollte es auch möglich sein, Daten wieder herunterzuladen, sodass sie im Programmcode weiterverarbeitet werden können.
Der Cloud-Speicher muss keine hochkomplizierte Speicherstruktur aufweisen. Es reicht aus, wenn alle hochgeladenen Daten in ein Verzeichnis gepackt werden.

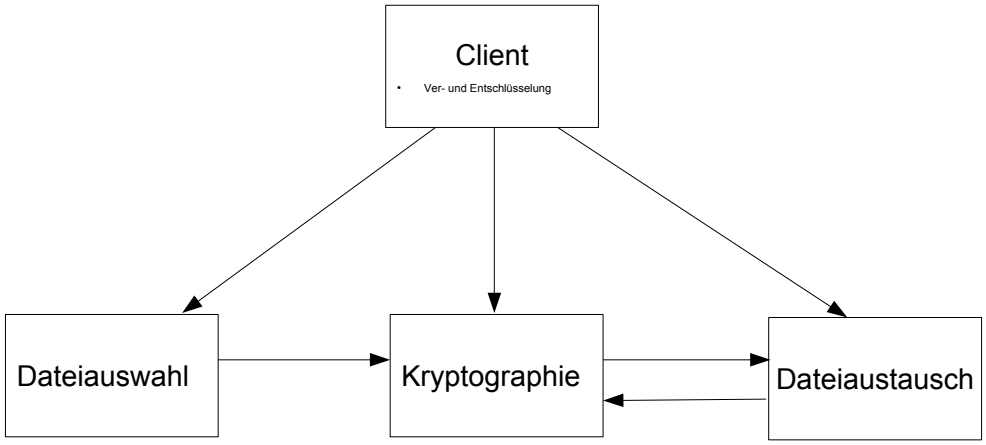


Abbildung 3.2: Die Komponentenzerlegung des Client.

In dieser Arbeit wurde *HTTP* als Server-Schnittstelle verwendet. Der Speicher kann auf einem beliebigem Verzeichnis gestartet werden. Je nachdem, wo man die hochgeladenen Daten ablegen will.
Es werden die *HTTP*-Methoden *POST* und *GET* unterstützt. Mit der POST-Methode können Daten aus einem Formular hochgeladen werden. Für den *HTTP*-Payload wird hier das übliche Datenformat zu Übertragung von Daten via *HTTP* genommen, also *multipart/form-data; boundary*.
Um Daten wieder herunterladen zu können, wird die GET-Methode verwendet. Wichtig ist, dass der übergebene URL-Pfad auf dem Dateinamen der Datei endet, die man herunterladen möchte. Die Daten im Speicher sollen auch nach dem Herunterladen erhalten bleiben. Mit einem expliten *DELETE*-Befehl könnten die Daten permanent vom Speicher gelöscht werden.
Auch wird eine *HTTPS*-Verbindung unterstützt und kann als Grundlage genommen werden.

3.4.2 Entwurf der Client-Bibliothek

Die Implementierung des Clients setzt sich aus drei Komponenten zusammen (siehe auch Abbildung: 3.2). Die erste Komponente befasst sich mit dem Auswählen der zu verschlüsselnden Datei und mit der Aufbereitung der Daten als Eingabe für die nächste Komponente. In der zweiten Komponente finden die kryptographischen Operationen statt und die letzte Komponente sorgt für den Dateiaustausch zwischen Client und Cloud-Speicher. Die zweite und dritte Komponente funktionieren wechselseitig, da Dateien verschlüsselt und hochgeladen werden können oder aber sie werden heruntergeladen und entschlüsselt.

3.4.2.1 Komponente zur Dateiauswahl

Eine Dateischnittstelle des Browsers erlaubt Zugriff auf das lokale Dateiverzeichnis des Client-Rechners und stellt verschiedene Dateioperationen zur Verfügung. Die vom Client gewählte Datei soll eingelesen werden; Allerdings nicht als Ganzes, sondern in einzelnen Blöcken, da diese für die Weiterverarbeitung günstig sind. Liest man eine große Datei ein, so kann es passieren, dass der Arbeitsspeicher des Browsers überläuft und die Webanwendung verlangsamt. Die Größe der Blöcke kann variabel gewählt werden. Es sollte darauf geachtet werden, dass sie nicht zu groß sind, da sonst die Weiterverarbeitung durch den Browser aus den oben beschriebenen Gründen zu einem Problem führen könnte. Besser sind daher kleinere Blöcke zu wählen.

Als nächstes können die Blöcke einzeln zur Verarbeitung an die nächste Komponente weitergereicht werden.

3.4.2.2 Komponente für kryptographische Verfahren

Als Eingabe dieser Komponente werden die Blöcke aus der vorherigen Komponente zur Dateiauswahl erwartet.

Hier wird die Datensicherheit der Blöcke gewährleistet. Es können Blöcke ver- und entschlüsselt werden und zusätzlich wird die Integrität gesichert. Alle kryptographischen Operationen sind durch eine Kryptoschnittstelle verfügbar und können clientseitig im Browser ausgeführt werden.

Die Chunks werden nun durch *AES-GCM* symmetrisch verschlüsselt. Der dazu nötige Schlüssel ist 128 Bit lang und wird zufällig generiert. Er dient der Verschlüsselung aller Chunks einer Datei.
AES-GCM mit 128 Bit Schlüssel ist nach aktuellem Kenntnisstand sicher. [6]
Anschließend wird der Dateischlüssel durch das asymmetrische Verfahren *RSA-OAEP* mit einem 2048 Bit Modulus verschlüsselt. Ebenfalls das Verfahren *RSA-OAEP* mit 2048 Bit gilt nach aktuellen Erkenntnissen als sicher. [6]
Die Integrität der Daten wird zusammen mit der Verschlüsselung über *AES-GCM* berechnet. Bei *AES-GCM* handelt es sich um ein kombiniertes Verfahren. Nach der Verschlüsselung werden die Geheimtextdaten und der Authentifizierungs-Tag zusammen ausgegeben.

Als Eingabe für die nächste Komponente zum Dateiaustausch werden die gerade berechneten Werte in einer einheitlichen binären Datenstruktur zusammengestellt.
Das Paket könnte folgende Werte beinhalten:

- **Dateiname:**
  Der Dateiname wird verwendet, um nach dem Entschlüsseln und Zusammenfügen der Chunks, die Datei auch wieder benennen zu können.

- **Dateigröße:**
  Die Dateigröße dient der Speicherreservierung für die ganze Datei, die nach dem Entschlüsselungsprozess wieder aus den einzelnen Chunks zusammengebaut wird.

- **Dateityp:**
  Der Dateityp hilft später nachvollziehen zu können, welche Art von Datei verschlüsselt wurde.

- **Chunknummer:**
  Die Chunknummer gibt an, um welchen Chunk es sich handelt. Hier ist es wichtig, dass die Chunks in der richtigen Reihenfolge zusammengesetzt werden, um die ursprüngliche Datei zu erhalten.

- **verschlüsselter Dateischlüssel:**
  Enthält den Dateischlüssel, um die Chunks einer Datei auch wieder richtig entschlüsseln zu können. Der Dateischlüssel ist asymmetrisch verschlüsselt.

- **Initialisierungsvektor:**
  Dieser Vektor wird für jeden Chunk zufällig generiert und für die Entschlüsselung mit *AES-GCM* verwendet.

- **Authentifizierungs-Tag:**
  Der Authentifizierungs-Tag hält die Integrität jedes einzelnen Chunks fest. Falls die Integrität nicht stimmt, wird der Entschlüsselungsprozess abgebrochen werden und es wird ein Fehler auf der Browserkonsole ausgegeben.

- **verschlüsselter Chunk:**
  Hier befinden sich die symmetrisch chiffrierten Nutzdaten eines Blockes einer Datei.

Um die Datenstruktur wieder korrekt auslesen zu können, wird die Länge der einzelnen Felder durch ein vorangestelltes Feld gespeichert. Das vorangestellte Feld ist von einer festen Größe und kann somit nach dem Auslesen übersprungen werden.
Die Felder Dateiname, Dateigröße, Dateityp und Chunknummer werden als UTF-16 Zeichenkette abgespeichert. Hier muss allerdings angemerkt werden, dass mit einer Zeichenkodierung von einem Byte pro Zeichen gerechnet wird. Also werden Zeichen, die mit mehr als 1 Byte kodiert sind, fehlinterpretiert. Da es sich in dieser Arbeit aber um einen Prototypen handelt, kann diese Einschränkung geduldet werden.

Der Wert des Feldes Dateityp wird zudem als MIME abgespeichert.
Die restlichen Felder der Datenstruktur, also verschlüsselter Dateischlüssel, Initialisierungsvektor, Authentifizierungs-Tag und verschlüsselter Chunk sind im Binärformat.

Wird die oben beschriebene Datenstruktur verwendet, so werden Dateiname, Dateigröße, Dateityp und verschlüsselter Dateischlüssel mehrfach abgespeichert. Dabei würde es ausreichen, diese Werte zum ersten Chunk einer Datei zu speichern. Ein Nachteil ist dann aber, dass zwei unterschiedliche Formate entstehen würden und es müsste eine Fallunterscheidung gemacht werden, um die Datenstruktur nach dem Herunterladen auch wieder richtig entpacken zu können. In dieser Arbeit wurde sich daher für ein einheitliches Format entschieden und eine Mehrfachspeicherung in Kauf genommen.

Schwachstellen der binären Datenstruktur sind, dass bestimmte Werte nicht verschlüsselt werden und zusätzlich fälschungssicher sind. Ein Angreifer könnte über den Dateinamen, die Dateigröße und den Dateitypen auf den Inhalt der Daten schließen. Um das Problem zu beheben, könnten die gerade aufgeführten Werte zusätzlich über die gewählte hybride Verschlüsselung gesichert werden. Hier wird die zusätzliche Verschlüsselung allerdings nicht weiter betrachtet, da sie auf dem schon implementierten Verschlüsselungsverfahren beruht und somit keine neuen Erkenntnisse für diese Arbeit liefert.
Ein weiteres Problem ist, dass die Integrität der Chunknummern nicht geschützt ist. Ein Angreifer kann demnach die Reihenfolge der Chunks beliebig vertauschen, ohne das etwas auffällt. Die zusätzliche Berechnung eines *MACs* über die Chunknummern würde das Problem lösen oder die Chunknummern werden über die hier implementierte hybride Verschlüsselung gesichert. So hätte man die Schutzziele *Vertraulichkeit* und *Integrität* beide erfüllt.

Der Entschlüsselungsprozess bekommt als Eingabe aus der Komponente zum Dateiaustausch eine heruntergeladene binäre Datenstruktur. Diese Datenstruktur muss zunächst entpackt werden, um die einzelnen abgespeicherten Werte zu extrahieren. Der private Schlüssel wird jetzt für die Entschlüsselung des Dateischlüssels genommen. Ist der Dateischlüssel einmal entschlüsselt, so können als nächstes die Daten eines Chunks dechiffriert werden und die Integrität wird überprüft. Die entschlüsselten und authentifizierten Blöcke werden jetzt anhand der Chunknummer in die richtige Reihenfolge gebracht, gespeichert und können dem Admin als Klartextdatei zur Verfügung gestellt werden.

3.4.2.3 Komponente zum Dateiaustausch

Diese Komponente kümmert sich um den Datenaustausch zwischen Client und Speicher-Server.

Zum einen gibt es eine Operation zum Hochladen von Daten. Diese Operation nimmt das binäre Paket aus der vorherigen Komponente, benennt es nach einem bestimmten Schema und reicht es an den Speicher weiter. Das Benennungsschema enthält den Dateinamen und die Chunknummer. Den gleichen Dateinamen für zwei unterschiedliche Dateien zu wählen ist grundsätzlich möglich. Auf dem Speicher allerdings wird zwischen den beiden Dateien unterschieden, indem der Dateiname indexiert wird.

Zum anderen sollte auch die Möglichkeit bestehen, Daten wieder herunterladen zu können. Es werden anhand des Dateinamens die passenden Daten herausgesucht. Nach dem Herunterladen können die Daten zurück an die Komponente für kryptographische Verfahren gereicht werden.

3.4.3 Webserver

Der Webserver innerhalb des Systems ist für die Bereitstellung der Webanwendung zuständig. Auf ihm werden HTML, JavaScript und CSS-Dateien gespeichert.

Es wird davon ausgegangen, dass der Webserver gut gesichert in einem geschützten Bereich liegt. Außerdem müssen Webserver und Cloud-Speicher streng voneinander getrennt sein. Ein Angreifer darf nicht in der Lage sein, den öffentlichen Schlüssel gegen einen selbst erzeugten Schlüssel auszutauschen. Alle Clients würden folglich ihre Dateischlüssel mit dem vom Angreifer erzeugten Schlüsselpaar verschlüsseln und der Angreifer ist in der Lage, diese wieder zu entschlüsseln.

3.4.4 Plattformunabhängigkeit des Client

Bei der Entwicklung des Systems wurde auf die Plattformunabhängigkeit der Client-Anwendung geachtet. Die fertige Webanwendung läuft somit unter den aktuellen Standardbrowsern auf Desktop-Endgeräten, wie auch auf mobilen Endgeräten.
Zur Erreichung der Plattformunabhängigkeit wurde bei jeder verwendeten Technologie, die Unterstützung durch das Endgerät geprüft. Nur die Technologien, die von den oben beschriebenen Browsern unterstützt werden, wurden für die Entwicklung genommen. Eine hilfreiche Webseite zur Überprüfung ist `http://caniuse.com`.

Die graphische Benutzeroberfläche sollte sich mobilen, wie auch Desktop-Browsern anpassen können.

4 Implementierung

Dieses Kapitel beschäftigt sich mit der Umsetzung der Softwarelösung. Dabei wird ein relevanter Programmcode referenziert und seine Bedeutung für das System wird erläutert. Zunächst werden die bei der Implementierung wichtigsten verwendeten Technologien vorgestellt. Desweiteren wird die Wahl des in der Arbeit verwendeten *JavaScript*-Frameworks begründet. Danach gibt es einen Überblick über die Architektur des Systems, die Einrichtung des Cloud-Servers und die Implementierung der entscheidenden Komponenten aus der Client-Bibliothek wird besprochen. Auch werden Probleme, die während der Implementierung gelöst werden mussten, diskutiert.

4.1 Benutzte Technologien

In dieser Sektion wird eine Auswahl der in der Implementierung verwendeten Technologien vorgestellt.

JavaScript wurde zur Programmierung des Frameworks genommen. Die graphische Benutzeroberfläche wurde mit Hilfe von *HTML*, *CSS* erstellt.

4.1.1 Droopy

Bei *Droopy* handelt es sich um einen einfachen HTTP-Server, der mit den Grundfunktionen zum Hochladen (POST-Methoden) und Herunterladen (GET-Methoden) von Dateien ausgestattet ist. Er ist in der Programmiersprache *Python* geschrieben. Nachdem das Python-Skript heruntergeladen wurde, kann es direkt über die Konsole ausgeführt werden. Voraussetzung ist natürlich, das Python installiert ist. Als Parameter können dem Konsolenaufruf der Port übergeben werden, über den der Droopy-Server läuft. [11]

In dieser Arbeit wurde der Droopy-Server als Speicher-Server verwendet, da die HTTP-Schnittstelle als hinreichend generisch gewertet werden kann. Er läuft auf dem Webserver unter dem Port 9000. Die Anfrage an den Droopy-Server muss über einen HTTPS-Proxy

geleitet werden, denn der Webserver nutzt HTTPS und würde die Anfrage blockieren. Mit der Schnittstelle *XMLHttpRequest* wurden einfache Funktionen zum Hoch- und Herunterladen von Dateien geschrieben.

4.1.2 XMLHttpRequest

XMLHttpRequest ist eine Programmierschnittstelle für *JavaScript*. Sie wird von allen gängigen Browsern implementiert, ausgenommen von *Opera Mini*. Mit Hilfe dieser Schnittstelle können Daten zwischen einem Server und einem Client über das HTTP-Protokoll ausgetauscht werden. Sämtliche HTTP-Anfragemethoden, darunter *GET*, *POST* und *PUT*, können über ein *XMLHttpRequest*-Objekt genutzt werden.
Operationen fürs Senden und Empfangen von Daten können asynchron oder synchron aufgerufen werden. Asynchrone Aufrufe bieten den Vorteil, dass nicht gewartet wird bis die Anfrage ausgeführt wurde, sondern in der Zwischenzeit auch andere Zeilen im Code bearbeitet werden können. [18]

4.1.3 Web Cryptography API

Die *Web Cryptography API* ist eine W3C-standardisierte Kryptoimplementierung des Browsers, die über *JavaScript* genutzt werden kann. Sie beinhaltet grundlegende kryptographische Verfahren zur Verschlüsselung und zur Signaturberechnung, sowie zur Integritätsberechnung. Zudem wird sie von fast allen aktuellen Browsern zum größten Teil unterstützt. *Opera Mini* ist der einzige Browser, der die *Web Cryptography API* gar nicht unterstützt.
Gegenüber anderen Implementierungen hat die *Web Cryptography API* den Vorteil, dass sie nativ ist und relativ schnell über den Browser ausgeführt werden kann.

4.1.4 W3.css

Das frei benutzbare *Cascading Style Sheet*-Framework von der *W3-School* bietet alles, was für die graphische Entwicklung einer modernen Webapplikation angemessen ist. Es werden Layouts und Designs für alle HTML-Elemente angeboten.
Für die Darstellung der HTML-Seite verwendet *W3.css* das responsive Webdesign. Somit kann es sich jeder Browserfenstergröße, unabhängig davon ob ein Desktop-Endgerät oder ein mobiles Endgerät genutzt wird, exakt anpassen. [16]

4.1.5 Forge

Das in dieser Arbeit gewählte JavaScript-Framework ist *forge*. *Forge* ist eine Implementierung von TLS in JavaScript. In dieser Arbeit werden insbesondere die kryptographischen Verfahren des Frameworks genutzt. Es beinhaltet standardisierte Operationen zum Ver-und Entschlüsseln, zur Integritätsprüfung und zum Signieren von Daten. [9]
Es wurde sich für dieses Framework entschieden, da alle in der Arbeit verwendeten kryptographischen Verfahren aus der *Web Cryptography API* auch in *forge* implementiert sind.

4.2 Wahl der Kryptoimplementierungen

In dieser Arbeit wurden die *Web Cryptography API* und das JavaScript-Framework *forge* als Kryptoimplementierungen verwendet. Es wurde darauf geachtet, dass beide Implementierungen über die gleichen kryptographischen Funktionen verfügen, um bei der Ver-und Entschlüsselung und der Integritätsprüfung die gleiche Sicherheit gewährleisten zu können.

4.3 Plattformunterstützung

Die Verschlüsselung ist die Hauptfunktionalität dieses Systems und kann von jedem Nutzer durchgeführt werden. Zu den unterstützten Browsern zählen: *Safari*, *Firefox*, *Chrome*, *iOS Safari* und *Android Chrome*.

Die Entschlüsselung wurde einstufig über die *Web Cryptography API* implementiert. Sie kann nur vom Admin durchgeführt werden. Es soll in dieser Arbeit lediglich gezeigt werden, dass eine Entschlüsselung der Dateien auch wieder möglich ist. Wichtig hierbei war, dass der gewählte Browser für die Entschlüsselung die *Web Cryptography API* unterstützt. Daher wurde sich in dieser Arbeit für den Desktop-Browser *Firefox* entschieden.

Falls das asymmetrische Schlüsselpaar aus Sicherheitsgründen irgendwann einmal erneuert werden muss, so sollte die Generierung ebenfalls im Desktop-Browser *Firefox* durch den Admin erfolgen.

Für die graphische Benutzeroberfläche wurde ein responsives Design gewählt. Ein responsives Design hat den Vorteil, dass die Darstellung sich dem Browser des Endgerätes anpasst.

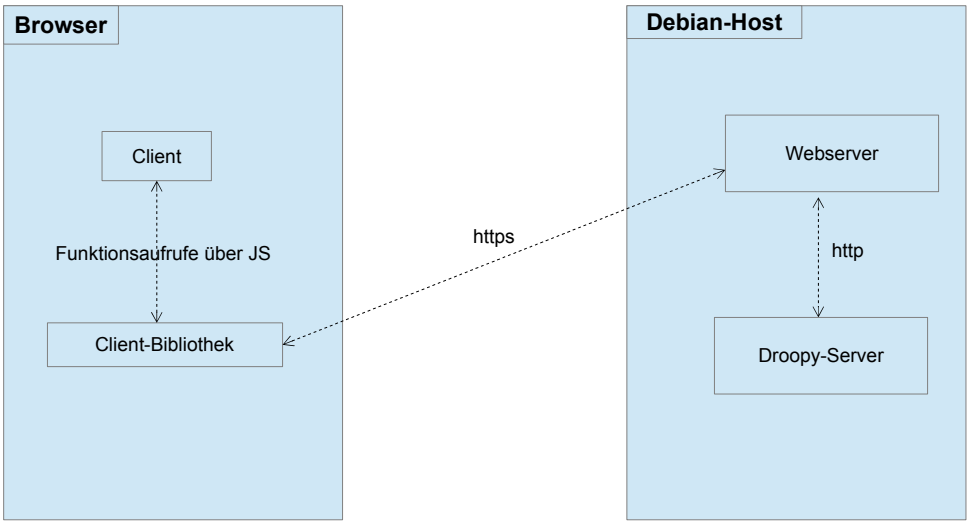


Abbildung 4.1: Die konkrete Umsetzung des Systems.

4.4 Architektur des Systems

Die Architektur des Sytems wird in Abbildung 4.1 dargestellt. Das System ist grob in zwei Bereiche einzuteilen.
Auf der linken Seite der Abbildung ist der *Browser*-Bereich zu sehen. Der *Client* ist die graphische Benutzeroberfläche der Webanwendung. Hierüber kann ein Nutzer mit dem System interagieren. Die eigentliche Funktion des Systems ist in der *Client-Bibliothek* implementiert. Über JavaScript-Funktionsaufrufe werden bestimmte Aktionen oder Ereignisse abgearbeitet, die innerhalb des *Clients* stattfinden.
Auf der rechten Seite ist der entfernte *Debian-Host* dargestellt. Der *Debian-Host* wurde extra zum Zwecke dieser Arbeit vom Lehrstuhl *Verteilte Systeme* bereitgestellt. Er kümmert sich um die Auslieferung der Webanwendung über einen vorinstallierten Apache-*Webserver*. Als Speicher-Server wurde in dieser Arbeit der *Droopy-Server* (Abschnitt: 4.1.1) gewählt, der auf dem *Debian-Host* gehostet wird. HTTP-Anfragen an den *Droopy-Server* werden über den *Webserver* umgeleitet. Diese Umleitung wurde eingerichtet, um speziell Problemen mit der *Same Origin Policy* aus dem Weg zu gehen.

4.5 Navigation innerhalb der Client-GUI

Die Client-GUI ist aus 4 Teilen aufgebaut, nämlich die Startseite, die Verschlüsselungsseite, die Entschlüsselungsseite und die Webseite zur Generierung des Schlüsselpaares. Die letzten zwei genannten Webseiten sind nur für den Admin gedacht.

Es gibt eine Startseite, die der Nutzer als erstes ansteuern kann. Die Startseite ist in Abbildung 4.2a dargestellt. Auf der Startseite wird die Funktion der Webanwendung kurz beschrieben und der Nutzer hat die Möglichkeit sich grob zu orientieren. Über eine Navigationsbar im oberen Bereich kommt der Nutzer zur Verschlüsselungsseite und zur Entschlüsselungsseite.

Auf der Verschlüsselungsseite kann der Nutzer eine zu verschlüsselnde Datei aus dem lokalen Verzeichnis auswählen und ihr einen Namen geben. Die Auswahl des Namens sollte aus Sicherheitsgründen nicht auf den Inhalt der Datei schließen lassen. Weiterhin kann ein Speicherdienst angegeben werden, um die verschlüsselten Daten später hochzuladen. Über einen Button auf dem unteren Teil der Seite kann die Hauptfunktion ausgelöst werden. Die ausgewählte Datei wird in Blöcke eingelesen, verschlüsselt und dann hochgeladen. Eine Fortschrittsleiste soll der Überwachung des Prozesses dienen. Die Webseite zur Verschlüsselung ist in Abbildung 4.2b zu sehen.

Die Entschlüsselungsseite in Abbildung 4.2c ist nur für den Admin gedacht. Sie ist ähnlich wie die Verschlüsselungsseite aufgebaut, mit der Logik einer Entschlüsselung. Über ein Textfeld kann die Datei mit Namen angegeben werden, die heruntergeladen und entschlüsselt werden soll. Zusätzlich gibt es ein Feld, mit dem der private Schlüssel ausgewählt werden kann, der für den Entschlüsselungsvorgang unabdingbar ist.

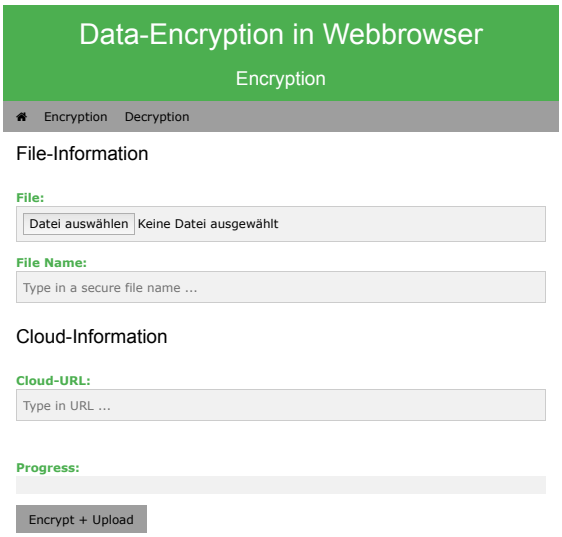
Als letztes ist noch die Webseite zur Schlüsselpaargenerierung anzuführen (Abbildung 4.2d). Diese ist nicht Teil der eigentlichen Webanwendung und kann somit auch nicht über die Navigationsleiste angesteuert werden. Der Admin kann diese Seite zum Initialisieren des Systems und zum Aufrischen des asymmetrischen Schlüsselpaares verwenden.

Von jeder Seite, außer der Webseite zur Generierung des Schlüsselpaares, kommt man wieder zur Startseite zurück.

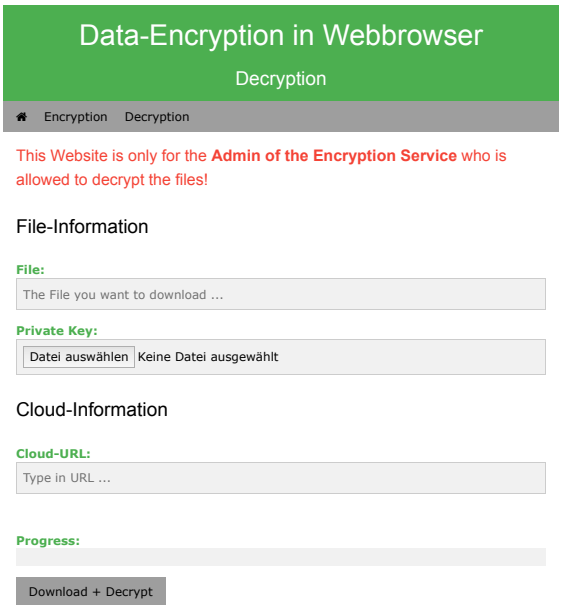
4.6 Umsetzung der Client-Bibliothek

Dieser Abschnitt beschäftigt sich mit der Implementierung der Client-Bibliothek, die bereits in der Konzeption des Frameworks angesprochen wurde. Innerhalb der einzelnen Komponenten werden interessante Stellen des Programmcodes referenziert und erläutert.

(a) Die Startseite des Systems.



(b) Die Webseite zur Verschlüsselung.



(c) Die Webseite zur Entschlüsselung.

(d) Die Webseite zur Schlüsselgenerierung.

Abbildung 4.2: Die Webseiten des Software-Prototyps.

4.6.1 Komponente zur Dateiauswahl

4.6.1.1 Datei-Chunking

Die Verschlüsselung großer Dateien im Browser führt zu Problemen, da diese meist nicht komplett in den Arbeitsspeicher eingelesen werden können und den Browser um einiges verlangsamen. Daher wurde sich in dieser Arbeit für das Datei-Chunking entschieden, das sich darum kümmert eine Datei in Blöcke einzulesen. Durch diese Vorgehensweise können Blöcke im Programmfluss viel leichter und effizienter verarbeitet werden.

Hat der Nutzer eine Datei ausgewählt und klickt auf die Schaltfläche *Encrypt+Upload*, so wird die *main*-Funktion aufgerufen. Die *main*-Funktion kümmert sich um das Einlesen der Datei in Chunks und anschließend um die Weiterverarbeitung. In Listing 4.1 ist ein Ausschnitt der *main*-Funktion dargestellt, der zeigen soll, wie das Datei-Chunking in dieser Arbeit umgesetzt wurde.
In einer Art Schleife liest die Funktion Chunk für Chunk der Datei ein. Die Chunkgröße ist dabei variabel und kann ja nach gewünschter Granularität festgelegt werden. Hier wurde eine Chunkgröße von 1 MB genommen. Bei der Schleife handelt es sich um eine spezielle Schleife, da hier asynchrone Aufrufe vom *FileReader*-Objekt behandelt werden müssen. Aus diesem Grund kann auch keine normale for-Schleife verwendet werden, da diese bereits in der nächsten Iteration wäre, aber der asynchrone Aufruf des *FileReader*-Objektes noch gar nicht abgearbeitet wurde.
Der initiale Schleifendurchlauf startet mit dem Aufruf der Funktion *chunkReaderBlock()* in Zeile 38 und jede weitere Iteration beginnt über den Aufruf in Zeile 60. In jeder Schleifeniteration wird nun ein Chunk eingelesen und durch die Funktion *chunkProcessing()* weiterverarbeitet. Die *chunkProcessing()*-Funktion kümmert sich um die Verschlüsselung und das Hochladen jeden Chunks.
Wurde die Datei komplett eingelesen oder es gibt einen Fehler, so terminiert die Schleife und kehrt mit einem *return* zum ihrem Aufrufer zurück.

```
1 // Hier findet die Chunk-Verarbeitung statt in einer asynchronen
     Schleife (Hauptfunktion des Skriptes)
2 // getriggert durch Button "Encrypt+Upload"
3 function main(){
4   // inspiriert von: alediaferia
5   // link: http://stackoverflow.com/questions/14438187/javascript-
     filereader-parsing-long-file-in-chunks
6
7   var fileSize = selectedFile.size;
8   var chunkSize = 1024*1024; // 1 MB als Chunkgroesse
9   var offset = 0;
```

```
10  var chunkReaderBlock = null;
11  var chunkCounter = 0;
12
13  var cloudURL = document.getElementById("InputURL").value;
14  var fileName = document.getElementById("InputFileName").value;
15
16  var handlingReadEvent = function(evt){
17
18    if(evt.target.error == null){
19      offset += chunkSize;
20      chunkProcessing(evt.target.result);
21    }else {
22      console.log("Fehler beim Lesen: " + evt.target.error);
23      return;
24    }
25
26  }
27
28  chunkReaderBlock = function(offset, length, selectedFile){
29
30    var r = new FileReader();
31    var blob = selectedFile.slice(offset, (length + offset));
32    r.onload = handlingReadEvent;
33    r.readAsArrayBuffer(blob);
34
35  }
36
37  // initialer Aufruf der asynchronen Schleife
38  chunkReaderBlock(offset, chunkSize, selectedFile);
39
40
41  // Implementierung der Chunk-Verarbeitung (Verschluesselung,
     Hochladen)
42  function chunkProcessing(fileBuffer){
43
44    // Hier wird jeder eingelesene Chunk verschluesselt und
     hochgeladen
45
46  }
47
48  // Hochladen eines Chunks mittels XMLHttpRequest
49  function sendBlob(blob){
50
51    xhr_send.open('POST', cloudURL, true);
```

```
52
53      xhr_send.onload = function(){
54        if(offset >= fileSize){
55          // Hier muss nichts gemacht werden!
56          console.log("Datei wurde erfolgreich verschluesselt und
    hochgeladen!");
57        }else{
58          chunkCounter++;
59          // Aufruf der naechsten Iteration der asynchronen Schleife
60          chunkReaderBlock(offset, chunkSize, selectedFile);
61        }
62      }
63
64
65      var formData = new FormData();
66
67      formData.append("upfile", blob, fileName+" "+chunkCounter);
68
69      xhr_send.send(formData);
70
71
72    }
73
74  }
```

Listing 4.1: Ausschnitt der *main*-Funktion zeigt Datei-Chunking.

4.6.2 Komponente für kryptographische Verfahren

4.6.2.1 Zweistufige Kryptoimplementierung

Die Verwendung von nur einer Kryptoimplementierung für die Verschlüsselung kann auf verschiedenen Plattformen zu Kompatibilitätsproblemen führen. Beispielsweise kann eine fehlende Unterstützung der kryptographischen Funktionen eine Nicht-Verschlüsselung zur Folge haben. Die in dieser Arbeit benutzte *Web Cryptography API* wird nicht von allen Browsern vollständig unterstützt.
Daher wurde sich hier für eine zweistufige Implementierung der hybriden Verschlüsselung entschieden. Die zweistufige Implementierung bietet den großen Vorteil, dass eine Verschlüsselung unabhängig von der Plattform garantiert werden kann. Werden die

verwendeten kryptographischen Funktionen von der *Web Cryptography API* nicht unterstützt, so gibt es ein Fallback auf das von außen eingebundene JavaScript-Framework *forge*. Mögliche Performanz-Verluste hierdurch werden akzeptiert.

Die Überprüfung der Unterstützung der *Web Cryptography*-Schnittstelle für die Verschlüsselung wird in Listing 4.2 exemplarisch gezeigt. Für die symmetrische Schlüsselgenerierung und anschließende asymmetrische Verschlüsselung wurde genau der gleiche Entscheidungsalgorithmus genommen.
Es ist unklar mit welchem Browser das Skript ausgeführt wird. Hinzu kommt noch, dass die *Web Cryptography*-Schnittstelle mancher Standardbrowser durch ein Anbieter-Präfix individualisiert ist. Zum Beispiel verwendet der Browser Safari das Präfix *webkitSubtle*. Daher müssen alle Schnittstellen gesondert überprüft werden.
Der Algorithmus ist in der Lage, eine Nicht-Unterstützung der *Web Cryptography API* zu erkennen. Die Schnittstelle hat in diesem Fall den Wert *undefined* oder es entsteht bei der Überprüfung ein Fehler, der vom äußeren *try-catch*-Block gefangen und bearbeitet wird.
Desweiteren wird auch erkannt, wenn die verwendeten kryptographischen Funktionen nicht verfügbar sind, weil während der Verschlüsselung ein Fehler auftritt. Auch der hier auftretende Fehler wird vom umschließenden *try-catch*-Block behandelt.
Ist die *Web Cryptography API* also undefiniert oder es kommt zu einem der oben beschriebenen Fehlerfälle, so wird eine Variable namens *counter* hochgezählt. Nur bei unzureichender Unterstützung der *Web Cryptography API* hat die *counter*-Variable am Ende den Wert 3 und löst einen Fallback auf die Implementierung des JavaScript-Frameworks *forge* aus.

Zusätzlich wurde hier das Problem einer Mehrfach-Verschlüsselung gelöst. Möglicherweise wird in Zukunft *window.crypto.subtle* als einheitliche Schnittstelle angeboten und die alten individualisierten Schnittstellen können weiterhin genutzt werden. Um eine doppelte Verschlüsselung durch diese Änderung auszuschließen, wird eine boolschen Variable *encStat* eingeführt. Diese Variable hält fest, dass eine Verschlüsselung bereits schon stattgefunden hat. Der nachfolgende Code wird somit übersprungen.

```
1 // Implementierung der Chunk-Verarbeitung (Verschluesselung,
    Hochladen)
2 function chunkProcessing(fileBuffer){
3
4   var counter = 0;
5   var decided = false;
6
7
8   if(!decided){
9     try{
```

```
10        // Ueberpruefung der Web Crypto-Schnittstelle fuer Safari
11        if(window.crypto.webkitSubtle){
12
13          encryptWC(window.crypto.webkitSubtle, fileBuffer,
      chunkCounter)
14          .then(function (result){
15            sendBlob(result);
16          });
17
18          decided = true;
19
20        }else if (window.crypto.webkitSubtle == undefined){
21          counter++;
22        }
23      }catch(e){
24        console.log(e.message);
25        counter++;
26      }
27    }
28
29    if(!decided){
30      try{
31        // Ueberpruefung der Web Crypto-Schnittstelle fuer Internet
      Explorer
32        if(window.msCrypto.subtle){
33
34          encryptWC(window.msCrypto.subtle, fileBuffer, chunkCounter
      )
35          .then(function (result){
36            sendBlob(result);
37          });
38
39          decided = true;
40        }else if(window.msCrypto.subtle == undefined){
41          counter++;
42        }
43      }catch(e){
44        console.log(e.message);
45        counter++;
46      }
47    }
48
49    if(!decided){
50      try{
```

```
51        // Ueberpruefung der Web Crypto-Schnittstelle fuer andere
      Browser (Firefox, Chrome, usw.)
52        if(window.crypto.subtle){
53
54          encryptWC(window.crypto.subtle, fileBuffer, chunkCounter)
55          .then(function (result){
56            sendBlob(result);
57          });
58
59          decided = true;
60
61        }else if(window.crypto.subtle == undefined){
62          counter++;
63        }
64      }catch(e){
65        console.log(e.message);
66        counter++;
67      }
68    }
69
70
71    // Fallback auf JavaScript-Framework forge
72    if(counter == 3){
73
74      var result = encryptJF(fileBuffer, chunkCounter);
75      sendBlob(result);
76
77    }
78  }
```

Listing 4.2: Der Entscheidungsalgorithmus zur Wahl einer geeigneten kryptographischen Implementierung.

4.6.2.2 Hybride Verschlüsselung durch Funktionsabschluss

Die hybride Verschlüsselung wurde in dieser Arbeit durch einen Funktionsabschluss implementiert.
Die Schlüsselgenerierung und die Verschlüsselung mussten getrennt werden, da nur ein Schlüssel pro Datei generiert und asymmetrisch verschlüsselt werden sollte. Anschließend sollte mit diesem Schlüssel alle Chunks einer Datei verschlüsselt werden.
Ein zu lösendes Problem dabei war die Asynchronität der *Web Cryptography API*.

Zunächst musste der Schlüssel asynchron generiert werden. Eine anschließende Verschlüsselung kann erst durchgeführt werden, wenn der Schlüssel verfügbar ist. Die Lösung war, das Schlüsselversprechen zu externalisiern, sodass Schlüsselgenerierung und Verschlüsselung gemeinsam darauf Zugriff haben. Durch das *Schlüsselverprechen*-Objekt konnte jetzt nachdem die Generierung des Schlüssel abgeschlossen war, die Verschlüsselung eingeleitet werden.
Ein weiterer Vorteil eines Funktionsabschlusses ist, dass sicherheitskritische Variablen, wie der symmetrische Schlüssel oder das Schlüsselversprechen, auf einer lokalen Ebene deklariert werden können und somit in einem sicheren Scope liegen. Nur die Funktionen, die durch die Closure erstellt und zurückgegeben werden, haben Zugriff auf die lokal definierten Variablen. Denkbare Angriffe, wie das *Cross Site Scripting* können wir so aus dem Weg gehen.

Als Beispiel eines Funktionsabschlusses wird in Listing 4.3 die hybride Verschlüsselung über die *Web Cryptography API* implementiert. Die Verschlüsslung über das JavaScript-Framework *forge* ist analog aufgebaut, mit dem einzigen Unterschied, dass es keine asynchronen Aufrufe gibt.
Die Closure *prepareEncryptWebCrypto()* gibt durch ihren Aufruf die Funktionsdefinitionen für die Funktionen *generateAndEncryptKeyWebCrypto()* und *encryptWebCrypto()* zurück. Die Funktion *generateAndEncryptKeyWebCrypto()* generiert als erstes den Dateischlüssel und verschlüsselt ihn anschließend asymmetrisch mit dem öffentlichen Schlüssel. Danach kann die Funktion *encryptWebCrypto()* aufgerufen werden. Sie wird auf Grundlage des Schlüsselversprechen *keyPromise* aufgerufen. Denn erst wenn der Schlüssel *keyObject* generiert ist, kann mit diesem auch verschlüsselt werden. Die Funktion *encryptWebCrypto()* kümmert sich um die symmetrische Verschlüsselung der Chunks und das anschließende Packen eines Blobs, in dem alle wichtigen Informationen zur Verschlüsselung festgehalten werden.
Beide Funktionen arbeiten über eine Kette von asynchronen Aufrufen. Diese Kette entsteht durch das *then*-Konstrukt, das durch die Implementierung der *Web Cryptography API* mitgeliefert wird. In der Funktion *generateAndEncryptKeyWebCrypto()* wird beispielsweise zuerst der Dateischlüssel durch den Aufruf *crypto.generateKey()* generiert. Danach wird der Schlüssel durch *then(saveSessionKey)* abgespeichert und zurückgegeben. Das Besondere hierbei ist, dass der Rückgabewert an den nächsten Aufruf der Kette (*then(exportKeyObject)*) weitergereicht wird und dort wiederverwendet werden kann. Außerdem hängen die asynchronen Aufrufe durch diese Kette voneinander ab. Ein Aufruf kann nur getätigt werden, wenn der Aufruf der Funktion davor ein Ergebnis geliefert hat.

```
1 // Implementierung der Schluesselgenerierung und Verschluesselung
    mit Web Cryptography API
```

```
2  function prepareEncryptWebCrypto(){
3    var keyObject;
4    var keyPromise;
5    var exportedKey;
6    var encryptedKey;
7
8    // Generierung des Dateischluessels mit anschliessender
       Verschuesselung
9    function generateAndEncryptKeyWebCrypto(crypto){
10
11     keyPromise = crypto.generateKey({name: "AES-GCM", length:
       128}, true, ["encrypt","decrypt"]).
12
13     then(saveSessionKey).
14     then(exportKeyObject).
15     then(importPublicKey).
16     then(encryptKeyObject);
17
18
19     function saveSessionKey(key){
20       keyObject = key;
21       return key;
22     }
23
24     function exportKeyObject(keyObject){
25       crypto.exportKey("raw", keyObject).
26       then(function (exportedSessionKey){
27         exportedKey = exportedSessionKey;
28       });
29     }
30
31     function importPublicKey(){
32       return crypto.importKey("spki", publicKey, {name: "RSA-OAEP"
       , hash: {name: "SHA-256"}}, false, ["encrypt"]).
33       then(function (importedPublicKey){
34         return importedPublicKey;
35       });
36     }
37
38     function encryptKeyObject(importedPublicKey){
39       crypto.encrypt({name: "RSA-OAEP",}, importedPublicKey,
       exportedKey).
40       then(function (encryptedSessionKey){
41         encryptedKey = encryptedSessionKey;
```

```
42
43          console.log("Schluesselgenerierung mit Web Crypto API ist
    fertig!");
44        });
45      }
46    }
47
48    // Verschluesselung der Chunks
49    function encryptWebCrypto(crypto, chunk, chunkNum){
50
51      var tag;
52      var vector = window.crypto.getRandomValues(new Uint8Array(16))
    ;
53      var encryptedData; // mit Auth Tag
54      var normalizedCiphertext; // ohne Auth Tag
55
56
57      return keyPromise.
58
59      then(encryptData).
60      then(extractTag).
61      then(normalizeCiphertext).
62      then(packageResults);
63
64
65      function encryptData(){
66
67        return crypto.encrypt({name: "AES-GCM", iv: vector,
    tagLength: 128}, keyObject, chunk).
68        then(function (ciphertext){
69          encryptedData = new Uint8Array(ciphertext);
70          return new Uint8Array(ciphertext);
71        });
72      }
73
74      function extractTag(ciphertext){
75        tag = new Uint8Array(16);
76        var offset = (ciphertext.length-16);
77        for(var i=ciphertext.length-16; i<ciphertext.length;i++){
78          tag[i-offset] = ciphertext[i];
79        }
80      }
81
82      function normalizeCiphertext(){
```

```
83      normalizedCiphertext = new Uint8Array(encryptedData.
    byteLength-16);
84
85      for(var i=0; i<normalizedCiphertext.length;i++){
86        normalizedCiphertext[i] = encryptedData[i];
87      }
88    }
89
90
91    function packageResults(){
92      var dataName = selectedFile.name;
93      var chunkNumber = String(chunkNum);
94      var dataSize = String(selectedFile.size);
95      var fileType = selectedFile.type;
96      var encryptedDataLength = String(normalizedCiphertext.
    byteLength);
97      var lengthKeyStr = String(encryptedKey.byteLength);
98
99      var lengthKeyStrSize = new Uint8Array([stringToByteArray(
    lengthKeyStr).length]);
100     var lengthDataName = new Uint8Array([stringToByteArray(
    dataName).length]);
101     var lengthChunkNum = new Uint8Array([stringToByteArray(
    chunkNumber).length]);
102     var encryptedDataStrLength = new Uint8Array([
    stringToByteArray(encryptedDataLength).length]);
103     var lengthTag = new Uint8Array([tag.byteLength]);
104     var dataSizeStrLength = new Uint8Array([stringToByteArray(
    dataSize).length]);
105     var fileTypeLength = new Uint8Array([stringToByteArray(
    fileType).length]);
106
107
108     return new Blob(
109       [
110         lengthDataName,       //Always a 1 byte unsigned integer
111         dataName,
112         dataSizeStrLength,
113         dataSize,
114         fileTypeLength,
115         fileType,
116         lengthChunkNum,       //Always a 1 byte unsigned integer
117         chunkNumber,
118         lengthKeyStrSize,
```

```
119             lengthKeyStr,            //Always a 1 byte usigned
    integer
120             encryptedKey,        //"length" bytes long
121             vector,            // 16 byte
122             lengthTag,           // 1 byte
123             tag,
124             encryptedDataStrLength,
125             encryptedDataLength,       // 4 byte
126             normalizedCiphertext         //Ciphertext
127
128
129         ],
130
131           {type:"application/octet-stream"}
132         );
133     }
134   }
135
136   // Rueckgabe der Funktionsdefinition durch Closure
     prepareEncryptWebCrypto()
137   return [generateAndEncryptKeyWebCrypto, encryptWebCrypto];
138 }
```

Listing 4.3: Funktionsabschluss der hybriden Verschlüsselung durch *Web Cryptography API*.

4.6.2.3 Packen der entschlüsselten Chunks

Natürlich muss garantiert werden können, dass die entschlüsselten Chunks wieder in der richtigen Reihenfolge zusammengesetzt werden, um die Klartextdatei zu erhalten, die man zu Anfang verschlüsselt hat. Daher werden die Chunks in aufsteigender Reihenfolge, angefangen beim *Index 0*, heruntergeladen und entschlüsselt. Danach wird für jeden Chunk die Funktion *packPlaintextData()* aufgerufen. Diese Funktion kümmert sich um das Zusammensetzen der einzelnen Chunks zu einer Datei. Sie ist in Listing 4.4 exemplarisch dargestellt.

Zu Anfang der Funktion werden die Variablen *plaintext* und *offset* initialisiert. Die Variable *plaintext* ist ein Uint8Array und stellt ein Puffer für die Klartextdatei dar, die aus allen Chunks zusammengesetzt wird. Diese Initialisierung findet immer nur für das erste Chunk jeder Datei statt. Danach werden in einer for-Schleife die Bytes jedes Chunks in das *plaintext*-Array einsortiert. Die Variable *offset* wird selbstverständlich für jeden

Aufruf der Funktion *packPlaintextData()* erneut angepasst. Zusätzlich wird sie aber auch durch ein return zurückgegeben. Dieser Rückgabewert kann im Folgenden dazu verwendet, um zu erkennen, ob das *plaintext*-Array voll ist. Wenn die Klartextdatei vollständig gepackt ist, so wird sie dem Admin zum Download bereitgestellt und kann danach eingesehen werden.

```
1  function packPlaintextData(dataSize, chunk, chunkNr){
2
3    // Initialisierung
4    if(parseInt(chunkNr) == 0){
5      plaintext = new Uint8Array(dataSize);
6      offset = 0;
7    }
8
9    // Einsortieren der Chunks
10   for(var i = offset; i < (offset+chunk.length); i++){
11     plaintext[i] = chunk[i-offset];
12   }
13
14   // Neuinitialisierung
15   offset += chunk.length;
16
17   return offset;
18
19 }
```

Listing 4.4: Packen der entschlüsselten Chunks durch die Funktion *packPlaintextdata()*.

4.6.3 Komponente zum Dateiaustausch

4.6.3.1 Automatisches Suchen und Herunterladen von Chunks

Für das Herunterladen der Chunks wurde ein Automatismus gewählt. Eine große verschlüsselte Datei mit 2 GB bei einer Chunkgröße von 1 MB kann aus bis zu 2000 Chunks bestehen. Es wäre für den Admin sehr mühselig und umständlich, jeden Chunk manuell herunterladen zu müssen. Zudem kann das automatische Herunterladen in einen durchgehenden Programmfluss eingebettet werden. Die Datei wird heruntergeladen, entschlüsselt, gepackt und der Durchlauf fängt wieder von vorne an.

Ein Algorithmus, der das automatische Suchen und Herunterladen von Chunks umsetzt ist in Listing 4.5 abgebildet.

Die einzige vom Admin gemachte Angabe ist der Dateiname, der zu herunterladenen Datei. Den Rest erledigt der Algorithmus durch den Aufruf der *main()*-Funktion über den Button *Download+Decrypt*.
Der Algorithmus hat einen ähnlichen Aufbau wie die Funktion für das Datei-Chunking (Listing 4.1). Mit Hilfe einer asynchronen Schleife kann jeder Chunk einzeln heruntergeladen und anschließend bearbeitet werden. Hier kann keine synchrone Schleife, wie zum Beispiel eine *for*- oder *while*-Schleife gewählt werden, da das Einlesen und das Entschlüsseln der eingelesenen Blobs asynchron abläuft. Die synchronen Schleifen würden nicht bis zur Abarbeitung eines asynchronen Ereignisses warten und es mit Beginn der nächsten Schleifeniteration überspringen.
Der initiale Schleifendurchlauf wird durch den Aufruf der Funktion *receiveBlob()* in Zeile 51 gestartet. Nun läuft die Schleife für alle Chunks, die zu einer Datei gefunden werden können. Durch die Variable *chunkCounter* wird der aktuelle Index eines Chunks gespeichert. Der Index fängt bei 0 an und wird aufsteigend bis zum letzten Chunk erhöht. Diese Sortierung spiegelt auch gleichzeitig die richtige Reihenfolge wieder, in der die Chunks zu einer Datei zusammmengesetzt werden müssen.
Nachdem alle Chunks einer Datei heruntergeladen sind, beendet die asynchrone Schleife sich durch den Download der Klartextdatei.

```
1  // In einer asynchronen Schleife werden alle Chunks einer Datei
       heruntergeladen und weiterverarbeitet (Hauptfunktion)
2  // getriggert durch den Button "Download+Decrypt"
3  function main(){
4
5    var receiveBlob = null;
6    var receivedBlob;
7    var blobName = document.getElementById("fileToDownload").value;
8    var cloudURL = document.getElementById("InputURL").value;
9    var chunkCounter = 0;
10
11
12   var processReceivedBlob = function(evt){
13
14     // Einlesen und Entpacken der Blobs
15
16     // Entschluesseln der Chunks
17     decrypt(ciphertextWithTag, iv, encryptedSessionKey)
18     .then(function (chunk){
19       var plaintextChunk = new Uint8Array(chunk);
20
21       var offset = packPlaintextData(parseInt(dataSize, 10),
     plaintextChunk, chunkNr);
```

```
22
23        // Die Schleife wird beendet durch den Download der
      Klartextdatei.
24        if(offset == parseInt(dataSize, 10)){
25
26          downloadPlaintextBlob(createBlob(fileType), dataName,
      fileType);
27          alert('Decryption is finished. You can now view your file!
      ');
28
29        }else{
30          chunkCounter++;
31          // Hier beginnt der Aufruf der naechsten
      Schleifeniteration!
32          receiveBlob();
33        }
34      })
35      .catch(function (e){
36        console.log(e.message);
37      });
38
39    }
40
41
42    receiveBlob = function(){
43      xhr_receive.open('GET', 'https://stud01.vs.uni-due.de:443/
      droopy/'+blobName+' '+chunkCounter, true);
44      xhr_receive.responseType = 'blob';
45      xhr_receive.send(null);
46      xhr_receive.onload = processReceivedBlob;
47    }
48
49    // initialer Durchlauf der Schleife beginnt hier!
50    receiveBlob();
51
52  }
```

Listing 4.5: Asynchrone Schleife für das automatische Suchen und Herunterladen von Chunks.

5 Evaluation

In diesem Kapitel findet die kritische Auswertung des in dieser Arbeit entwickelten Systems statt. Es gibt zwei verschiedene Analysen.
Die erste Analyse wird sich mit der Performanz des Systems beschäftigen. Dazu werden die Laufzeiten der kryptographischen Verfahren, die auf den verschiedenen Plattformen erreicht werden, gemessen und evaluiert.
Da es sich beim Prototypen um ein sicherheitskritisches System handelt, wird auch die Sicherheit in einer zweiten Analyse beurteilt.

5.1 Laufzeitanalyse des Systems

In diesem Abschnitt geht es um die Laufzeitanalyse des entwickelten Systems. Zunächst wird der Aufbau des Testkontextes geklärt. Danach werden die Tests anhand bestimmter Kriterien durchgeführt und am Ende ausgewertet.

5.1.1 Aufbau des Tests

Die gemessenen Werte wurden in unterschiedliche Kategorien unterteilt. Es wurden hauptsächlich Messungen für die Hauptkomponenten des Systems durchgeführt, die aus Abschnitt 3.4.2 bekannt sind. Somit ergeben sich die Kategorien: *Datei-Chunking*, *Verschlüsselung*, *Hochladen*. Zusätzlich wird noch die Gesamtlaufzeit des Prozesses der Verschlüsselungsseite gemessen und betrachtet. Die Entschlüsselung in dieser Arbeit soll nur exemplarisch zeigen, dass man die Daten auch wieder zurückerhält, die man verschlüsselt hat. Daher wurde eine Auswertung für die Entschlüsselung nicht durchgeführt.
Der Test fand auf unterschiedlichen Plattformen statt. So konnten die unterschiedlichen Kryptoimplementierungen, nämlich die *Web Cryptography API* und das JavaScript-Framework *forge*, getestet werden.
Unter den Plattformen befinden sich:

MacBook Pro:

- 2,53 GHz Intel Core 2 Duo Prozessor
- 4 GB 1067 MHz DDR3-RAM
- OS X El Capitan 10.11.6 (64-bit)
- Firefox 52.0
- Chrome 56.0.2924.87
- Safari 10.0.3

iPad Pro:

- A9X Chip (64-bit) mit integriertem M9 Coprozessor
- 2 GB LPDDR4-RAM
- iOS 10.2.1
- Safari 10.0

Samsung Galaxy Note 4:

- Qualcomm Snapdragon 805 mit 2,7 GHz, Quad-Core
- 3 GB RAM
- Android Chrome 56.0.2924.87

```
1  function encryptWebCrypto(crypto, chunk, chunkNum){
2    // Timestamp fuer Start Verschluesselung
3    if(chunkCounter == 0){
4      encryption_timestamp_start = Date.now();
5    }
6
7    var tag;
8    var vector = window.crypto.getRandomValues(new Uint8Array(16));
9    var encryptedData; // mit Auth Tag
10   var normalizedCiphertext; // ohne Auth Tag
11
12
13   return keyPromise.
14
15   then(encryptData).
16   then(extractTag).
17   then(normalizeCiphertext).
18   then(packageResults);
```

```
19
20
21    function encryptData(){
22      return crypto.encrypt({name: "AES-GCM", iv: vector, tagLength:
        128}, keyObject, chunk).
23      then(function (ciphertext){
24        // Timestamp fuer Ende Verschluesselung
25        if(chunkCounter == 0){
26          encryption_timestamp_end = Date.now();
27          encryption_time_index = listTimeValues(
      encryption_time_index, encryption_time, (
      encryption_timestamp_end - encryption_timestamp_start));
28        }
29
30        encryptedData = new Uint8Array(ciphertext);
31        return new Uint8Array(ciphertext);
32      });
33    }
34
35    // restliche Verarbeitung + Packen der hochzuladenen Datei
36  }
```

Listing 5.1: Messung der Verschlüsselung mit *Web Cryptography API*.

Der Test wurde in der Skriptsprache *JavaScript* geschrieben. Ein Ausschnitt, bei dem die Verschlüsselung eines Chunks mit der *Web Cryptography API* gemessen wird, ist in Listing 5.1 dargestellt. Die Messung fängt in Codezeile 4 an und endet genau nach dem asynchronen Aufruf der Verschlüsselungsmethode in Zeile 26. Eine if-Bedingung sorgt dafür, dass immer nur die Zeit des ersten Chunks einer Messrunde genommen wird. Die Initialisierung des Vektors und anderer Variablen wurde bewusst drin gelassen, da diese zur Verschlüsselung dazu gehören. Der Test für das JavaScript-Framework *forge* ist analog aufgebaut. Zur Durchführung des Tests wurde eine 10 MB große Eingabedatei und eine Chunkgröße von 1 MB zugrunde gelegt. Er wurde für alle Kategorien insgesamt 50-mal durchgeführt. Der Durchschnittswert der Messungen wurde durch den Median repräsentiert, da dieser unempfindlich gegenüber Ausreißern ist. Um zusätzlich ein stabiles Maß für die Streuung zu erhalten, wurde der Median der absoluten Abweichungen genommen. Der Median und das Streuungsmaß für die getesteten Plattformen wurden in einem Säulendiagramm dargestellt. Die Streuung ist als Fehlerbalken verzeichnet.

5.1.2 Durchführung des Tests

In dieser Sektion findet die Durchführung der Tests nach dem oben beschriebenen Aufbau statt.

5.1.2.1 Generierung des Dateischlüssels mit Verschlüsselung

In der Abbildung 5.1 sind die Ergebnisse der Messungen der Schlüsselgenerierung mit anschließender Verschlüsselung dargestellt. Es wurde für die Verschlüsselung mit *AES-GCM* ein 128 Bit großer Dateischlüssel erzeugt und dann mit dem asymmetrischen Verfahren *RSA-OAEP* gesichert. Dieser Vorgang findet pro Eingabedatei genau einmal statt, sodass wir für jede neue Datei ein neuen Schlüssel haben. Die Generierung und Verschlüsselung passiert bereits schon nach dem Auswählen der Datei und nicht im eigentlich Hauptprozess der Anwendung. Aufgrund dieser Eigenschaft wurde diese Messung in der Gesamtlaufzeit nicht berücksichtigt.

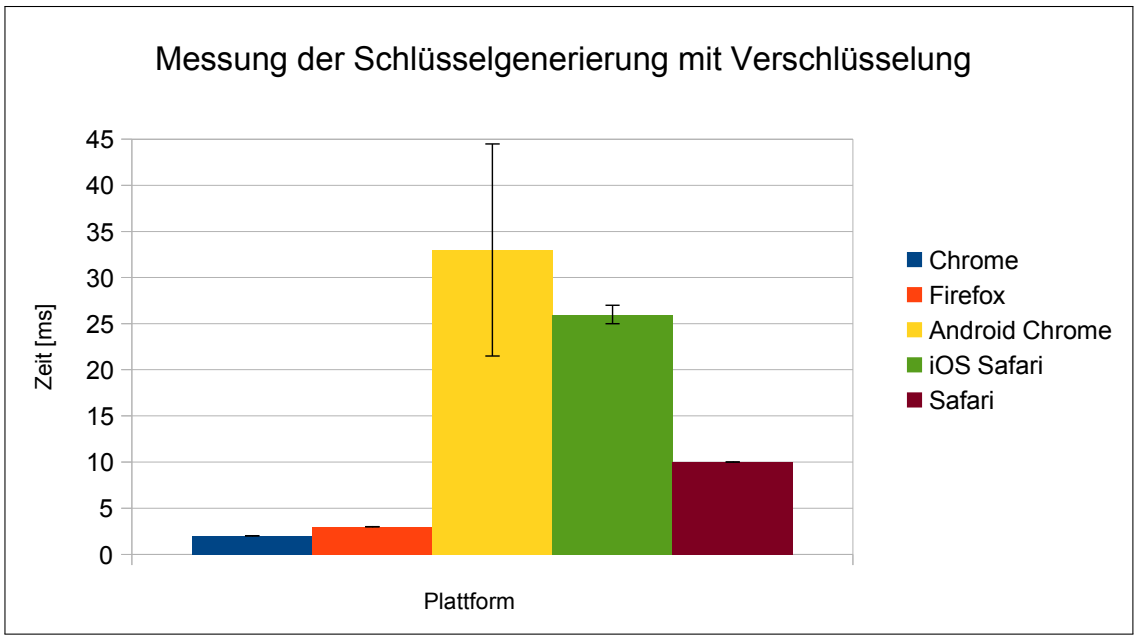


Abbildung 5.1: Messung der Schlüsselgenerierung mit Verschlüsselung über RSA-2048 im OAEP Modus.

Die Browser *Android Chrome* und *iOS Safari* sind erkennbar langsamer als *Chrome*, *Firefox* und *Safari*.

Android Chrome ist der langsamste Browser mit 33 ms und *Chrome* ist der schnellste Browser mit einer durchschnittlichen Laufzeit von 2 ms.

Chrome ist ungefähr um den Faktor 1,5 schneller als *Firefox*, um den Faktor 16,5 schneller als *Android Chrome*, um den Faktor 13 schneller als *iOS Safari* und um den Faktor 5 schneller als der Browser *Safari*.

Die Plattform *Firefox* ist bei der dargestellten Messung um den Faktor 11 schneller als *Android Chrome*, um den Faktor 8,6 schneller als *iOS Safari* und in etwa um die Größenordnung 3,3 schneller als *Safari*.

Der Browser *Android Chrome* ist ungefähr um den Faktor 1,27 langsamer als *iOS Safari* und um den Faktor 3,3 langsamer als *Safari*.

iOS Safari ist um den Faktor 2,6 langsamer als *Safari*.

5.1.2.2 Datei-Chunking

In Abbildung 5.2 sind die durchschnittlichen Maßzahlen für die Laufzeit des Datei-Chunking auf den unterschiedlichen Plattformen zu sehen.

Android Chrome braucht deutlich länger für das Chunking als die restlichen Plattformen.

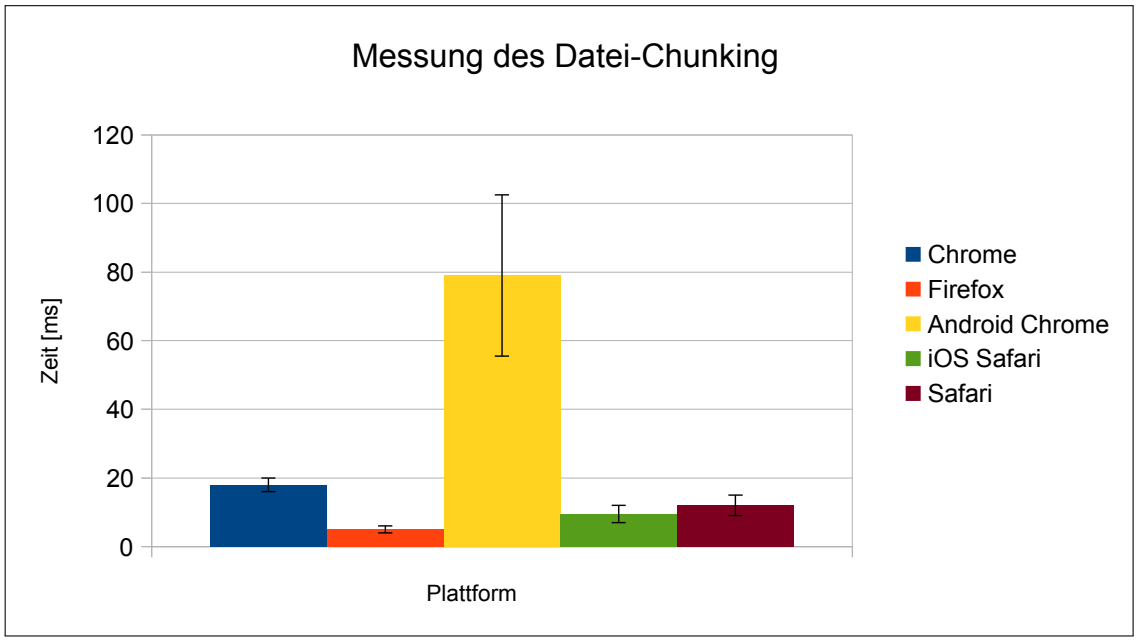


Abbildung 5.2: Messung des Datei-Chunking auf den verschiedenen Plattformen.

Chrome ist in etwa um den Faktor 3,6 langsamer als *Firefox* im Einlesen der Chunks. Weiterhin ist *Chrome* um den Faktor 4,4 schneller als der Browser *Android Chrome*, um den Faktor 1,89 lansamer als *iOS Safari* und um den Faktor 1,5 langsamer als *Safari*.

Firefox ist im Datei-Chunking der schnellste Browser mit 5 ms. Er ist um den Faktor 15,8 schneller als *Android Chrome*, um den Faktor 1,9 schneller als *iOS Safari* und um den Faktor 2,4 schneller als *Safari*.

Android Chrome ist der langsamste Browser mit einem Durchschnittswert von 79 ms. Er ist um den Faktor 8,32 langsamer als *iOS Safari* und um den Faktor 6,58 langsamer als der Browser *Safari*.

iOS Safari ist im Vergleich mit *Safari* um den Faktor 1,26 schneller im Chunking.

5.1.2.3 Verschlüsselung

Die Messung der Verschlüsselung mit AES-128 im GCM Modus wird in Abbildung 5.3 dargestellt. Auf der y-Achse des Säulendiagramm ist die Zeit in Millisekunden abgetragen und auf der x-Achse sind die gemessenen Plattformen angegeben. Die Durchschnittswerte wurden pro Chunk gemessen, da in dieser Arbeit ein Datei-Chunking implementiert wurde.

Es fällt sofort auf, dass die Browser *Chrome*, *Firefox*, *Android Chrome* deutlich schneller sind, als die Browser *iOS Safari* und *Safari*.

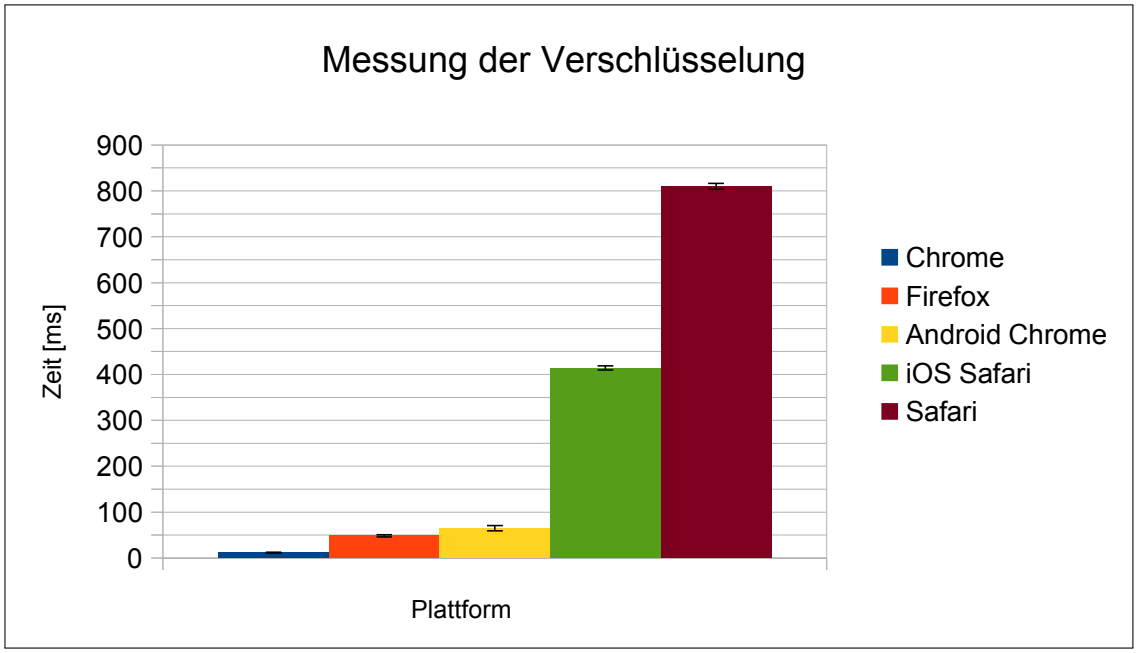


Abbildung 5.3: Messung der Verschlüsselung mit AES-128 im GCM Modus.

Der Browser *Safari* ist ungefähr um den Faktor 1,96 langsamer als der Browser *iOS Safari*, um den Faktor 12,46 langsamer als *Android Chrome*, um die Größe 16,70 langsamer als der Browser *Firefox* und um den Faktor 67,5 langsamer als *Chrome*.

iOS Safari ist in etwa 6,37 langsamer als *Android Chrome*, um den Faktor 8,54 langsamer als die Plattform *Firefox* und um den Faktor 34,5 langsamer als *Chrome*.

Die Plattform *Android Chrome* ist ungefähr um den Faktor 1,34 langsamer als *Firefox* und um die Größe 5,42 langsamer als der Browser *Chrome*.

Abschließend ist *Firefox* um den Faktor 4,04 langsamer als der Browser *Chrome*.

5.1.2.4 Hochladen

In der Abbildung 5.4 ist die Messung des Uploads dargestellt. Hier ist zu beachten, dass die y-Achse des Säulendiagramm erst bei 3700 ms anfängt.

Die Messung wurde mit einer Uploadgeschwindigkeit von bis zu 3 MBit/s durchgeführt. Die Durchschnittswerte werden auch hier auf einen Chunk bezogen.

Die Laufzeiten der Uploads auf den verschiedenen Plattformen unterscheiden sich nur wenig voneinander. Der Upload mit *Android Chrome* braucht am längsten.

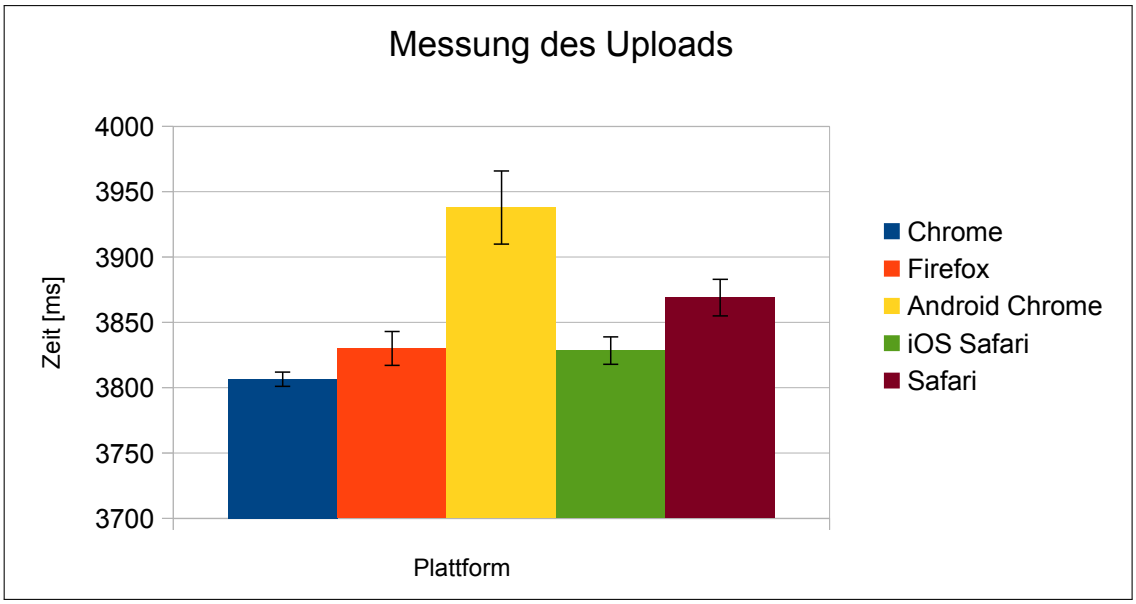


Abbildung 5.4: Messung des Uploads zum Speicher.

Der Browser *Chrome* ist ungefähr um den Faktor 1,01 schneller als *Firefox*, um den Faktor 1,03 schneller als *Android Chrome*, um den Faktor 1,006 schneller als *iOS Safari* und ungefähr um die Größe 1,02 schneller als der Browser *Safari*.

Firefox ist ungefähr um den Faktor 1,03 schneller als *Android Chrome*, ist ungefähr gleich so schnell wie *iOS Safari*, um den Faktor 1,01 schneller als der Browser *Safari*.

Der Browser *Android Chrome* ist in etwa um den Faktor 1,03 lansamer als der Browser *iOS Safari* und um den Faktor 1,02 langsamer als *Safari*.

Abschließend lässt sich feststellen, dass der Browser *iOS Safari* ungefähr um den Faktor 1,01 schneller ist als *Safari* mit einem durchschnittlichen Wert von 3869 ms.

5.1.2.5 Gesamtlaufzeit

Die durchschnittliche Gesamtlaufzeit (dargestellt in Abbildung 5.5) gibt an, wie lange der Gesamtprozess der Verschlüsselungsseite braucht, bis er 10 MB mit einer Chunkgröße von 1 MB komplett verarbeitet hat. Die gemessenen Einzelkomponenten für *Datei-Chunking*, *Verschlüsselung* und *Upload* sind darin enthalten. Die Schlüsselgenerierung des Dateischlüssels mit anschließender Verschlüsselung wird in der Anwendung bereits schon nach dem Auswählen der Datei gemacht. Somit wurde diese Komponente einzeln getestet und ist nicht in der Gesamtlaufzeit enthalten.

Der größte Unterschied ist zwischen dem Browser *Chrome* und *Safari*. Mit 8401 ms ist *Chrome* schneller als *Safari*.

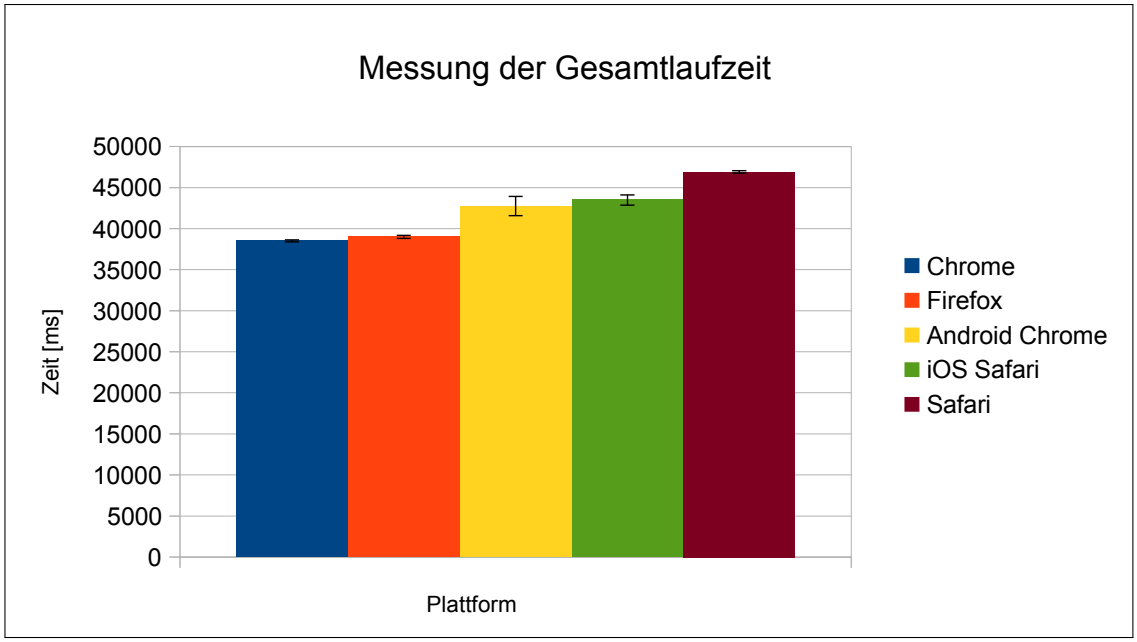


Abbildung 5.5: Messung der Gesamtlaufzeit unter den verschiedenen Plattformen.

Der Browser *Chrome* ist in diesem Test ungefähr um den Faktor 1,01 schneller als *Firefox*, um den Faktor 1,11 schneller als die Plattform *Android Chrome*, um den Faktor 1,13 schneller als *iOS Safari* und um den Faktor 1,22 schneller als *Safari*.

Firefox ist in etwa um den Faktor 1,096 schneller als *Android Chrome*, um den Faktor 1,12 schneller als *iOS Safari* und um den Faktor 1,20 schneller als der Browser *Safari*.

Die Plattform *Android Chrome* ist in der Messung 5.5 um den Faktor 1,02 schneller *iOS Safari*. Hier gibt es kaum einen Unterschied. Etwas deutlicher ist der Unterschied zwischen *Android Chrome* und *Safari*. Der Browser *Andriod Chrome* ist mit einem Faktor von 1,1 schneller als *Safari*.

Abschließend ist die Gesamtlaufzeit der Verschlüsselungsseite für *iOS Safari* um den Faktor 1,08 verschieden von der Gesamtlaufzeit des Browsers *Safari*. *iOS Safari* ist also ein wenig schneller als *Safari*.

5.1.3 Auswertung des Tests

Bei allen Messungen muss das Scheduling des Betriebsystems und eine Latenz in den Netzwerkzeiten berücksichtigt werden. Die relativ hohe Streuung des Browsers *Android Chrome* in jeder Messung könnte auf natürliche Schwankungen dieser Art zurückzuführen sein.

Die Durchführung der Kryptographieoperationen mit der *Web Cryptography API* unter den Browsern *Chrome*, *Firefox*, *Android Chrome* sind deutlich schneller, als die Durchführung mit *forge* unter den Plattformen *iOS Safari* und *Safari*. Dieses Verhältnis zeigt, dass die zweistufige Kryptoimplementierung in dieser Arbeit als erfolgreich zu werten ist.
Das JavaScript-Framework *forge* ist mit einer Verschlüsselung von ungefähr 800 ms schon sehr langsam. Hier könnte das Framework in Zukunft durch ein Anderes ausgetauscht werden, um auch bei einem Fallback einen gute Performanz liefern zu können. Im Rahmen dieser Arbeit ist der große Geschwindigkeitsverlust aber zu akzeptieren, da das Framework *forge* die Alternativimplementierung und somit die zweite Wahl ist.
Allerdings fällt auf, dass die Schlüsselgenerierung mit Verschlüsselung (Abschnitt 5.1.2.1) in *Android Chrome* sichtbar länger dauert, als beispielsweise bei *Firefox* und *Chrome*. Dabei arbeiten beide Browser auf Basis der *Web Cryptography API*. Daher muss die Ursache für diesen großen Unterschied in der Browserarchitektur oder in der Betriebssytemarchitektur des Android-Mobilgerätes liegen.

Das in der Arbeit implementierte Datei-Chunking ist ebenfalls ein großer Erfolg für die Performanz des Systems. Der langsamste Browser *Android Chrome* braucht ungefähr 80 ms, um einen Chunk der Datei einzulesen. Würde man jetzt den Anwendungsfall betrachten, dass ein Nutzer eine große Datei mit mehreren Gigabyte verschlüsseln will, so ist der Durchsatz durch das Chunking enorm höher, als wenn die Datei als Ganzes in

den Browser eingelesen wird. Der Browser würde viel langsamer werden, möglicherweise würde die Anwendung einfrieren, da der Arbeitsspeicher des Browsers überläuft.

Würde man die Chunkgranularität des Systems ändern, so ließe sich die Performanz mit den vorhandenen Werten ungefähr abschätzen. Es ist zu erwarten, dass die Performanz annähernd gleich bleibt. Denn je kleiner die Chunkgröße, desto mehr Chunks müssen verarbeitet werden, aber desto kleiner könnten Einzellaufzeiten wie zum Beispiel für den Upload sein. Je größer die Chunks, desto weniger Chunks einer Datei entstehen, die aber zu höheren Laufzeiten führen würden. Daher wurde dieser Fall in der Evaluation nicht gesondert betrachtet.

Alles in allem, läuft die Verschlüsselung des Prototypen auf einigen Plattformen und liefert eine relativ gute Performanz. Das wiederum begründet die Tauglichkeit des Systems. Im besten Fall hat der Nutzer für die Verschlüsselung eine Gesamtlaufzeit von ungefähr 38,5 s, wenn von einer 10 MB Eingabedatei und einer Chunkgröße von 1 MB ausgegangen wird.
Der größte Anteil in der Gesamtzeit macht der Upload aus und die Verschlüsselung im Vergleich einen kleinen Anteil. Die Internetgeschwindigkeit kann vom System nicht beeinflusst werden. Sie ist bei jedem Nutzer anders. Aber die Struktur der hochzuladenen Dateien kann modifiziert werden, indem bestimmte Werte eingespart werden. So könnte sicherlich ein höherer Durchsatz erzielt werden.

5.2 Sicherheitsanalyse des Systems

In diesem Abschnitt findet die Sicherheitsanalyse des Systems statt. Die informationstechnischen Sicherheitsziele *Vertraulichkeit*, *Verfügbarkeit* und *Integrität* werden als Maßstab genommen. Es wird geschaut, wie gut die einzelnen Ziele vom Prototyp umgesetzt und berücksichtigt wurden. Mögliche Schwachstellen werden in Zusammenhang mit ihren Angriffsvektoren dargestellt und kritisch diskutiert.

5.2.1 Vertraulichkeit

Der erste untersuchte Abschnitt in der Sicherheitsanalyse befasst sich mit dem Thema *Vertraulichkeit*. *Vertraulichkeit* bedeutet den Dateninhalt vor der Einsicht durch fremde Personen zu schützen.

Der hier entwickelte Software-Prototyp verwendet eine hybride Verschlüsselung. Eine hybride Verschlüsselung hat den Vorteil, dass neben der Verschlüsselung der Daten auch

noch der Dateischlüssel chiffriert wird, ohne das ein geheimer Schlüssel sicher ausgetauscht werden muss. Somit kann auch der Dateischlüssel sicher auf dem Speicher abgelegt werden. Zudem wurden sichere Kryptoverfahren genommen, die nach heutiger Kenntnis nicht zu brechen sind.

Weiterhin wurde die Verschlüsselung durch einen Funktionsabschluss implementiert. Sicherheitskritische Variablen, wie beispielsweise der symmetrische Schlüssel, können so auf funktionaler Ebene definiert werden und können nicht über das *Cross Site Scripting* ausgespäht werden.

Aber die beste Verschlüsselung würde nichts bringen, wenn man nicht eine Instanz hätte, der man vertrauen kann. In diesem System ist die vertrauenswürdige Instanz der *Admin*. Er verwaltet den privaten Schlüssel. Ein Vertrauensbruch könnte dazu führen, dass der Admin den privaten Schlüssel an unbefugte Personen weiterreicht. Einen potenziellen Vertrauensbruch zu verhindern, ist unmöglich und würde die Grenzen dieses Systems überschreiten.

Betrachten wir die Verschlüsselung einmal genauer, so fällt auf, dass nur die Nutzdaten und der Schlüssel berücksichtigt werden. Metainformationen wie Dateiname, Dateigröße und der Dateityp bleiben unverschlüsselt. Angenommen ein Angreifer schafft sich Zugang zum Speicher und kennt die Struktur der hochgeladenen Daten. Über den Dateinamen, die Dateigröße und den Dateityp kann er jetzt auf den ungefähren Inhalt einer Datei schließen. Um diesem Angriff zu entgehen, könnte man die genannten Werte durch das schon implementierte Verfahren verschlüsseln. Das implementierte Verfahren würde über die Verschlüsselung hinaus sogar noch zusätzlich für die Integrität der Daten sorgen.

Sobald die verschlüsselten Daten an den Server versendet werden, verlassen sie den geschützten Bereich des Browsers. Hier übernimmt das Internetprotokoll *HTTPS* die Verschlüsselung der Kommunikation zwischen Client und Server und sorgt für einen sicheren Datenaustausch. Ein *Man-in-the-Middle*-Angriff könnte mit der verschlüsselten Kommunikation nichts anfangen. Hier sind die Metainformationen wie Dateiname, Dateigröße und Dateityp noch geschützt. Aber bei der permanenten Speicherung auf dem Server, wie oben bereits angesprochen, nicht mehr.

5.2.2 Integrität

Der nächste Abschnitt der Sicherheitsanalyse beschäftigt sich mit der *Integrität* des Systems. Die *Integrität* kümmert sich um die Korrektheit der Daten und um die Korrektheit der Funktionsweise von Systemen. [4]

Die Integrität der verschlüsselten Daten, also der Nutzdaten und des Dateischlüssels, ist gesichert. Über das Verfahren *AES-GCM* werden die Nutzdaten und über *RSA-OAEP* mit *SHA-256* wird der Dateischlüssel fälschungssicher gemacht. In beiden Fällen also kann eine Modifikation der Daten erkannt werden.

Allerdings wird hier die Integrität der Chunknummern in der hochgeladenen Datei nicht berücksichtigt. Ein Angreifer könnte die Reihenfolge der Chunks ändern und eine Entschlüsselung unmöglich machen. Die Lösung ist ein zusätzlichen *MAC* über die Chunknummern zu berechnen. Noch praktischer ist aber die Sicherung über das implementierte Verschlüsselungsverfahren. Somit hat man nicht nur die Integrität, sondern auch eine Verschlüsselung.

Die Funktionsweise des Gesamtsystems hat ebenfalls einen Schwachpunkt aufzuweisen. Die Webanwendung wird von einem Webserver ausgeliefert. Ein Angreifer könnte das System verletzen, indem der Code der Webanwendung modifiziert oder gar ganz ausgetauscht wird. Nutzer würden auf falsche Webseiten geleitet und die Angreifer könnten sich die Daten sofort abgreifen. Sie können ein eigenes Schlüsselpaar erzeugen und es gegen das vorhandene austauschen. Jegliche Dateien, die durch das System verschlüsselt werden, können nun von den Angreifern entschlüsselt werden. Durch das *Secure-Hosting* kann eine Prüfungsinstanz eingerichtet werden, zum Beispiel ein weiterer Server. Dieser prüft dann bei Auslieferung der Webseite über einen Hashwert, ob es sich um die richtige Seite handelt.

5.2.3 Verfügbarkeit

Der letzte angesprochene Abschnitt in der Analyse ist die *Verfügbarkeit* des Systems. Sie steht zwar nicht im Fokus dieser Arbeit, wird aber dennoch besprochen. Die *Verfügbarkeit* bedeutet, dass die Funktionen eines IT-Systems sowie die Daten ständig vorhanden sein müssen und von den Anwendern wie vorgesehen genutzt werden können. [4]

Ein Vorteil der entwickelten Webanwendung ist, dass sie statisch ist. Sie spricht nur mit dem Speicher-Server, um die Daten zum Speichern herauszuschicken. Die Verschlüsselung und somit die Hauptfunktionalität findet ausschließlich im Browser statt. Sie hängt nicht von der Bereitschaft anderer Server ab und wird auch nicht durch sie beeinflusst. Wenn man davon ausgeht, dass die Integrität der Funktionalität der Webanwendung, wie oben angesprochen, gesichert ist, dann ist eine Verschlüsselung zu jedem Zeitpunkt durchführbar und kann durch einen Angriff nicht verhindert werden.

Nun kann es passieren, dass durch einen Angriff Daten auf dem Speicher gelöscht werden. Vor dem Löschen kann das System nicht schützen. Aber die gespeicherten Daten könnten

auf andere Server gespiegelt werden. Hat man also einen Verlust von Daten, so können diese durch eine Wiederherstellung gerettet werden.

Auch könnte der private Schlüssel abhanden kommen. Dem Verlust müsste kein Angriff zu Grunde liegen. Es würde reichen, wenn der *Admin* den Schlüssel verlegt. Alle damit verschlüsselten Dateien wären unbrauchbar. Es besteht die Möglichkeit den privaten Schlüssel zu vervielfältigen und durch ein *Backup* wieder herzustellen. Allerdings wird die Angriffsfläche auf das System größer, denn je mehr private Schlüssel existieren, desto mehr müssen vor Angriffen geschützt werden.

Wie oben schon angemerkt heißt *Verfügbarkeit* auch, dass die Funktionen eines IT-Systems dauerhaft genutzt werden können. Stürzt der Webserver wegen eines *DOS*-Angriffes ab, so kann die Anwendung nicht mehr gehostet werden. Um dieses Problem zu beheben, könnte ein *Storage-Backend* für den Code der Webanwendung eingerichtet werden. Dieser wird angesteuert, wenn der Webserver nicht mehr verfügbar ist.

6 Fazit und Ausblick

Zu Anfang der Arbeit wurden Anforderungen (Abschnitt 3.1) aufgestellt, die für die Entwicklung des Prototypen erfüllt werden mussten. Darunter war die Umsetzung einer hybriden Ver- und Entschlüsselung im Browser, das Datei-Chunking, das Hoch-und Herunterladen von Dateien zu einem Speicher mit generischer Schnittstelle, die Integrität der Daten und die Plattformunabhängigkeit des Sytems.

Die hybride Verschlüsselung wurde mit einem zweistufigem Konzept implementiert. Dadurch konnte eine Plattformunabhängigkeit der Kryptoimplementierung erreicht werden. Wird die Standardimplementierung nicht unterstützt, so weicht das System auf eine von außen eingebundene Implementierung aus und kann somit eine Verschlüsselung trotzdem garantieren.

Aufbauend auf der Verschlüsselung wurde zusätzlich die Integrität der Daten gesichert, um Modifikationen ausschließen zu können. Dazu wurden bewusst Verfahren genommen, die beides unterstützen. Die verwendeten Verfahren sind *AES-GCM* und *RSA-OAEP*.

Das eine Entschlüsselung der Daten auch wieder möglich ist, sollte in dieser Arbeit nur exemplarisch gezeigt werden.

Desweiteren wurde im Vorfeld darauf geachtet, dass der Durchsatz der Verschlüsselung im Browser optimal ist. Daher wurde ein Datei-Chunking umgesetzt. Große Dateien werden in Blöcke zerlegt und können besser in den Arbeitsspeicher des Browsers geladen werden, um danach weiter verarbeitet zu werden.

Die Speicher-Schnittstelle zum Speichern der verschlüsselten Dateien wurde generisch gewählt, um im späteren Einsatz des Systems ein möglichst breites Spektrum an Cloud-Speichern zu unterstützen. Hierzu wurde eine *HTTP*-Schnittstelle genommen, die mit wenig Aufwand von vielen Cloud-Anbietern zu Verfügung gestellt werden kann. Zum Hoch-und Herunterladen wurden die *HTTP*-Methoden *GET* und *POST* verwendet.

Die Laufzeit- und Sicherheitsanalyse in Kapitel 5 zeigen, wie gut im Einzelnen die Anforderungen des Prototypen umgesetzt wurden.

Der Prototyp erfüllt zwar alle Anforderungen, die zu Beginn der Arbeit gestellt wurden, aber er ist auch eine stark vereinfachte Lösung, die in Zukunft noch weiter ausgebaut werden kann, um am Ende dem alltäglichen Gebrauch Stand zu halten. Die Ausbaufähigkeit kann unter die Kategorien *Sicherheit*, *Performanz* und *Benutzerfreundlichkeit* zusammengefasst werden. Hier werden ein paar Anregungen zu Weiterentwicklungen gemacht.

Die Sicherheit des Systems kann an einigen Stellen weiter verbessert werden (siehe Abschnitt 5.2).
Sie hängt selbstverständlich auch stark von der Verwendung der kryptographischen Verfahren ab. Möglicherweise lassen sich schnellere und modernere Verfahren implementieren, die noch mehr Sicherheit gewährleisten können.

Die zweite ausbaufähige Kategorie ist die Performanz des Systems. Die hochgeladenen Daten können durch zwei verschiedene Blobformate dargestellt werden. Im ersten Blobformat werden alle wichtigen Informationen zur Datei gesammelt. Die Informationen wie den Dateinamen, den Dateityp und die Dateigröße können dann in den folgenden Blobs eingespart werden. Dadurch kann der Durchsatz des Hochladens erhöht werden und es gibt keinen unnötigen Mehraufwand.
Desweiteren könnte die Ausführung der Webapplikation über Worker-Threads laufen. Durch Worker-Threads kann eine Nebenläufigkeit in *JavaScript* erzeugt werden. Neben dem Hochladen der Daten könnte das Einlesen des nächsten Chunks stattfinden. Dadurch würde sich viel Zeit sparen lassen.

Die letzte hier angeführte Kategorie ist die Benutzerfreundlichkeit des Systems. Um die Benutzerfreundlichkeit des Sytems zu steigern, kann grundsätzlich die GUI überarbeitet werden. Die gemachten Eingaben des Nutzers könnten auf Korrektheit überprüft und die Benutzeroberfläche könnte ansprechender und barrierefreier gestaltet werden.
Aber auch die Entschlüsselung könnte nutzerfreundlicher gestaltet werden. Die hier entwickelte Entschlüsselung funktioniert über die Implementierung der *Web Cryptography API* und darf nur vom Admin durchgeführt werden. Von Interesse wäre die Entschlüsselung auch für den Standardnutzer zu ermöglichen und ein externes Tool für die Entschlüsselung zu entwicklen, sodass auch größere Dateien mit mehreren Gigabytes ohne Probleme entschlüsselt werden können.
Zum Schluss wäre noch die Praktikabilität der Umsetzung einer Schlüsselverteilung anzumerken, bei der mehrere Nutzer sich eine Datei teilen können.

Literaturverzeichnis

[1] Arnd Böken: *Zugriff auf Zuruf?* https://www.heise.de/ix/artikel/Zugriff-auf-Zuruf-1394430.html. – [Online; Stand 23. November 2016]

[2] Ben Rodenhäuser: *Damit sie nicht aus allen Wolken fallen.* http://www.manager-magazin.de/unternehmen/it/a-582750-2.html. Version: 2008. – [Online; Stand 23. November 2016]

[3] BSI: *Cloud Computing Grundlagen.* https://www.bsi.bund.de/DE/Themen/DigitaleGesellschaft/CloudComputing/Grundlagen/Grundlagen_node.html. – [Online; Stand 23. November 2016]

[4] BSI: *IT-Grundschutz.* https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzKataloge/Inhalt/Glossar/glossar_node.html. Version: 2013. – [Online; Stand 15. März 2017]

[5] BSI: *IT-Grundschutz: Rollen.* https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzKataloge/Inhalt/Rollendefinitionen/rollendefinitionen_node.html. Version: 2013. – [Online; Stand 22. Dezember 2016]

[6] BSI: *Kryptographische Vefahren: Empfehlungen und Schlüssellängen.* https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf?__blob=publicationFile. Version: 2016. – [Online; Stand 1. Dezember 2016]

[7] Cryptomator: *Cryptomator.* https://cryptomator.org. – [Online; Stand 26. Oktober 2016]

[8] David A. McGrew, John Viega: *The Galois/ Counter Mode of Operation (GCM).* http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf. – [Online; Stand 13. Februar 2017]

[9] Digital Bazaar: *Forge.* https://github.com/digitalbazaar/forge#contact. – [Online; Stand 20. Februar 2017]

[10] Moriarty, et al.: *RSA Cryptography Specifications Version 2.2*. https://tools.ietf.org/html/rfc8017. Version: 2016. – [Online; Stand 12. Januar 2017]

[11] Pierre Stackp: *Droopy- easy file receiving*. http://stackp.online.fr/?p=28. – [Online; Stand 20. Februar 2017]

[12] Prof. Dr. Martin Kappes: *Netzwerk- und Datensicherheit*. http://link.springer.com/book/10.1007/978-3-8351-9202-7. Version: 2007. – [Online; Stand 23. November 2016]

[13] RSA Laboratories: *RSAES-OAEP Encryption Scheme*. http://www.inf.pucrs.br/~calazans/graduate/TPVLSI_I/RSA-oaep_spec.pdf. – [Online; Stand 26. Februar 2017]

[14] Stephan Spitz, Michael Pramateftakis und Joachim S.: *Kryptographie und IT-Sicherheit*. Vieweg+Teubner, 2011

[15] Thomas Hartmann: *Whisply: Dateien im Browser mit Ende-zu-Ende Verschlüsselung versenden*. http://www.macwelt.de/news/Whisply-Dateien-im-Browser-mit-Ende-zu-Ende-Verschluesselung-versenden-9890168.html. Version: 2015. – [Online; Stand 26 Oktober 2016]

[16] w3schools: *W3.CSS Tutorial*. https://www.w3schools.com/w3css/. – [Online; Stand 20. Februar 2017]

[17] Wätjen, Dietmar: *Kryptographie*. Spektrum- Akademischer Verlag, 2008

[18] Wikipedia: *XMLHttpRequest — Wikipedia, Die freie Enzyklopädie*. https://de.wikipedia.org/w/index.php?title=XMLHttpRequest&oldid=162213072. Version: 2017. – [Online; Stand 13. März 2017]