Studienarbeit

Auswahl und Implementierung geeigneter kryptographischer Verfahren für die verschlüsselte und signierte Proxy-Kommunikation

Vorgelegt von Frank Thiem

geboren am 25.03.1989 in Finsterwalde

Studiengang Informationssystemtechnik

Betreuer
Dipl.-Inf. Tenshi Hara
Dr.-Ing. Thomas Springer

Verantwortlicher Hochschullehrer
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill

Tag der Einreichung
04. 07. 2014

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Bei der Auswahl und Auswertung des Materials sowie bei der Herstellung des Manuskripts habe ich Unterstützungsleistungen von folgenden Personen erhalten:

- Tenshi Hara
- Thomas Springer

Weitere Personen waren an der geistigen Herstellung der vorliegenden Arbeit nicht beteiligt.

Dresden, 04.07.2013 Frank Thiem

Kurzfassung

Ziel der Studienarbeit war es für das MapBiquitous-Projekt eine geeignete Verschlüsslung und Signierung für die Kommunikation innerhalb der SANE-Architektur zu finden und diese zu implementieren.

Zunächst wurde erfasst welche Methoden bereits existieren und welche davon hier verwendet werden könnten. Dadurch wurde TLS ausgewählt. Um ein möglichst sicheres System bieten zu können, wurden die verwendbaren Cipher Suites auf ein paar wenige reduziert.

Bei der Implementierung wurde der Webserver Apache 2, auf dem der INSANE läuft, passend konfiguriert und ein Prototyp für die Kommunikation mit HTTPS entworfen, wobei dieser noch nicht in die Android-App integriert werden konnte, da es sowohl mit dem ADT als auch mit Android selbst immer wieder Probleme gab, die das Testen der Funktionsfähigkeit der Android-App verhindert haben.

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur Lehrstuhl Rechnernetze

AUFGABENSTELLUNG FÜR DEN GROSSEN BELEG

THEMA: Auswahl und Implementierung geeigneter kryptographischer Verfahren für die verschlüsselte und signierte Proxy-Kommunikation

Name, Vorname:	Thiem, Frank	Studiengang:	IST (DPO 2005)
Matrikel-Nummer:	3479338	Projekt/Schwerpunkt:	MapBiquitous/Mobile
Erster Gutachter:	Dipl.-Inf. Tenshi Hara	Zweiter Gutachter:	Dr.-Ing. Thomas Springer
Beginn am:	06.01.2014	Einzureichen bis:	05.07.2014

ZIELSTELLUNG

SANE ist eine Crowdsourcing-basierte, verteilte Proxy-Plattform zur Bereitstellung von Crowdsourcing-bezogenen Diensten. Im Rahmen des MapBiquitous-Projektes, einem integrierten, ortsbezogenen Dienstes für den Innen- und Außenbereich, wurde erfolgreich die Machbarkeit nachgewiesen. Das Konzept beruht auf einer verteilten Infrastruktur von mobilen Clients, prokurierenden Zwischenentitäten (den SANEs) und Crowdsourcing-Servern, die beliebige Daten und Informationen kommunizieren. Die prokurierte Kommunikation findet derzeit unverschlüsselt statt, weshalb personenbeziehbare Daten ungeschützt übermittelt werden. Die Verwendung von HTTPs ist gegebenenfalls nicht möglich. Auch existiert derzeit keine Authentifizierung der Nutzer gegenüber dem System und umgekehrt.

Ziel der Belegarbeit ist die Untersuchung von kryptographischen Methoden zur Verschlüsselung und Signierung der Kommunikation innerhalb der SANE-Architektur. Der Schwerpunkt der Betrachtungen soll dabei auf effizienten asymmetrischen Kryptographieverfahren für die Verschlüsselung und Signatur liegen. Ein ausgewähltes, bereits existierendes und als zuverlässig nachgewiesenes Verfahren, wie zum Beispiel TLS, soll unter Berücksichtigung der besonderen Anforderungen der prokurierten Kommunikation in den SANE-Prototyp implementiert werden. Erweist sich dies als nicht durchführbar, sollen die konzeptuellen Grundgedanken existierender Verfahren angepasst, erweitert und bedarfsgerecht implementiert werden. Im Anschluss sollen die kryptographisch relevanten Aspekte evaluiert werden. Die Kompatibilität zur Basisarchitektur von MapBiquitous muss gewährleistet bleiben.

SCHWERPUNKTE

- Recherche verwandter Arbeiten zur asymmetrischen Verschlüsselung/Signatur
- Auswahl eines existierenden kryptographischen Verfahrens
- Prototypische Umsetzung des Konzepts
- Erarbeitung einer Evaluationsmethodik
- Evaluation und Bewertung der Ergebnisse

i. A. Braun

Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill
(verantwortlicher Hochschullehrer)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

3DES Triple Data Encryption Standard

ADT Android Developer Tools

AES Advanced Encryption Standard

BSI Bundesamt für Sicherheit in der Informationstechnik

CA Certificate Authority

CBC Cipher Block Chaining

CCM Counter with CBC-MAC

DES Data Encryption Standard

DH Diffie–Hellman

DHE Diffie–Hellman Ephemeral

DSS Digital Signature Standard

ECC Elliptic Curve Cryptography

ECDH Elliptic Curve Diffie–Hellman

ECDHE Elliptic Curve Diffie–Hellman Ephemeral

ECDSA Elliptic Curve Digital Signature Algorithm

ENISA European Union Agency for Network and Information Security Agency

FTP File Transfer Protocol

GCM Galois Counter Mode

GPS Global Positioning System

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

INSANE Indoor Navigation Server Access Network Entity

NSA National Security Agency

PHP PHP: Hypertext Preprocessor

PSK Pre-shared Key

RC4 Ron's Code 4

RSA Kryptosystem von Rivest, Shamir und Adleman

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

WLAN Wireless Local Area Network

1. Einleitung

Kommunikationsnetze sind inzwischen überall in der Gesellschaft integriert und somit nicht mehr wegzudenken. Aber wie in vielen Fällen werden auch Kommunikationsnetze nicht nur für Handlungen genutzt, die man als positiv erachtet. Daher ist es auch hier notwendig, Maßnahmen gegen unerwünschte Zugriffe und Manipulationen zu ergreifen. Dies heißt bei Kommunikationsnetzen, dass Informationen, die übertragen werden sollen, verschlüsselt werden müssen. Außerdem muss überprüfbar sein, ob Informationen korrekt sind bzw. ob sie manipuliert wurden. Um dies zu garantieren sind Konzelations- und Authentikationssysteme unerlässlich.

Um die notwendigen Konzelations- und Authentikationssysteme leicht zugänglich zu machen und um sicherzustellen, dass diese korrekt funktionieren, wurde der Standard Secure Sockets Layer (SSL) veröffentlicht. Dabei wurde SSL bis zur Version 3.0 entwickelt. Danach wurde SSL unter der Bezeichnung TLS weiterentwickelt und liegt zurzeit in der Version 1.2 vor.

Somit ist es möglich und auch notwendig, schon bei kleineren Anwendungen die Kommunikation mit kryptographischen Methoden abzusichern. Dementsprechend wird dies auch bei dem MapBiquitous-Projekt angestrebt. MapBiquitous ist eine Navigationssoftware die innerhalb von Gebäuden navigieren kann. Außerdem soll MapBiquitus mit Hilfe von Crowdsourcing aktuell gehalten werden. Das heißt, dass die Nutzer jederzeit Veränderungen an den entsprechenden Server schicken können, damit der Server die Karte anpassen kann. Damit dies trotz Angriffe korrekt von statten geht, muss man auch hier kryptographische Methoden einsetzen.

1.1. MapBiquitous

Wie der Name schon vermuten lässt, ist MapBiquitous eine Navigationssoftware, die dem Nutzer hilft, sein Ziel zu erreichen. Dabei beschränkt sich MapBiquitous nicht auf die Navigation im Freien, sondern unterstützt auch die Navigation in Gebäuden. Um das bewerkstelligen zu können, nutzt MapBiquitous einerseits Daten von anderen Projekten, wie GoogleMaps oder OpenStreets, für die Navigation im Freien und andererseits separate Daten von Building-Servern für die Navigation in Gebäuden.

1.1.1. Architektur von MapBiquitous

1.1.1.1. Basisarchitektur

MapBiquitous hat eine dezentralisierte Client-Server-Architektur. Wie man in Abb.1.1 sieht, besteht MapBiquitous aus dem Client und mehreren Servern, die verschiedene Aufgaben erfüllen. Der „Directory Service“dient dazu, die Building-Server ausfindig zu machen. Die Building-Server stellen die Informationen für die Navigation in den Gebäuden zur Verfügung. Der „Outdoor Routing Service“repräsentiert hier einen Server von beispielsweise GoogleMaps oder OpenStreets. Für die Navigation werden natürlich nicht nur Kartendaten benötigt sondern auch Positionsdaten. Diese erhält MapBiquitous

über Global Positioning System (GPS) oder Wireless Local Area Network (WLAN). Wobei man die Positionsbestimmung per WLAN besonders für die Navigation innerhalb von Gebäuden benötigt, da GPS im Allgemeinen da kaum bzw. gar nicht einsetzbar ist. [vgl. Spr11]

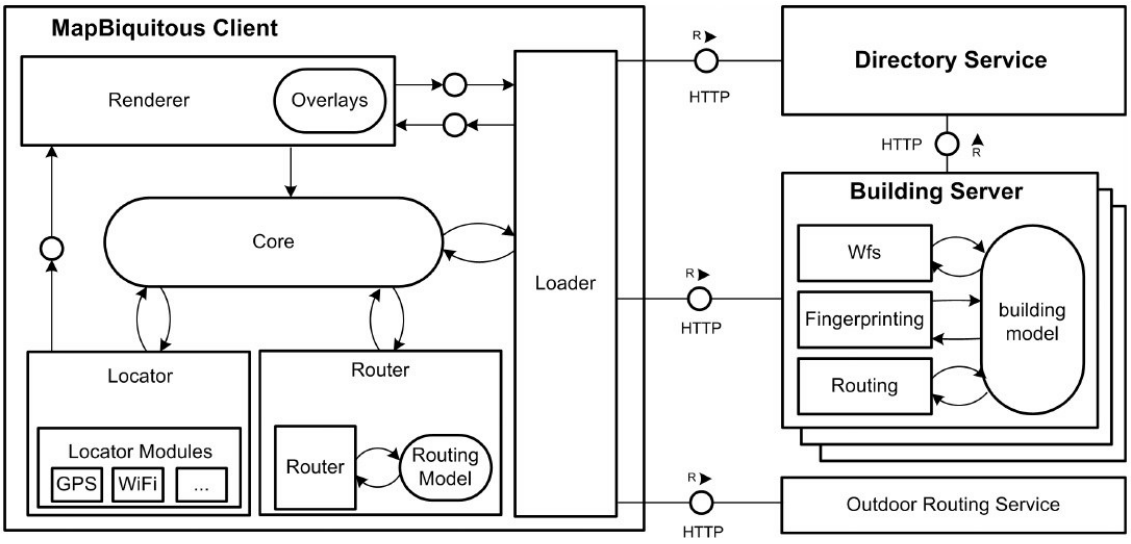

Abb. 1.1.: Basisarchitektur von MapBiquitous [Spr11]

1.1.1.2. Architektur mit Crowdsourcing

Da sich die Umwelt ständig ändert, wurde MapBiquitous so erweitert, dass die Kartendaten schnell und flexibel angepasst werden können. Dazu wurde ein Crowdsourcing-System mit Hilfe eines Indoor Navigation Server Access Network Entity (INSANE) eingebaut. Der INSANE ist ein Proxy zwischen den Clients und den Building-Servern.[Vgl. Har12]

1.1.2. Kommunikation

Da die Clients im Crowdsourcing-System gegenüber den Building-Servern möglichst anonym sein sollen [siehe Har12], ist es notwendig die Kommunikation zwischen Client und INSANE zu verschlüsseln. Denn wenn ein Building-Server den unverschlüsselten Datenverkehr zwischen Client und Server einsehen kann, könnte er anhand dieser Daten Rückschlüsse ziehen, von welchem Client er welche Daten erhalten hat. Auf diese Weise könnte er das Anonymisieren durch den INSANE wirkungslos machen.

1.2. Kryptosysteme

1.2.1. Symmetrische und asymmetrische Kryptosysteme

Bei einem symmetrischen kryptographischen System haben beide Kommunikationspartner den gleichen Schlüssel und können somit gleichermaßen ver- und entschlüsseln. Bei einem asymmetrischen kryptographischen System besitzt jeder Kommunikationspartner sein

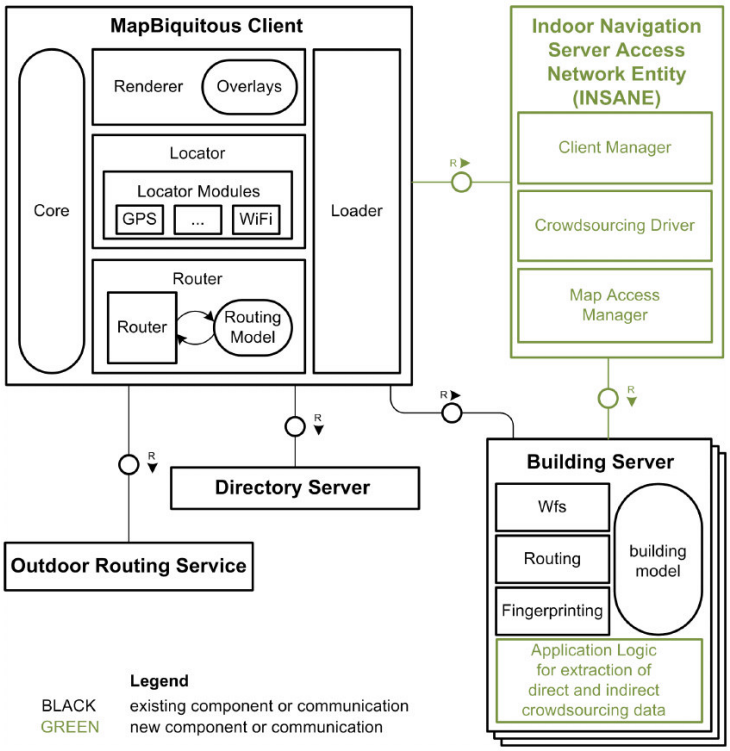

Abb. 1.2.: Architektur von MapBiquitous mit Crowdsourcing [Har12]

eigenes Schlüsselpaar. Dieses Schlüsselpaar besteht aus einem öffentlichen Schlüssel, der an die Kommunikationspartner gesendet wird, und aus einem privaten Schlüssel, der unbedingt geheimgehalten werden muss. Der öffentliche Schlüssel kann zum Verschlüsseln bei einem Konzelationssystem und zum Signatur verifizieren bei einem Signatursystem genutzt werden. Dementsprechend wird der private Schlüssel zum Entschlüsseln bzw. zum Erstellen der Signatur verwendet. Da asymmetrische Kryptoverfahren auf schwierigen mathematischen Problemen basieren, steht die Schlüssellänge im Zusammenhang mit der Rechenleistung aktueller Rechner. Denn bei einem zu kurzen Schlüssel könnte der geheime Schlüssel in akzeptabler Zeit aus dem öffentlichen Schlüssel berechnet werden könnte.

TLS nutzt sowohl symmetrische als auch asymmetrische kryptographische Systeme. Dabei wird wie bei einem typischen Hybridsystem vorgegangen, in dem der Schlüssel für das symmetrische System mit dem asymmetrischen System ausgetauscht wird[Vgl. DR08]. Man verwendet deswegen solche kryptographischen Systeme um die benötigten Rechenkapazitäten zu minimieren.

1.2.2. Blockchiffre und Stromchiffre

Bei den Konzelationssystemen, also bei der Verschlüsselung, wird noch zwischen Stromchiffren und Blockchiffren unterschieden. Mit Stromchiffren kann man Nachrichten beliebiger Länge verschlüsseln. Während man mit Blockchiffren nur Nachrichten bestimmter Länge verwenden kann. Dementsprechend müssen bei Blockchiffren die Blöcke irgendwie aufgefüllt werden, wenn die zu verschlüsselnden Informationen nicht genau auf ein Vielfaches der Blocklänge passen.

1.3. Schutzziele und Angriffe

1.3.1. Schutzziele

Die hier angestrebten Schutzziele sind Vertraulichkeit, Integrität und Zurechenbarkeit. Vertraulichkeit bedeutet, dass Informationen nur Berechtigten bekannt werden. Den Verlust von Vertraulichkeit kann man mit Kryptographie zwar nicht erkennen, aber man kann ihn verhindern. Mit der Integrität verhält es sich genau andersherum. Mit Kryptographie ist man zwar nicht in der Lage die Manipulation von Daten zu verhindern, aber man kann diese erkennen. Dementsprechend bedeutet Integrität, dass die Daten nicht unerkannt modifiziert werden können. Bei der Zurechenbarkeit geht es darum, dass man eine Nachricht einem bestimmten Sender zuordnen kann. Allerdings kann man die Zurechenbarkeit nur mit asymmetrischen Systemen erreichen. [vgl. Fra]

Bei der Umsetzung dieser Schutzziele werden die kryptographischen Systeme nicht in symmetrische und asymmetrische Kryptosysteme eingeteilt, sondern in Konzelationssysteme und Authentikationsysteme. Wobei man Authentikationsysteme auf Basis von asymmetrischen Kryptosystemen im Allgemeinen als Signatursysteme bezeichnet. Konzelationssysteme haben die Aufgabe, die Vertraulichkeit zu gewährleisten. Während die Authentikationssysteme die Integrität garantieren. Wenn man allerdings noch die Zurechenbarkeit erreichen möchte, muss man Signatursysteme einsetzen. [vgl. Fra]

1.3.2. Passive Angriffe

Bei diesen Angriffen nutzt der Angreifer nur die Informationen, bei denen man davon ausgeht, dass sie im Allgemeinen bekannt sind. Zu diesen Informationen zählen das

System, bestehend aus den Algorithmen und den Protokollen, der öffentlichen Schlüssel und die Daten, die man aus der Beobachtung erhält. [vgl. Fra]

1.3.3. Aktive Angriffe

Bei den aktiven Angriffen verwendet der Angreifer nicht nur die öffentlich zur Verfügung stehenden Informationen, sondern er versucht außerdem noch dem Inhaber von geheimen Informationen, wie dem geheimen Schlüssel, zusätzliche Informationen zu entlocken. Dies kann beispielsweise durchgeführt werden, indem der Angreifer das Opfer dazu bringt, eine vom Angreifer gewählte Nachricht zu verschlüsseln. So erhält der Angreifer eine Nachricht mit dem zugehörigen Ciphertext und kann somit eventuell Rückschlüsse auf den Schlüssel ziehen. [vgl. Fra]

1.4. Transport Layer Security

1.4.1. Was ist TLS?

TLS ist auch bekannt als SSL und wird als Standard genutzt, um die Kommunikation zwischen zwei Geräten zu verschlüsseln. Um dies möglich zu machen, werden in dem Standard die Algorithmen festgelegt, die genutzt werden müssen, um eine sichere Verbindung aufzubauen und zu betreiben[vgl. DR08]. Da TLS nur ein Standard ist, gibt es dementsprechend mehrere Varianten für die Umsetzung von diesem. Die bekannteste Variante ist wohl OpenSSL. Was nicht nur daher kommt, dass OpenSSL in vielen Linuxsystemen verwendet wird, sondern auch durch den bekannt gewordenen Heartbleedbug wie unter anderem bei Heise Online [siehe Heib] berichtet wird.

1.4.1.1. DES und 3DES

Data Encryption Standard (DES) ist ein symmetrisches Verschlüsselungsverfahren, das blockweise verschlüsselt. Da DES allein angreifbar ist, wurde Triple Data Encryption Standard (3DES) ins Rennen geschickt, das daraus besteht, dass DES dreimal hintereinander mit verschiedenen Schlüsseln eingesetzt wird. Wobei bei dem ersten Durchgang verschlüsselt, beim zweiten entschlüsselt und beim dritten wieder verschlüsselt wird.[vgl. ST99]

1.4.1.2. Advanced Encryption Standard

Advanced Encryption Standard (AES) ist wie DES ein symmetrisches Verschlüsselungsverfahren und es arbeitet wie DES auch blockweise. Allerdings sind aktuell keine effektiven Angriffe gegen AES bekannt. Daher gilt, dass AES zurzeit sicher ist. AES bietet eigentlich Schlüssellängen von 128bit, 192bit und 256bit, bei TLS wird aber nur 128bit und 256bit eingesetzt.[Vgl. ST01]

1.4.1.3. RC4

Ron's Code 4 (RC4) ist ein symmetrisches Stromverschlüsselungsverfahren. RC4 wird im Allgemeinen nicht empfohlen, da es mehrere Angriffe gibt, die diese Verschlüsselung brechen können [siehe Sma+]. Zu den Institutionen, die empfehlen RC4 zu vermeiden, zählen das Bundesamt für Sicherheit in der Informationstechnik (BSI)[siehe Inf14] und die European Union Agency for Network and Information Security Agency (ENISA)[siehe

Sma+]. Außerdem besteht der Verdacht, dass die National Security Agency (NSA) RC4 in Echtzeit brechen können[siehe Heic].

1.4.1.4. RSA

Das Kryptosystem von Rivest, Shamir und Adleman (RSA) ist ein asymmetrisches Verschlüsselungsverfahren [siehe RSA] und kommt in erster Linie bei Zertifikatssystemen zum Einsatz. Nach [Sma+] empfiehlt die ENISA eine Schlüssellänge von mindestens 3072Bit und für Langzeitsystem sollte sie 15360Bit betragen.

1.4.1.5. DH und ECDH

DH ist ein Algorithmus zum Erzeugen eines geheimen Schlüssels bei zwei Kommunikationspartnern, denen kein sicherer Kanal zur Verfügung steht. Dabei berechnen die zwei Teilnehmer mit den öffentlichen Parametern g und n und ihren jeweiligen geheimen Parametern a bzw. b die öffentlichen Werte Y_a bzw. Y_b, wie man in Abb. 1.3 sieht. Diese Werte werden dann an den jeweiligen Kommunikationspartner geschickt. Mit dem öffentliche Teil des Kommunikationspartners und dem eigenen geheimen Parameter kann man dann den geheimen Schlüssel k bestimmen. [Vgl. DH76]

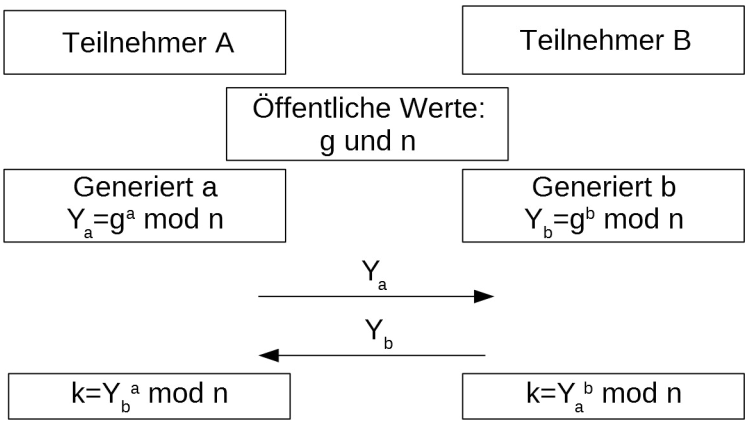

Abb. 1.3.: Funktionsweise von DH [vgl. DH76]

Elliptic Curve Diffie–Hellman (ECDH) funktioniert nach dem gleichen Prinzip wie DH. Der Unterschied ist das bei ECDH Elliptic Curve Cryptography (ECC) verwendet wird.

1.4.2. Funktionsweise und Umfang

TLS kann man in zwei Phasen einteilen. Phase eins ist die Handshake-Phase und Phase zwei ist der Datenaustausch. Die Handshake-Phase wird benötigt, um die Kommunikationspartner zu authentifizieren und um die notwendigen Informationen auszutauschen, damit man in der darauf folgenden Phase die Daten zuverlässig und sicher übermitteln kann. Außerdem ist TLS so konzipiert, dass es ohne Probleme mit Protokollen wie Hypertext Transfer Protocol (HTTP) oder File Transfer Protocol (FTP) zusammen benutzt werden kann.[Vgl. DR08]

1.4.2.1. Cipher Suite

Als Cipher Suite wird bei TLS eine Kombination aus bestimmten Algorithmen bezeichnet. Somit enthält ein Cipher Suite immer die Information, welche Mechanismen man zum Schlüsselaustausch, zum Verschlüsseln des Datenverkehrs und für den Hash nutzt. Ein Cipher Suite wird im Allgemeinen in der Form TLS_RSA_WITH_AES_128_CBC_SHA angegeben. In diesem Beispiel steht nach TLS RSA, was bedeutet, dass das Kryptosystem von Rivest, Shamir und Adleman (RSA) für den Schlüsselaustausch genutzt werden soll. Außerdem wird für das Authentifizieren ebenfalls RSA verwendet, da nach dem RSA und vor dem WITH keine weiteren Angaben stehen. Nach dem WITH steht AES 128 CBC, dies gibt an, wie verschlüsselt werden soll. AES steht dabei für Advanced Encryption Standard (AES) mit einer Schlüssellänge von 128 Bit. CBC gibt die Betriebsart an. Hier ist das der Cipher Block Chaining (CBC) Mode. In Tabelle 1.1 sind die verwendbaren Algorithmen aufgeführt, die durch [DR08], [BW+06], [SCM08], [Res08], [Kim+11], [KK11] und [MB12] festgelegt werden. Wobei annon bedeutet, dass keine Zertifikate zum Einsatz kommen.

Tab. 1.1.: Mögliche Algorithmen

Schlüsselaustausch	Zertifikat	Verschlüsselung mit Betriebsmodus	Hash
RSA	annon	RC4_128	MD5
DH	RSA	3DES_EDE_CBC	SHA
DHE	DSS	AES_128 mit CBC, CCM, CCM_8 oder GCM	SHA256
ECDH	ECDSA	AES_256 mit CBC, CCM, CCM_8 oder GCM	SHA384
ECDHE		ARIA_128 mit CBC oder GCM	
PSK		ARIA_256 mit CBC oder GCM	
		CAMELLIA_128 mit CBC oder GCM	
		CAMELLIA_256 mit CBC oder GCM	

1.4.2.2. Handshake

Wie man in Abbildung 1.4 sieht, kann man die Handshake-Phase noch einmal in vier Unterphasen gliedern. In Phase eins generiert der Client eine Zufallszahl und sendet diese mit einer Liste der verwendbaren Cipher Suites in der ClientHello-Nachricht an den Server. Der Server wiederum generiert daraufhin ebenfalls eine Zufallszahl und wählt aus der Liste der verwendbaren Cipher Suites einen Cipher Suite aus. Die Zufallszahl und den gewählten Cipher Suite schickt er dann an den Client. In Phase zwei sendet der Server sein Zertifikat. Wenn das Zertifikat nicht genug Informationen enthält, um einen Schlüsselaustausch mit dem gewählten Algorithmus durchzuführen, sendet der Server noch eine ServerKeyExchange-Nachricht mit den notwendigen Informationen. Zusätzlich kann der Server dann das Zertifikat vom Client anfordern. Daraufhin überprüft der Client erst einmal das Zertifikat vom Server. In der dritten Phase sendet der Client sein Zertifikat, wenn dies gefordert wurde, an den Server. Danach sendet der Client die ClientKeyExchange-Nachricht mit den notwendigen Informationen zum Schlüssel, mit dem die Daten verschlüsselt werden sollen. In der letzten Phase wird auf die ausgehandelte symmetrische Verschlüsselung gewechselt und beide Teilnehmer senden dann noch die Finished-Nachricht, um zu signalisieren, dass der Handshake beendet ist und der Datenaustausch beginnen kann.[Vgl. DR08]

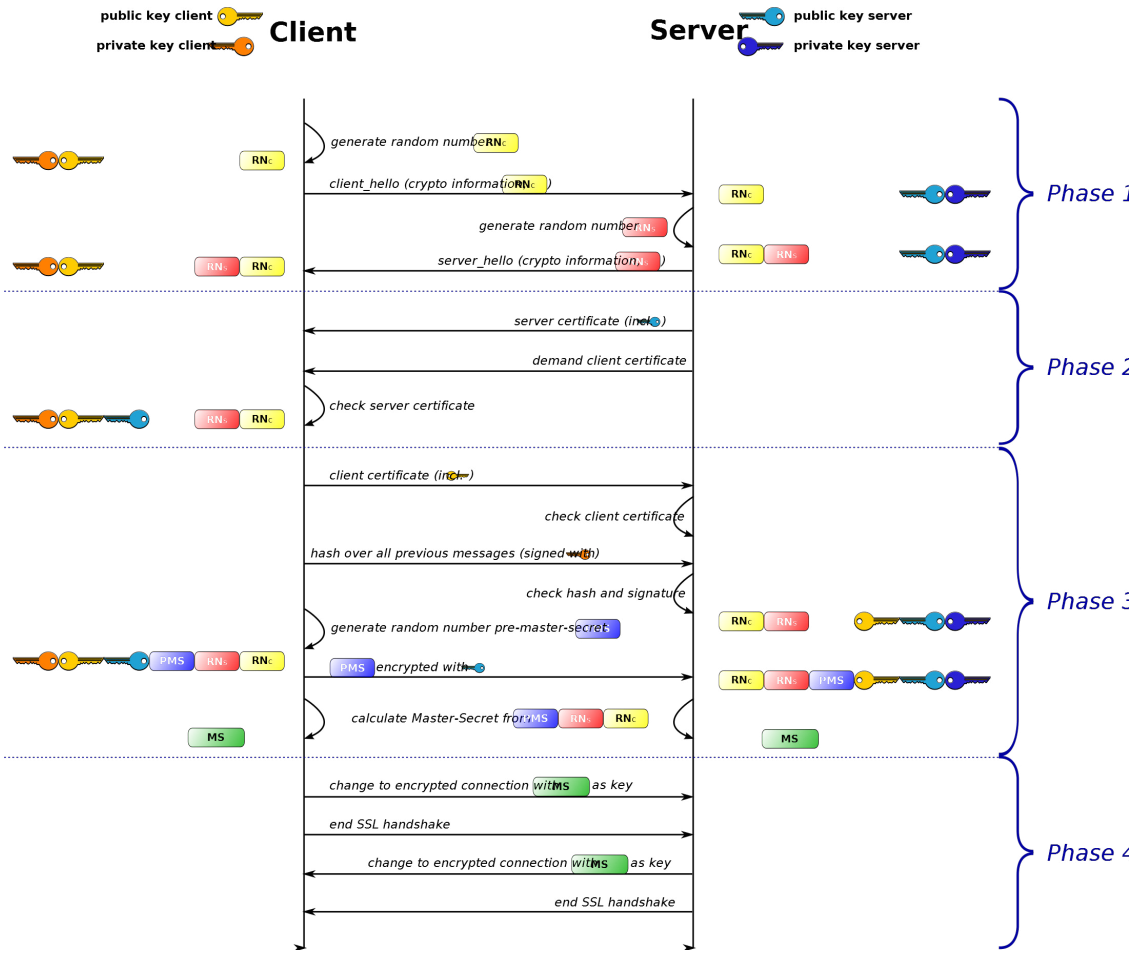

Abb. 1.4.: Handshake von TLS [Wik]

1.4.2.3. Datenaustausch

TLS ist so konzipiert, dass es ohne Probleme zwischen Transportprotokollen, wie Transmission Control Protocol (TCP), und Anwendungsprotokollen, wie HTTP oder FTP, eingesetzt werden kann. Um dies zu ermöglichen, wird in TLS nur festgelegt, dass die Daten vom Record Layer transportiert werden. Außerdem müssen die Daten fragmentiert, komprimiert und entsprechend des Verbindungsstatus verschlüsselt werden.[Vgl. DR08]

2. Konzept

2.1. Verschlüsselung

2.1.1. Vorbetrachtung

Um eine sichere Kommunikation zwischen Client und INSANE zu gewährleisten, standen verschiedene Variante zur Debatte. Eine Variante wäre gewesen, bekannte sichere Algorithmen, wie AES oder 3DES, selbst bzw. mit Hilfe von entsprechenden Bibliotheksdateien zu implementieren, da symmetrische Konzelationssysteme die effektivsten sind. Nun hat man bei symmetrischen Systemen das Problem, dass man einen gemeinsamen Schlüssel benötigt. Dieses Problem könnte man lösen, indem man ein asymmetrisches Konzelationssystem statt eines symmetrischen Konzelationssystems nutzt. Da aber asymmetrische Systeme nicht so effektiv sind und es sich bei den Clients um mobile Geräte handelt, bei denen die Ressourcen im Allgemeinen stark limitiert sind, sollte ein rein asymmetrische System vermieden werden. Hinzukommt, dass die öffentlichen Schlüssel so verteilt werden müssen, dass jeder öffentliche Schlüssel korrekt zugeordnet werden kann. Um ein möglichst effektives System zu haben, bleibt noch die Möglichkeit, dass man ein Hybridsystem einsetzt. Mit dem Hybridsystem würde man das Problem mit dem Austausch des gemeinsamen Schlüssels lösen, allerdings würde es auch das Problem mit dem öffentlichen Schlüssel vom asymmetrischen System übernehmen. Da aber diesem Problem mit dem Konzelationssystem nicht beizukommen ist, werden mögliche Lösungen im Abschnitt Signierungen genauer beleuchtet.

Bei TLS kommt genau so ein Hybridsystem zum Einsatz. Währenden des Handshakes wird ein asymmetrisches Konzelationssystem gewählt, um damit dann den Schlüssel auszutauschen. Im Anschluss wird dann mit dem im Handshake vereinbarten symmetrischen Konzelationssystem verschlüsselt. Daher liegt es hier nahe, eine bereits vorhandene Implementierung wie OpenSSL zu nutzen.

2.1.2. Verwendung von TLS für die Verschlüsselung

TLS bietet verschiedene Varianten für den Schlüsselaustausch. Für den Schlüsselaustausch mit RSA muss der Server ein gültiges Zertifikat besitzen. Wenn man auf Zertifikate verzichten möchte, könnte man auch DH bzw. ECDH einsetzen. Allerdings sollte man auch bei diesen Varianten Zertifikate nutzen, da man sonst sich nicht sicher sein kann mit, wem man gerade eine verschlüsselte Verbindung aufbaut. Daher wäre ein Man-in-the-Middle-Angriff hier sehr leicht durchführbar.

Aber auch die anderen Möglichkeiten mit Zertifikat sind nicht absolut sicher vor Man-in-the-Middle-Angriff. Wie in [Mav+12] gezeigt wird, bestehen nicht nur bei dem veralteten SSL3.0 sondern auch bei dem aktuellen TLS1.2 Standart Angriffsmöglichkeiten. In [Mav+12] werden zwei verschiedene Cross-Protocol-Angriffe aufgeführt. Der erste Angriff ist allerdings ab der TLS-Version 1.0 nicht mehr umsetzbar. Der zweite Angriff hingegen ist noch durchführbar, wobei die Erfolgswahrscheinlichkeit unter 2^{-40} liegt.

Um den schon genannten Cross-Protocol-Angriff vorzubeugen, wurde bestimmte Cipher Suites als nicht akzeptabel aussortiert. Für den Schlüsselaustausch wird dementsprechend nur Elliptic Curve Diffie–Hellman Ephemeral (ECDHE) genutzt. Für die Verschlüsselung der Daten wird AES verwendet. Da dies den Empfelungen nach [Sma+] entspricht. Des Weiteren sollten laut [Sma+] Hash-Funktionen wie MD5 und SHA1 in Zukunft nicht mehr verwendet werden. Die übrigbleibenden Cipher Suites sind in Tabelle 2.1 aufgeführt. Die Verschlüsselung ARIA steht zwar für TLSv1.2 auch zur Verfügung, diese wird aber nicht in [Sma+] erwähnt. Da es aber für ARIA ebenfalls zurzeit keine effektiven Angriffe gibt, kann man diese Verschlüsselung ebenfalls akzeptieren.

Tab. 2.1.: Mögliche Cipher Suites

Schlüsselaustausch	Zertifikat	Verschlüsselung	Betriebsmodus	Hash
ECDHE	RSA	AES_128/256	CBC	SHA256/384
ECDHE	RSA	AES_128/256	GCM	SHA256/384
ECDHE	RSA	CAMELLIA_128/256	CBC	SHA256/384
ECDHE	RSA	CAMELLIA_128/256	GCM	SHA256/384
ECDHE	RSA	ARIA_128/256	CBC	SHA256/384
ECDHE	RSA	ARIA_128/256	GCM	SHA256/384
ECDHE	ECDSA	AES_128/256	CBC	SHA256/384
ECDHE	ECDSA	AES_128/256	GCM	SHA256/384
ECDHE	ECDSA	CAMELLIA_128/256	CBC	SHA256/384
ECDHE	ECDSA	CAMELLIA_128/256	GCM	SHA256/384
ECDHE	ECDSA	ARIA_128/256	CBC	SHA256/384
ECDHE	ECDSA	ARIA_128/256	GCM	SHA256/384

2.2. Signierung

2.2.1. Vorbetrachtung

Da auch Nachrichten vom Client signiert werden sollten, wurden sowohl beim Client als auch beim INSANE Scheinfunktionen für das Signieren und Verifizieren erstellt. Allerdings muss man erst betrachten, gegen was dies schützen kann und soll. Durch ein Signatursystem kann man nachvollziehen, wer eine Nachricht unterzeichnet hat und ob etwas daran verändert wurde, seit sie signiert wurde. Dies ist aber nur möglich, wenn der öffentliche Schlüssel richtig zugeordnet wird. Um dies zu gewährleisten, müsste man über gesicherte Wege die öffentlichen Schlüssel verteilen und jeder Server bzw. Proxy, der mit den Clients kommuniziert, müsste alle öffentlichen Schlüssel speichern. Die andere Variante wäre, die öffentlichen Schlüssel durch eine dritte vertrauenswürdige Instanz signieren zu lassen. Auf die Art könnte man sicherstellen, dass einem Kommunikationspartner nicht der falsche öffentliche Schlüssel zugeordnet wird.

2.2.2. TLS und Signaturen

Bei TLS muss man zunächst berücksichtigen, dass nicht jede Nachricht signiert wird, sondern lediglich im Handshake. Dies ist aber vollkommen ausreichend, da für das nachfolgende Konzelationssystem nur die beiden Kommunikationspartner den Schlüssel

besitzen. Nun gibt es bei TLS auch das Problem der Vertrauenswürdigkeit der öffentlichen Schlüssel. TLS arbeitet mit Zertifikaten um diesem Problem entgegenzuwirken. Wobei in einem Zertifikat nicht nur der öffentlichen Schlüssel, sondern auch weitere Informationen enthalten sind. Ein oft verwendetes Zertifikatsformat ist das X509-Format. Damit man diese Zertifikate nutzen kann, benötigen sie aber brauchbare Signaturen. Was nun wieder zum ursprünglichen Problem führt. Allerdings muss man noch unterscheiden, ob sich Clients ebenfalls mit Zertifikaten authentifizieren müssen oder ob auf eine Client-Authentifizierung verzichtet wird.

2.2.2.1. TLS ohne Client-Zertifikaten

Wenn man Zertifikate selbst signieren lassen würde, hätte man wieder das Problem, dass man die Zertifikate wie Certificate Authority (CA)-Zertifikate behandeln und über sichere Kanäle verteilen müsste. Was, wenn man sich nicht um die Client-Zertifikate kümmern muss durchaus durchführbar aber aufwendig ist, da man immer noch die Zertifikatsliste so verteilen muss, dass jeder die richtig Zertifikatsliste erhält. Den Clients könnte man zum Beispiel eine aktuelle Liste der Zertifikate per Update der App zukommen lassen.

Eine Alternative wäre es separate CA-Stelle zu nutzen. Diese CA-Stelle könnte dann die Zertifikate der INSANEs und der Building-Server bestätigen. Das CA-Zertifikat könnte man in der Installation bzw. in den Updates mitliefern. In Abb. 2.1 sieht man den Aufbau diese Systems.

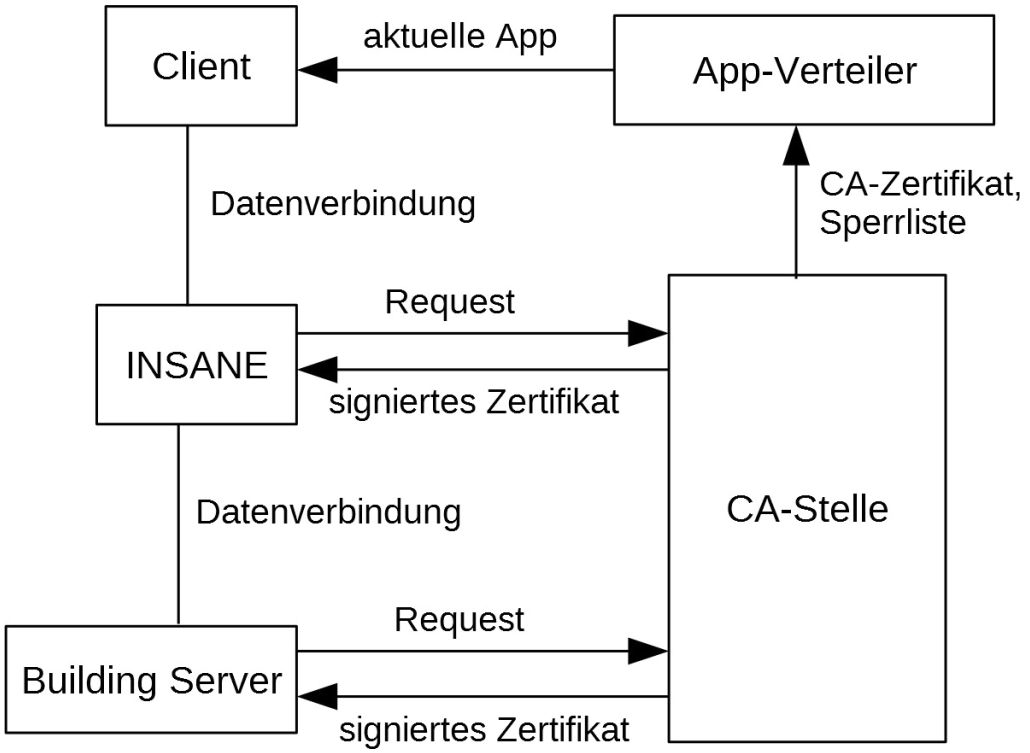

Abb. 2.1.: Verteilung der Zertifikate ohne Client-Zertifikate

2.2.2.2. TLS mit Client-Zertifikaten

Wenn man die Authentifizierung der Clients auf der Ebene von TLS durchführen möchte, ist es notwendig, dass man jedem Client ein gültiges Zertifikat zuordnen kann. Wenn dies ohne ein separate CA-Stelle betrieben werden soll, muss mit alle Teilnehmer eine Liste mit gültigen Zertifikaten synchron halten, was wieder Probleme beinhaltet, da die Synchronisation über einen sicheren Kanal stattfinden muss. Da aber Zertifikate im Allgemeinen nur für eine begrenzte Zeit gültig sind, bedeutet das, dass Teilnehmer mit ungültigen Zertifikaten nur über Umwege wieder eingebunden werden können.

Eine günstigere Methode wäre es, eine separate CA-Stelle zu nutzen. Diese CA-Stelle wäre für die Registrierung neuer Nutzer und entsprechend für das Signieren neuer Zertifikate verantwortlich. Für den Fall, dass ein Client ein neues Zertifikat benötigt, wird eine separate Anmeldemöglichkeit gebraucht, damit die CA-Stelle nur dann das Zertifikat signiert, wenn es auch wirklich zum Client gehört. Außerdem bedarf es noch einer Sperrliste für die Zertifikate, die zwar noch gültig wären, aber keine Sicherheit mehr gewährleisten. Dies ist der Fall wenn der geheime Schlüssel zu einem Zertifikat von einem Angreifer kopiert werden konnte.

Neben der Variante, dass nur eine CA-Stelle zum Einsatz kommt, gibt es hier auch noch die Variante, dass man zwei CA-Stellen einsetzt. Diese Variante wird in Abb. 2.2 dargestellt. Die erste CA-Stelle wäre dann, wie bei dem Fall, dass sich die Clients nicht mit Zertifikaten authentifizieren müssen, nur für die Bestätigung der Zertifikate, die für die Building-Server und INSANEs eingesetzt werden, verantwortlich. Dieses CA-Zertifikat und die zugehörige Sperrliste müssen dann alle Clients erhalten. Die zweite CA-Stelle ist dann wie in der vorherigen Variante für die Clientverwaltung verantwortlich. Allerdings auch nur für diese. Dementsprechend würden alle INSANEs und Building-Server beide CA-Zertifikate benötigen, damit sie sowohl mit den Clients als auch untereinander kommunizieren können. Damit die Clients mit der zweiten CA-Stelle eine sichere Verbindung aufbauen können, würde die zweite CA-Stelle ein Zertifikat benötigen, das von der ersten CA-Stelle signiert wurde, oder die Clients würden ebenfalls beide CA-Zertifikate benötigen.

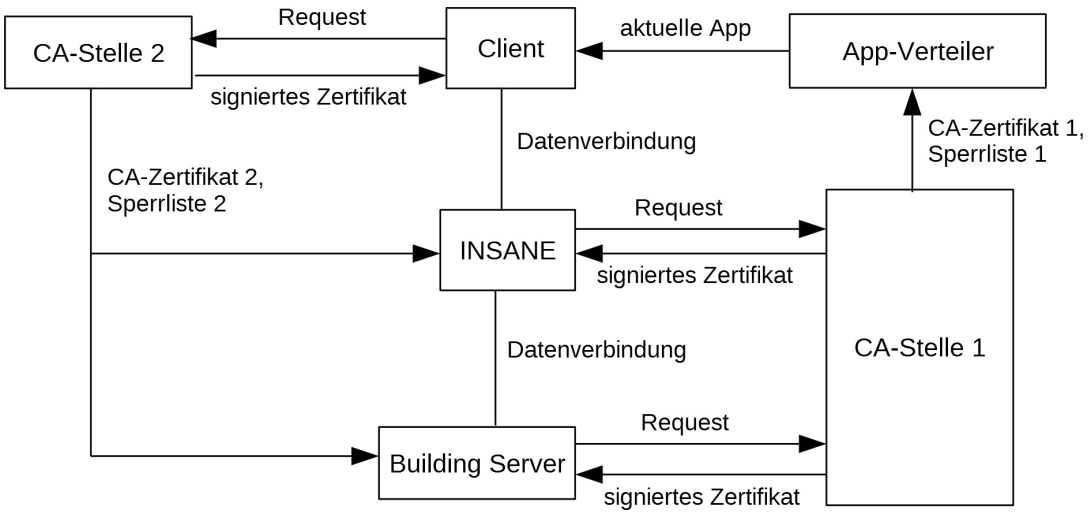

Abb. 2.2.: Verteilung der Zertifikate mit Client-Zertifikate

2.3. Angriffsmodell

Bei den Angriffsmodellen werden zwei unterschiedliche betrachtet. Das erste Angriffsmodell beinhaltet einen Angreifer, der das Netzwerk uneingeschränkt überwachen und manipulieren kann. Allerdings besitzt dieser Angreifer nur eingeschränkte Rechenkapazitäten bzw. Ressourcen. Beim zweiten Angriffsmodell wird davon ausgegangen, dass der Angreifer Zugang bzw. die Kontrolle über einen Teilnehmer erlagen konnte.

2.3.1. Verschlüsselung

Ein Angriff auf das Konzelationsystem hätte den Verlust der Vertraulichkeit als Ziel. Da bisher noch keine effektiven Angriffe gegen AES bekannt sind, liegen die Gefahren nicht in der Verschlüsselung selbst. Dementsprechend gibt es kaum Möglichkeiten die Verschlüsselung unbrauchbar zu machen.

2.3.1.1. Man in the Middle

Eine Variante wäre, dass der Angreifer versuchen müsste den Schlüssel zu ergattern. Dies wäre möglich wenn der Angreifer während des Handshakes alle Daten mitliest und aus diesen Informationen dann den Schlüssel für die Verschlüsselung berechnet. Da dies aber ohne entsprechend leistungsfähige Computer extrem aufwendig ist, müsste der Angreifer die Kommunikation komplett zwischenspeichern, um sie dann später entschlüsseln zu können. Dies würde einem passiven Angriff entsprechen und wäre nicht erkennbar.

Ein aktiver Angriff wäre zum Beispiel ein Man-in-the-Middle-Angriff wie in [Mav+12] beschrieben. Allerdings arbeitet dieser Angriff mit unterschiedlichen Cipher Suites, wenn man aber gezielt nur einen bestimmten Algorithmus für den Schlüsselaustausch zulässt, wäre ein derartiger Cross-Protocol-Angriff nicht mehr möglich. Um der Empfehlung aus [Sma+] zu folgen sollte man ECDHE für den Schlüsselaustausch nutzen. Falls man allerdings Kompatibilität zu älteren Geräten herstellen möchte, ist es eventuell notwendig von ECDHE auf DH oder RSA zu wechseln. Wenn man DH und RSA noch zusätzlich aktivieren möchte, sollte man zumindest sicherstellen, dass die TLS-Implementierung schwache Werte wie in Tabelle2.2 nicht zulässt und dass nur TLS und nicht SSL verwendet wird. Die Tabelle 2.2 bezieht sich auf $Y = g^x \bmod p$ vom DH-Schlüsselaustausch.

Tab. 2.2.: Schwache Werte für Y_s bei DH [Mav+12]

	$Y_s = 1$	$Y_s = 0$	$Y_s = -1$
NSS 3.12.6	Accept	Accept	Accept
OpenSSL 1.0.1	Reject	Reject	Reject
GnuTLS 3.0.18	Accept	Accept	Accept

2.3.1.2. Korrumpierter Teilnehmer

Ein korrumpierter Teilnehmer ist für das Konzelationssystem selbst kaum eine Gefahr, da die verwendeten Schlüssel nur für eine Verbindung gültig sind. Dementsprechend könnte ein Angreifer nur die Daten erhalten, die bei dem Teilnehmer gespeichert sind und die Daten, die übertragen werden während der Angreifer Zugriff hat.

2.3.2. Signierung

Hier wäre das Ziel eines Angriffs, dass sich der Angreifer als ein Teilnehmer ausgeben kann, der er nicht ist. Außerdem kann ein Angreifer das Ziel haben, gezielt nur bestimmte Nachrichten zu modifizieren. Die Sicherheit von Signatursystemen ist aufgrund ihrer schwierigen Umkehrfunktionen davon abhängig, welche Rechenkapazitäten der Angreifer hat. Eine andere Variante wäre es, wenn der Angreifer versucht, auf dem entsprechenden Rechner einzudringen und den privaten Schlüssel zu stehlen. Eine weitere Möglichkeit wäre es dass Opfer dazu zu bringen, die gewünschten Nachrichten zu signieren.

2.3.2.1. Man in the Middle

Da bei einem Angriff der Angreifer kaum brauchbare Informationen erhält, bleibt dem Angreifer nur das Erraten oder das aufwendige Berechnen des geheimen Schlüssels. Dementsprechend ist die Gefahr durch einen passiven Angriff sehr gering.

Der Angriff von [Mav+12] ist ein Beispiel wie ein aktiver Angriff aussehen kann. Da bei diesem Angriff nicht nur das Konzelationsystem außer Kraft gesetzt wird, sondern auch der Angreifer sich für diese Verbindung als Server ausgibt, zählt dieser aktive Angriff hier auch mit hinein.

2.3.2.2. Korrumpierter Teilnehmer

Wenn ein Teilnehmer durch einen Angreifer übernommen wurde oder geheime Daten durch den Angreifer kopiert wurden, müssen Zertifikate, von denen die zugehörigen geheimen Schlüssel vom Angreifer kopiert wurden, auf die entsprechende Sperrliste gesetzt werden. Da sich sonst der Angreifer als Opfer ausgeben kann, selbst wenn das Opfer bereits ein neues Zertifikat verwendet. Wenn allerdings keine CA-Stelle verwendet wurde, muss man das korrumpierte Zertifikat aus der Liste der gültigen Zertifikate entfernen. Falls dies aber nicht rechtzeitig geschieht, könnt sich der Angreifer eventuell mit eigenen Zertifikaten in die Liste eintragen.

2.4. Ausgewähltes System

Insgesamt folgt daraus, dass es am sinnvollsten ist, TLS mit eingeschränkten Cipher Suites zu nutzen. Bei den Zertifikaten ist es das Beste, eine CA-Stelle zu nutzen, die die Zertifikate der INSANEs und der Building-Server signiert. Während man ein zweite CA-Stelle für die Client nutzt. Wobei man hinsichtlich älterer Android-Clients vorsichtig sein sollte, da es anscheinend Sicherheitslücken beim KeyStore gibt, wie in [Heia] geschrieben wird.

3. Implementierung

3.1. Voraussetzungen

Um TLS nutzen zu können, wird ein SSL-Provider benötigt. Diese Rolle übernimmt meistens OpenSSL. Allerdings gibt es auch Alternativen dazu wie GnuTLS, die dies auch übernehmen können. Der Client wurde für Androidgeräte in der Programmiersprache Java programmiert. Die Verschlüsselung ist laut [Goob] seit API-1 verfügbar. In der App für MapBiquitous sind allerdings Funktionen enthalten die mindestens API-9 benötigen. Beim Server ist ein Webserver notwendig, der sowohl PHP als auch Hypertext Transfer Protocol Secure (HTTPS) unterstützt. Ein Webserver, der das bietet, ist zum Beispiel der Apache 2.

3.2. Aktivierung von TLS

3.2.1. Einbinden von TLS beim Client

Um in der aktuellen App die Cipher Suites einzuschränken, musste als erstes die SSLSocketFactory angepasst werden. Dazu wurde eine neue Klasse mit der Bezeichnung DefCipherSuitesSSLSocketFactory erstellt. Im Anhang A.1.1 befindet sich der Quellcode dieser Klasse. In der Variable „Cipher_Suites“werden die akzeptierten Cipher Suites festgelegt. Die Konfiguration wie sie nachfolgend zu sehen ist, wurde so gewählt, da die Zertifikate, die für die Test verwendet wurden, mit RSA-Schlüsseln arbeiteten.

```
\label{code:testjcs}
private static final String[] CIPHER_SUITES = {
        "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384",
        "TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256"};
```

Wenn man alle möglichen Cipher Suites, wie in 2.1 aufgeführt, aktivieren wollte, würde der Abschnitt wie folgt aussehen.

```
private static final String[] CIPHER_SUITES = {
        "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384",
        "TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256",
        "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384",
        "TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256",
        "TLS_ECDHE_RSA_WITH_CAMELLIA_256_CBC_SHA384",
        "TLS_ECDHE_RSA_WITH_CAMELLIA_128_CBC_SHA256",
        "TLS_ECDHE_RSA_WITH_CAMELLIA_256_GCM_SHA384",
        "TLS_ECDHE_RSA_WITH_CAMELLIA_128_GCM_SHA256",
        "TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384",
        "TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256",
        "TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384",
        "TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256",
```

```
        "TLS_ECDHE_ECDSA_WITH_CAMELLIA_256_CBC_SHA384",
        "TLS_ECDHE_ECDSA_WITH_CAMELLIA_128_CBC_SHA256"
        "TLS_ECDHE_ECDSA_WITH_CAMELLIA_256_GCM_SHA384",
        "TLS_ECDHE_ECDSA_WITH_CAMELLIA_128_GCM_SHA256"};
```

Allerdings muss man hier berücksichtigen, dass Android standardmäßig OpenSSL verwendet. Bei OpenSSL ist momentan aber kein ARIA implementiert. Dementsprechend müsste man eine anderen SSL-Provider nutzen. In [Con11] wird beschrieben wie man den SSL-Provider in Android wechseln kann. Außerdem sollte man wenn man zum Beispiel nur Elliptic Curve Digital Signature Algorithm (ECDSA)-Zertifikate verwendet die Cipher Suites für RSA entfernen.

Um jede Verbindung zu verschlüsseln, muss in den Klassen AsyncIndoorRouteTask, AsyncOutdoorRouteTask, AsyncINSANETask, GeocodingTask und AsyncHttpRequest unter anderem HttpURLConnection durch HttpsURLConnection ersetzt und mit der DefCipherSuitesSSLSocketFactory verknüpft werden. Wie dies vollständig aussieht, findet man im Anhang unter A.1.2.

Der nachfolgende Teil ist für das Laden des CA-Zertifikats verantwortlich. Hier wird dies aus einer Datei geladen, wenn man aber das CA-Zertifikat aus einer anderen Quelle beziehen möchte, muss man diesen Part entsprechend anpassen.

```
CertificateFactory cf = CertificateFactory.getInstance("X.509");
InputStream is = new BufferedInputStream(new FileInputStream("
    cert.crt"));
Certificate ca = cf.generateCertificate(is);
KeyStore ts = KeyStore.getInstance(KeyStore.getDefaultType());
keyStore.load(null, null);
keyStore.setCertificateEntry("ca", ca);
TrustManagerFactory tmf = TrustManagerFactory.getInstance(
    TrustManagerFactory.getDefaultAlgorithm());
tmf.init(ts);
TrustManager[] tm = tmf.getTrustManagers();
```

Bei der Anwendung von SecureRandom weist das ADT auf eine mögliche Unsicherheit mit dieser Klasse hin. Außerdem wird man auf [Gooa] verwiesen. Da erfährt man, dass SecureRandom nicht immer kryptographisch starke Werte liefert.

3.2.2. Einbinden von TLS beim INSANE

Um bei dem INSANE die Verschlüsselung nutzen zu können, wurde die Konfiguration so angepasst, dass der Webserver ebenfalls nur die schon in Tabelle2.1 aufgeführten Cipher Suites akzeptiert.

```
SSLCipherSuite -ALL:ECDHE-RSA-AES256-SHA384:ECDHE-RSA-AES128-
    SHA256:!EXP:!aNULL:!MD5:!SHA1:!NULL:!RC4:!RC2:!DES:!3DES:!
    IDEA:!DH:!DHE:!RSA
```

Mit dieser Zeile werden die zulässigen Cipher Suites auf TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 und TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 beschränkt, so wie es auch beim Client gemacht wurde. Wie diese Zeichenkette aufgebaut ist, hängt von OpenSSL ab. Dementsprechend kann man mit Befehl „openssl ciphers -v "[String]"“auf

der Konsole die aktiven Cipher Suites anzeigen lassen, wobei [String] durch die entsprechende Zeichenkette zu ersetzen ist. Mit „:“werden die einzelnen Optionen voneinander getrennt. Durch „-ALL“werden zunächst alle Cipher Suites deaktiviert. Danach werden die gewünschten Cipher Suites hinzugefügt. Wenn man nicht nur einzelne Cipher Suites sondern ein ganze Gruppe von Cipher Suites auf ein mal hinzufügen möchte, kann man entsprechende Abkürzungen nutzen, die auf [Apa] oder auf [Opeb] zu finden sind. Die nachfolgenden angegeben Gruppen mit dem „!“verbieten die Cipher Suites, die diese Algorithmen benutzen, und sorgen so dafür, dass diese auch nicht mehr nachträglich hinzugefügt werden können. Um den Befehl „openssl“nutzen zu können, benötigt man OpenSSL, das man zum Beispiel auf [Opea] erhält.

3.3. Zertifikate

3.3.1. Voraussetzungen

Wie auch schon bei der Verschlüsselung wird auch bei der Signierung keine spezielle Android API vorausgesetzt, da alle notwendigen Komponenten seit API 1 vorhanden sind. Dies ändert aber nichts daran, dass MapBiquitous mindestens die API 9 benötigt. Für den INSANE gelten hier auch die gleichen Voraussetzungen wie bei der Verschlüsselung.

3.3.2. Generierung der Zertifikate

3.3.2.1. Selbstsignierte Zertifikate

Das Generieren der Zertifikate wurde mit OpenSSL durchgeführt. Mit dem nachfolgenden Beispiel wird ein selbst signiertes Zertifikat mit Schlüssel erzeugt. Wenn man ECDSA einsetzen möchte, muss man die Kommandos entsprechend anpassen.

```
openssl req -x509 -days 365 -newkey rsa:3072 -out cert.crt -
    keyout key.key
```

3.3.2.2. Durch CA signierte Zertifikate

Um ein durch CA-Stelle signiertes Zertifikat zu erhalten, muss man zuerst ein Request erstellen. Hier in diesem Beispiel wird ein neuer RSA-Schlüssel erstellt und der dazugehörige Request erzeugt.

```
openssl req -newkey rsa:3072 -out request.csr -keyout key.key
```

Im zweiten Schritt signiert die CA-Stelle den Request und erzeugt so das endgültige Zertifikat.

```
openssl x509 -req -days 365 -in request.csr -CA ca.pem -CAkey ca
    .key -out cert.crt
```

Mit dem fertigen Zertifikat kann sich dann der Antragssteller gegenüber jedem, der das CA-Zertifikat erhalten hat und akzeptiert, authentifizieren. Daher ist es hier besonders wichtig, dass man keine falschen CA-Zertifikate akzeptiert.

3.3.3. Konfiguration des Apache

Bei der Apache-Konfiguration muss man die Orte angeben, wo sich der Schlüssel bzw. das Zertifikat befinden, mit dem sich der Server authentifizieren soll. Dazu werden die ersten beiden Zeilen benötigt. Im zweiten Teil wird angegeben, wo sich das CA-Zertifikat befindet. Mit diesem Zertifikat werden dann die Zertifikate der Clients überprüft. Mit SSLVerifyClient wird festgelegt, ob sich ein Client authentifizieren muss oder nicht. Durch SSLVerfiyDepth wird bestimmt wie lang die Zertifikatskette sein darf, bevor das Zertifikat des Clients als ungültig abgelehnt wird. Hier ist dieser Wert auf eins gesetzt, was bedeutet, dass nur Zertifikate, die direkt durch die CA-Stelle signiert wurden, oder Zertifikate, die selbst signiert wurden, akzeptiert werden.

```
SSLCertificateFile "server.crt"
SSLCertificateKeyFile "server.key"

SSLCACertificateFile "caclient.crt"
SSLVerifyClient require
SSLVerifyDepth  1
```

3.3.4. Clients

Um eine sichere Verbindung aufzubauen, wurde bereits unter 3.2.1 gezeigt, wie man das CA-Zertifikat einbinden kann. Um eine Authentifizierung des Clients zu ermöglichen, muss natürlich das Zertifikat und der zugehörige geheime Schlüssel des Clients geladen werden. Dafür wurde der nachfolgende Teil eingefügt. Hier wird ebenfalls aus einer Datei der geheime Schlüssel und das zugehörige Zertifikat geladen. Falls man aber keine Authentifizierung des Clients auf der TLS-Ebene möchte, kann man diesen Teil auch entfernen.

```
KeyManagerFactory kmf = KeyManagerFactory.getInstance(
    KeyManagerFactory.getDefaultAlgorithm());
KeyStore ks = KeyStore.getInstance("PKCS12");
FileInputStream fisk = new FileInputStream("sicher.p12");
ks.load(fisk, "secret".toCharArray());
kmf.init(ks, "secret".toCharArray());
KeyManager[] km = kmf.getKeyManagers();
```

4. Validierung

4.1. Android und das ADT

Von Anfang an gab es immer wieder Probleme mit dem Android Developer Tools (ADT). Das ADT soll die Entwicklung von Apps für Android erleichtern, hier hat es allerdings eher Probleme bereitet. Zu diesen Problemen zählten unter anderem, dass Kommentare in der Entwicklungsumgebung als Fehlerquelle angezeigt wurden und dass man für Datentypen wie „String“aufgefordert wurde, eine neue Klasse zu erstellen. Die Probleme mit den Datentypen konnte man beheben, indem man die passenden Bibliotheken von Hand eingebunden hat. Derartige Fehler wurden anscheinend weitestgehend behoben. Eine weitere Problemquelle waren die Virtuel Devices. Die meisten Virtuel Devices waren sehr langsam und aufgrund ihrer extrem langsamen Reaktionszeiten waren sie nur sehr schwer zu bedienen. Manche Virtuel Devices wie der Intel Atom x86 haben komplett den Dienst verweigert und sind direkt beim Starten abgestürzt.

Da MapBiquitous auf den virtuellen Geräten überhaupt nicht laufen wollte, wurde die App auf ein ASUS TF300 installiert. Dies war aber ebenfalls nicht von Erfolg gekrönt, da die App, wenn sie mal gestartet war, nach ein bis zwei Sekunden abstürzte. Um dennoch die Funktionsfähigkeit der Implementierung zu testen, wurde versucht den relevanten Teil in eine Test-App auszulagern. Da diese ebenfalls nicht funktionierte, wurde der Quellcode in möglichst kleine Teile zerlegt, um den Ursprung des Problems zu finden. Dabei wurde festgestellt das Problem bei HttpsURLConnection.connect() liegt. Um ein Problem mit TLS auszuschließen, wurde es auf HTTP reduziert. Dies zeigt aber nur das mit HttpURLConnection.connect() den gleichen Fehler verursachte. Da die Test-App nun auch nutzlos geworden war, wurde der Quellcode mit Hilfe von NetBeans in eine normale Java-Anwendung ausgelagert. Die HTTP-Variante lief dabei fehlerfrei durch. Das gilt auch für die HTTPS-Verbindung.

4.2. Test der Apache-Konfiguration

Um zu überprüfen, ob mit der Apache-Konfiguration auch das erreicht wurde, was man wollte, wurden der Webbrowser Firefox und der Webbrowser Chrome für Android eingesetzt. Da Firefox nur TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 von den möglichen Cipher Suites(siehe 2.1) unterstützt, wurde die Konfiguration des Webservers dafür so angepasst, dass nur dieser Cipher Suite akzeptiert wird. Zusätzlich wurde der Schlüssel und das Zertifikat von Firefox geladen, um sich beim Webserver authentifizieren zu können. Nach dem erfolgreichen Verbinden mit dem Webserver liefert Firefox die Informationen, die in Abb. 4.1 zu sehen sind.

Bei Android wurden das Zertifikat des Webservers und das Schlüsselpaar für den Client systemweit hinzugefügt. Da Firefox dies aber nicht unterstützt, wurde hier auf Chrome ausgewichen. Mit Chrome konnte man ohne Probleme das passende Schlüsselpaar des Clients auswählen und sich authentifizieren. Bei der Verschlüsselung gab Chrome das Gleich wie auch schon vorher Firefox an.

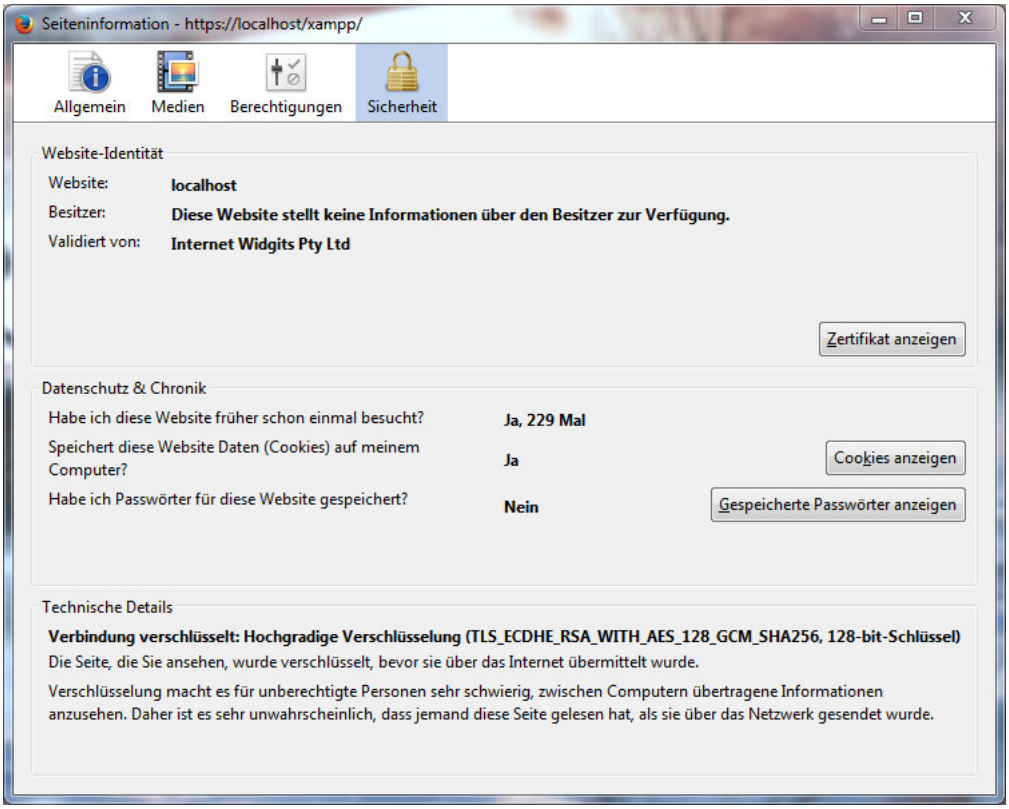

Abb. 4.1.: Seiteninformationen mit Cipher Suite in Firefox

4.3. Test außerhalb von Android

Da unter Android nichts Testfähiges erstellt werden konnte, wurde mit NetBeans eine einfache Java-Anwendung zum Testen geschrieben. Der Quellcode befindet sich im Anhang unter A.2. Diese Anwendung lädt das Zertifikat des Server und das Schlüsselpaar zum Authentifizieren aus den entsprechenden Dateien und nachdem es sich mit Hilfe dieser Informationen mit dem Webserver verbunden hat, wurde der Cipher Suite ausgegeben. Da aber hier keine OpenSSL, sondern JSSE erweitert durch JCE Unlimited Strength Jurisdiction Policy Files zum Einsatz kommt, muss man hier darauf Rücksicht nehmen, dass weder ARIA noch CAMELLIA unterstützt werden und dass der Betriebsmode Galois Counter Mode (GCM) nicht verwendet werden kann. Dementsprechend kamen hier TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 und TLS_ECDHE_RSA_WITH_AES_256 _CBC_SHA384 zum Einsatz.

5. Zusammenfassung und Ausblick

5.1. Zusammenfassung

In dieser Arbeit wurde zuerst betrachtet, wie man den Datenaustausch am besten verschlüsseln und signieren könnte. Dabei wurde nicht nur nach entsprechende Algorithmen gesucht, sondern auch nach schon vorhandenen Implementierungen. Eine Implementierung, die dem genügt, ist OpenSSL. Insbesondere da Android standardmäßig OpenSSL nutzt. Da aber OpenSSL nicht nur starke Cipher Suites von TLSv1.2 unterstützt, war es notwendig die existierenden Cipher Suites genauer unter die Lupe zu nehmen und eventuell fragwürdige und ungünstige Cipher Suites auszusortieren. Dementsprechend blieben nur ein paar zurück, die für die Verschlüsselung infrage kamen. Da für das Signieren öffentliche Schlüssel notwendig sind, wurde hier auf Zertifikate zurückgegriffen, die im TLS-Handshake eingesetzt werden. Des Weiteren wurden Ansätze zur Verteilung dieser Zertifikate erstellt.

In der Implementierungsphase wurde der Webserver Apache 2, auf dem der INSANE lief, von der Konfiguration her so angepasst, dass nur die Cipher Suites, die vorher in der Konzeptphase festgelegt wurden,akzeptiert werden. Beim Client wurde dafür eine neue Klasse erstellt, um die Sockets mit den eingeschränkten Cipher Suites zu erzeugen. Außerdem wurde der Quellcode so angepasst, das die Clients ein CA-Zertifikat laden können, um ihren Kommunikationspartner authentifizieren zu können. Damit sich die Clients auch beim Server authentifizieren können, wurde das Laden von einem Zertifikat und dem zugehörigen Schlüssel ebenfalls eingebaut.

Beim Testen konnte nur über Umwege nachgewiesen werden, dass der Webserver die Konfiguration richtig umsetzt, da bei den Android-Apps grundlegende Funktionen Fehler verursacht haben, die nicht behoben werden konnten. Daher wurde der relevante Teil des Quellcodes ausgelagert in eine einfache Java-Anwendung. So konnte sichergestellt werden, dass der Quellcode funktioniert. Dabei wurde darauf geachtet, dass nur Klassen und Funktionen verwendet wurden, die laut [Goob] auch zur Verfügung stehen.

5.2. Ausblick

Da die genaue Ursache für das Problem mit HttpURLConnection.connect() nicht festgestellt werden konnte, müsste man erst dies lösen damit der Java-Quellcode von Nutzen sein kann. Des Weiteren wäre zu überlegen ob man nicht ein System für die Nutzerverwaltung anlegen sollte, um entsprechend automatisiert für die Clients Zertifikate ausgeben zu können. Außerdem könnte man mit anonymisierten Zertifikaten die Nachverfolgbarkeit minimieren. Dies könnte in der Form geschehen, dass die Zertifikate keine clientbezogenen Daten außer einer temporären User-ID beinhalten. Diese temporäre User-ID wäre dann so lange gültig wie das Zertifikat. So könnt zwar die Nutzerverwaltung jeder Zeit sagen, welche Nutzer welche User-ID hat bzw. hatte. Aber die Nutzerverwaltung erfährt nicht wann von wem welche Daten übertragen wurden. Der INSANE hingegen kann nur sagen wann welche Daten übertragen wurden aber nicht von welchem Nutzer da er nur weis das es sich um eine

legitimen Nutzer handelt. Bis hierhin bräuchte man die User-ID eigentlich noch nicht. Wenn man aber ein Belohnungssystem einbauen möchte, muss man Nutzer und Handlungen in Verbindung bringen können. Durch die User-ID kann der INSANE der Nutzerverwaltung eine Hinweis senden, so dass die Nutzerverwaltung dem Nutzer eine entsprechende Belohnung gutschreiben kann. Wobei der Hinweis nur ein Minimum an Informationen enthalten sollte. Damit die Nutzerverwaltung keine bzw. kaum Rückschlüsse auf den Nutzer zeihen kann. Dies liefert natürlich immer noch keine vollständige Anonymität. Falls man aber echte Anonymität und nicht nur Pseudonymisierung haben möchte, wird man wohl um ein Anonymisierungssystem, wie zum Beispiel einem MIX-Netzwerk nicht drumherum kommen.

A. Anhang

A.1. Quellcode für den Androidclient

A.1.1. DefCipherSuitesSSLSocketFactory

```
public class DefCipherSuitesSSLSocketFactory extends SSLSocketFactory {

  private static final String[] CIPHER_SUITES = {
        "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384",
        "TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256"};

  private final SSLSocketFactory factory;

  public DefCipherSuitesSSLSocketFactory(SSLSocketFactory factory){
    this.factory = factory;
  }

  @Override
  public String[] getDefaultCipherSuites() {
    return CIPHER_SUITES;
  }

  @Override
  public String[] getSupportedCipherSuites() {
        return CIPHER_SUITES;
  }

  @Override
  public Socket createSocket(Socket s, String host, int port, boolean autoClose) throws IOException {
    Socket socket = this.factory.createSocket(s, host, port, autoClose);
    ((SSLSocket) socket).setEnabledCipherSuites(CIPHER_SUITES);
    return socket;
  }

  @Override
  public Socket createSocket(String host, int port) throws IOException, UnknownHostException {
    Socket socket = this.factory.createSocket(host, port);
    ((SSLSocket) socket).setEnabledCipherSuites(CIPHER_SUITES);
```

```
34      return socket;
    }
36
    @Override
38  public Socket createSocket(String host, int port, InetAddress
        localHost, int localPort) throws IOException,
        UnknownHostException {
      Socket socket = this.factory.createSocket(host, port,
          localHost, localPort);
40    ((SSLSocket) socket).setEnabledCipherSuites(CIPHER_SUITES);
      return socket;
42  }

44  @Override
    public Socket createSocket(InetAddress host, int port) throws
        IOException {
46    Socket socket = this.factory.createSocket(host, port);
      ((SSLSocket) socket).setEnabledCipherSuites(CIPHER_SUITES);
48    return socket;
    }
50
    @Override
52  public Socket createSocket(InetAddress address, int port,
        InetAddress localAddress, int localPort) throws IOException
         {
      Socket socket = this.factory.createSocket(address, port,
          localAddress, localPort);
54    ((SSLSocket) socket).setEnabledCipherSuites(CIPHER_SUITES);
      return socket;
56  }

58 }
```

A.1.2. Quellcode für HTTPS-Verbindung

```
    URL url = new URL("https://"+host+":"+port);
2   HttpsURLConnection urlConnection=(HttpsURLConnection)url.
        openConnection();
    SSLContext context = SSLContext.getInstance("TLSv1.2");
4
    CertificateFactory cf = CertificateFactory.getInstance("X.509");
6   InputStream is = new BufferedInputStream(new FileInputStream("
        cert.crt"));
    Certificate ca = cf.generateCertificate(is);
8   KeyStore ts = KeyStore.getInstance(KeyStore.getDefaultType());
    keyStore.load(null, null);
10  keyStore.setCertificateEntry("ca", ca);
```

```
TrustManagerFactory tmf = TrustManagerFactory.getInstance(
    TrustManagerFactory.getDefaultAlgorithm());
tmf.init(ts);
TrustManager[] tm = tmf.getTrustManagers();

KeyManagerFactory kmf = KeyManagerFactory.getInstance(
    KeyManagerFactory.getDefaultAlgorithm());
KeyStore ks = KeyStore.getInstance("PKCS12");
FileInputStream fisk = new FileInputStream("sicher.p12");
ks.load(fisk, "secret".toCharArray());
kmf.init(ks, "secret".toCharArray());
KeyManager[] km = kmf.getKeyManagers();

context.init(km, tm, new SecureRandom());
SSLSocketFactory defCipherSuitesSSLSocketFactory = new
    DefCipherSuitesSSLSocketFactory(context.getSocketFactory());
urlConnection.setSSLSocketFactory(
    defCipherSuitesSSLSocketFactory);
urlConnection.setRequestMethod("GET");
urlConnection.setDoInput(true);
urlConnection.connect();
```

A.2. Testclient

```
package testsclient;

import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.MalformedURLException;
import java.net.ProtocolException;
import java.net.URL;
import java.security.KeyManagementException;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.security.UnrecoverableKeyException;
import java.security.cert.Certificate;
import java.security.cert.CertificateException;
import java.security.cert.CertificateFactory;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.net.ssl.HttpsURLConnection;
import javax.net.ssl.KeyManager;
import javax.net.ssl.KeyManagerFactory;
```

```
import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLSocketFactory;
import javax.net.ssl.TrustManager;
import javax.net.ssl.TrustManagerFactory;

public class TestSClient {

  public static void main(String[] args) {
    try {
      URL url = new URL("https://Server:443");
      HttpsURLConnection urlConnection=(HttpsURLConnection)url.
          openConnection();

      SSLContext context = SSLContext.getInstance("TLSv1.2");

      TrustManagerFactory tmf = TrustManagerFactory.getInstance(
          TrustManagerFactory.getDefaultAlgorithm());
      KeyStore ts = KeyStore.getInstance(KeyStore.getDefaultType
          ());
      FileInputStream fist = new FileInputStream("testserver.crt
          ");
      CertificateFactory cf = CertificateFactory.getInstance("X
          .509");
      InputStream caInput = new BufferedInputStream(fist);
      Certificate ca = cf.generateCertificate(caInput);
      ts.load(null, null);
      ts.setCertificateEntry("Server", ca);
      tmf.init(ts);
      TrustManager[] tm = tmf.getTrustManagers();

      KeyManagerFactory kmf = KeyManagerFactory.getInstance(
          KeyManagerFactory.getDefaultAlgorithm());
      KeyStore ks = KeyStore.getInstance("PKCS12");
      FileInputStream fisk = new FileInputStream("sicher.p12");
      ks.load(fisk, "pass".toCharArray());
      kmf.init(ks, "pass".toCharArray());
      KeyManager[] km = kmf.getKeyManagers();
      context.init(km, tm, new SecureRandom());
      SSLSocketFactory defCipherSuitesSSLSocketFactory = new
          DefCipherSuitesSSLSocketFactory(context.
          getSocketFactory());
      urlConnection.setSSLSocketFactory(
          defCipherSuitesSSLSocketFactory);

      urlConnection.connect();
      System.out.println(urlConnection.getCipherSuite());

    } catch (MalformedURLException ex) {
```

```
63          Logger.getLogger(TestSClient.class.getName()).log(Level.
                SEVERE, null, ex);
        } catch (ProtocolException ex) {
65          Logger.getLogger(TestSClient.class.getName()).log(Level.
                SEVERE, null, ex);
        } catch (IOException ex) {
67          Logger.getLogger(TestSClient.class.getName()).log(Level.
                SEVERE, null, ex);
        } catch (NoSuchAlgorithmException ex) {
69          Logger.getLogger(TestSClient.class.getName()).log(Level.
                SEVERE, null, ex);
        } catch (KeyStoreException ex) {
71          Logger.getLogger(TestSClient.class.getName()).log(Level.
                SEVERE, null, ex);
        } catch (CertificateException ex) {
73          Logger.getLogger(TestSClient.class.getName()).log(Level.
                SEVERE, null, ex);
        } catch (KeyManagementException ex) {
75          Logger.getLogger(TestSClient.class.getName()).log(Level.
                SEVERE, null, ex);
        } catch (UnrecoverableKeyException ex) {
77          Logger.getLogger(TestSClient.class.getName()).log(Level.
                SEVERE, null, ex);
        }
79    }

81  }
```

Literatur

[Apa] *Apache Module mod_ssl*. URL: http://httpd.apache.org/docs/2.4/mod/mod_ssl.html.

[BW+06] Simon Blake-Wilson u. a. *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)*. RFC 4492 (Informational). Updated by RFCs 5246, 7027. Internet Engineering Task Force, Mai 2006. URL: http://www.ietf.org/rfc/rfc4492.txt.

[Con11] Chris Conlon. *Installing an Alternate SSL Provider on Android*. Mai 2011.

[DH76] Whitfield Diffie und Martin E. Hellman. *New Directions in Cryptography*. Nov. 1976.

[DR08] Tim Dierks und Eric Rescorla. *The Trasport Layer Security (TLS) Protocol Version 1.2*. RFC 5246 (Proposed Standard). Internet Engineering Task Force, Aug. 2008. URL: http://www.ietf.org/rfc/rfc5246.txt.

[Fra] Elke Franz. *Vorlesung Kryptographie und Kryptoanalyse*.

[Gooa] *Some SecureRandom Thoughts*. URL: https://android-developers.blogspot.com/2013/08/some-securerandom-thoughts.html.

[Goob] *Übersicht über die Funktionen der APIs*. URL: https://developer.android.com/reference/.

[Har12] Tenshi Hara. »TOWARDS A RELIABLE ARCHITECTURE FOR CROWDSOURCING IN CONTEXT OF THE MAPBIQUITOUS PROJECT«. Diplomarbeit. Technische Universität Dresden, 2012.

[Heia] *Android 4.3 und älter: Schwachstelle in der Schlüsselverwaltung*. URL: http://heise.de/-2237970.

[Heib] *Der GAU für Verschlüsselung im Web: Horror-Bug in OpenSSL*. URL: http://heise.de/-2165517.

[Heic] *NSA entschlüsselt Webserver-Daten angeblich in Echtzeit*. URL: http://heise.de/-2041383.

[Inf14] Bundesamt für Sicherheit in der Informationstechnik. *Kryptographische Verfahren: Empfehlungen und Schlüssellängen*. 2014.

[KK11] Satoru Kanno und Masayuki Kanda. *Addition of the Camellia Cipher Suites to Transport Layer Security (TLS)*. RFC 6367 (Informational). Internet Engineering Task Force, Sep. 2011. URL: http://www.ietf.org/rfc/rfc6367.txt.

[Kim+11] WooHwan Kim u. a. *Addition of the ARIA Cipher Suites to Transport Layer Security (TLS)*. RFC 6209 (Informational). Internet Engineering Task Force, Apr. 2011. URL: http://www.ietf.org/rfc/rfc6209.txt.

[MB12] D. McGrew und D. Bailey. *AES-CCM Cipher Suites for Transport Layer Security (TLS)*. RFC 6655 (Proposed Standard). Internet Engineering Task Force, Juli 2012. URL: http://www.ietf.org/rfc/rfc6655.txt.

[Mav+12] Nikos Mavrogiannopoulos u. a. *A Cross-Protocol Attack on the TLS Protocol*. Raleigh, 2012.

[Opea] *Website von OpenSSL*. URL: https://www.openssl.org/.

[Opeb] *ciphers(1)*. URL: https://www.openssl.org/docs/apps/ciphers.html.

[RSA] R.L. Rivest, A. Shamir und L. Adleman. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*.

[Res08] Eric Rescorla. *TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode (GCM)*. RFC 5289 (Informational). Internet Engineering Task Force, Aug. 2008. URL: http://www.ietf.org/rfc/rfc5289.txt.

[SCM08] Joseph Salowey, Abhijit Choudhury und David McGrew. *AES Galois Counter Mode (GCM) Cipher Suites for TLS*. RFC 5288 (Proposed Standard). Internet Engineering Task Force, Aug. 2008. URL: http://www.ietf.org/rfc/rfc5288.txt.

[ST01] National Institute of Standards und Technology. *FIPS PUB 197: ADVANCED ENCRYPTION STANDARD (AES)*. Nov. 2001.

[ST99] National Institute of Standards und Technology. *FIPS PUB 46-3: Data Encryption Standard (DES)*. Okt. 1999.

[Sma+] Nigel P. Smart u. a.

[Spr11] Thomas Springer. *MapBiquitous - An Approach for Integrated Indoor/Outdoor Location-based Services*. 01062 Dresden, Germany: Technische Universität Dresden, 2011.

[Wik] URL: https://commons.wikimedia.org/wiki/File:SSL_handshake_with_two_way_authentication_with_certificates.svg.