# TLS meets Merkle & McEliece

von

*Elias Most, Benedikt Moneke und Florian Zouhar*

WEIRD SCIENCE CLUB

DARMSTADT

**an der**

**Lichtenbergschule**

**Europaschule, MINT-Excellence Center,**

**Internationale Begegnungsschule**

**Ludwigshöhstr. 105**

**64285 Darmstadt**

# Zusammenfassung

Unser Projekt beschäftigt sich mit der Integration des quantensicheren CMSS-Signaturverfahrens in das TLS Protokoll, das beispielsweise beim Online-Banking benutzt wird. Eine Signatur dient hierbei dafür, um sicherzustellen, dass man auch mit dem richtigen Server eine Verbindung herstellt und nicht mit einem als anderer Server getarnten. Somit wird gewährleistet, dass vertrauenswürdige Daten auch an den richtigen Adressaten geschickt werden.
Da heute verwendete Signaturverfahren wie RSA auf mathematischen Problem basieren, sind sie zwar für heutige Computer sicher, aber im kommenden Zeitalter der Quantencomputer können diese relativ einfach geknackt werden. Dadurch wäre die Sicherheit der Daten nicht gewährleistet.
Deshalb haben wir ein quantencomputersicheres Siganturverfahren namens CMSS in einen modifizierten TLS-Client und in einen selbstgeschriebenen TLS-Server und HTTP-Server eingebaut, um zu demonstrieren, dass dieses Verfahren genau so effizient wie aktuelle Verfahren ist, aber sicherer als RSA oder ähnliche Verfahren.
Zusätzlich haben wir das McEliece-Verfahren für die Übertragung des Verbindungspassworts benutzt, da aktuelle Verfahren wie Diffie-Hellman nicht quatencomputersicher sind.
Wir haben dann über diese Verbindung Dateien, wie z.B. eine HTML-Datei übertragen, und in einem von uns geschriebenen Browser mit TLS-Merkle-Unterstützung angezeigt.
Als nächstes ist der Aufbau einer TLS-Verbindung mit CMSS über das Internet zur Übertragung von Experiment-Daten für ein anderes Projekt des Weird Science Clubs geplant, sowie eine Integration in den Browser Firefox und in den Http-Server Tomcat.

# Inhaltsverzeichnis

# Einleitung

Im heutigen Zeitalter der digitalen Informationstechnik werden praktisch alle wichtigen Daten am Computer verwaltet: Online-Banking, ELSTER(digitale Steuererklärung), Online-Shopping etc. . Es ist bequem und einfach mal schnell z.B. seine Miete im Internet zu überweisen, aber ist es auch sicher?
In den Medien ist sehr häufig vom so genannten Phishing, dem Datenklau von z.B. sensiblen Bankdaten, die Rede. Dieses Phishing geschieht unter anderem über Emails, mit denen nichts ahnende Kunden auf gefälschte Seiten großer Banken gelenkt werden, die sie dazu auffordern, vertrauliche Informationen, wie PIN und TAN, die man für das Online-Banking benötigt, dort einzugeben. Tut der Kunde dies, so kann er damit rechnen, dass sich sein Konto in der nächsten Zeit wie von selbst leert, nur leider ist es meistens zu spät bis er den Betrug bemerkt, denn er war sich sicher, dass die Seite, auf der er seine Daten eingegeben hatte, auch seiner Bank gehört, denn es stand die vermeintlich richtige Adresse im Browser.
Um nun zu überprüfen, ob nun solche Seiten wirklich von der Bank kommen, gibt es schon sein längerem eine Lösung: Zertifikate. Diese Zertifikate werden nur von wenigen Firmen wie z.B. VeriSign ausgestellt, und sind durch eine digitale Signatur geschützt, damit man sie nicht kopieren kann. Diese Signaturverfahren gelten momentan als relativ sicher, da sie auf mathematischen Problemen wie beispielsweise der Primfaktorzerlegung beruhen, die man mit heutigen Rechnern nicht effizient genug knacken kann.
Allerdings steht ein neues Zeitalter vor der Tür, das der Quanten.

# Problemstellung und Zielsetzung

Diese Quanten mögen zwar an sich relativ harmlos erscheinen, und man könnte sich auch fragen, wie denn solche kleinsten Teilchen mathematische Probleme zu lösen vermögen. Hier kommen nun physikalische Gesetze ins Spiel, die es einem Quantencomputer ermöglichen, diese Probleme relativ schnell zu lösen, da er mit so genannten Qubits, der quantenmechanischen Entsprechung des Bits, die z.B. die Richtung des Spins zur Darstellung des Zustands nutzen, arbeitet. Die eingegeben Daten, dargestellt durch quantenmechanische Objekte mit bestimmten Eigenschaften, interferieren mit den bereits vorhandenen Quanten oder werden gezielt verschränkt, sodass die Lösung mit anderen Möglichkeiten superponiert, und zwar so, dass am Ende alle anderen Lösungen vernichten werden(Kollaps der Wellenfunktion) und nur die gesuchte Lösung übrig bleibt. Da diese physikalischen Vorgänge sehr schnell ablaufen, kann man in naher Zukunft, wenn die Entwicklung der Quantencomputer weiter fortschreitet, mit neuen leistungsfähigeren Computern rechnen, die heutige Sicherheitsverfahren mit geringem Zeitaufwand brechen werden.
Deshalb ist es sehr wichtig, dass man schon heute anfängt, sich über diese Probleme Gedanken zu machen und auch versucht eine Lösung zu finden, also etwas, das genauso sicher wie heutige Verfahren ist, aber nicht von einem Quantencomputer gebrochen werden kann.
Wir haben uns dieses Problems angenommen und haben ein Signaturverfahren der TU Darmstadt von einem Team um Johannes Buchmann[JB06] gefunden: CMSS, ein erweitertes Merkle-Signature-Sheme. Da die Gefahr beim Online-Banking immer mehr zunimmt. haben wir uns entschlossen, uns das Protokoll, das zum sicheren Datenaustausch benutzt wird, einmal genauer anzusehen und Merkle dort einzubauen. Als Ziel haben wir uns dabei gesetzt, Merkle vollständig in dieses Protokoll zu integrieren, einen kleinen HTTP-Server zu schreiben, der über dieses modifizierte Protokoll eine Webseite zu einem kompatiblen Browser schickt. Als Programmiersprache wurde Java von Sun Microsystems gewählt, da CMSS von der TUD in Java geschrieben wurde, und wir im Informatikunterricht bereits viel Erfahrung mit dieser Sprache gesammelt hatten.

# CMSS

## Aufbau MSS

CMSS,Coronado-Merkle Signature Sheme, ist eine verbesserte Version des Signaturverfahrens MSS, Merkle Signature Scheme, welches im Gegensatz zu den heute gängigen Signaturverfahren, RSA, DSA..., quantencomputersicher ist, da es nicht auf einen mathematischen Problem wie z.B. dem Faktorisieren großer Zahlen in Primfaktoren beruht.
MSS basiert auf dem Winternitz One-Time-Signature Sheme (OTSS), mit dem man Einmalsignaturen erzeugen kann, ein solches Schlüsselpaar besteht aus einem Signatur- und einem Verifizierungsschlüssel. Die Sicherheit dieses Verfahrens basiert auf der Existenz einer sicheren kryptografischen Hashfunktion.
Das Merkle-Signature-Sheme besteht aus bis zu $2^h$ Winternitz Signaturen, die mit einem Privatenschlüssel, der die Signierungsschlüssel enthält, erstellt und mit einem öffentlichen Schlüssel, der die Verfizierungsschlüssel enthält. Er ist als binärer Baum aufgebaut, in dem der Weg von einem Blatt bis zur Wurzel der

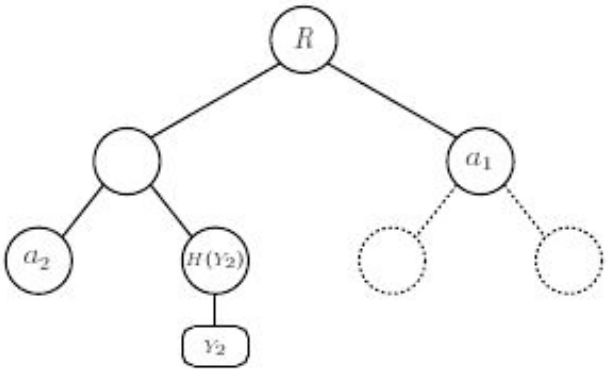


Abb.1 Merkle-Baum[JB06]

Verifizierungsweg für die Signatur ist. In der untersten Reihe sind die einzelnen Schlüsselpaare für das Signaturverfahren. Durch eine Hashfunktion kann man aus zwei unteren einen übergeordneten Knoten berechnen. Es gibt somit $2^h$ Signierungsschlüssel $X_i$ und dazugehörige $2^h$ Verifizierungsschlüssel $Y_i$.Mit den einzelnen Signierschlüsseln, kann man jeweils eine Datei verschlüsseln. Mit der Datei wird die fortlaufende Nummer i des Schlüsselpaares, der Verifizierungsschlüssel $Y_i$, die Signatur und der Verifikationspfad A geschickt, um die Verifikation zu ermöglichen.

Beim Verifizieren einer solchen Signatur wird nun zuerst die Einmalsignatur mit dem Verifizierungsschlüssel $Y_i$ überprüft, danach wird überprüft, ob der Verifizierungsschlüssel auch vom öffentlichen MSS Schlüssel stammt. Dazu wird der sog. Verifizierungspfad berechnet.
Der Verifikationspfad besteht aus allen „Nachbarknoten“ des Weges im Baum von dem Verifikationsschlüssel bis zur Wurzel.So kann man mit $Y_i$ und dem Pfad, bestehend aus den Nachbarblättern $a_1$, $a_2$,...,$a_i$ die Wurzel R berechnen. Wenn die ausgerechnete Wurzel mit dem Hashwert des öffentlichen Schlüssel übereinstimmt, ist der Verifikationsschlüssel echt und wenn dann mit dem Verifikationsschlüssel die Datei positiv auf Echtheit überprüft worden ist, stimmt der angegebene Absender mit dem tatsächlichen Absender überein.

**Aufbau CMSS**

Aufgrund der Komplexität des Baumes bei großer Höhe(großem h), welche für eine ausreichende Anzahl an Signaturen benötigt wird, da jede Signatur nur einmal verwendet werden kann, ist der Rechenaufwand nicht unerheblich. Deshalb wurde MSS zu CMSS verbessert, indem man nicht einen großen Baum erzeugt, sondern diesen in viele kleine Bäume zerlegt, welche immer durch den übergeordneten Baum signiert werden, sodass weiterhin nur die oberste Wurzel der öffentliche Schlüssel ist, man aber die Anzahl der Schlüsselpaare deutlich erhöhen kann ohne die nötige Rechenzeit überproportional zu erhöhen.

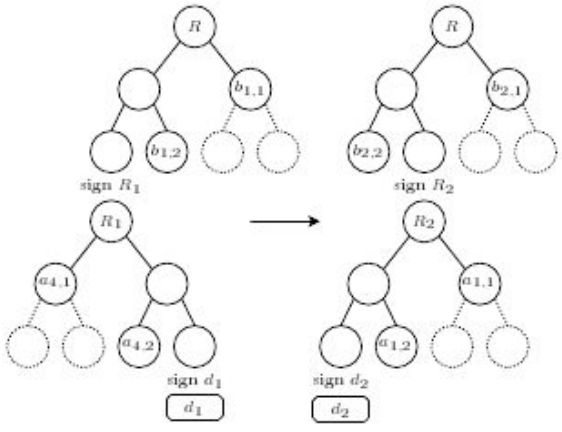


Abb.1 CMSS-Bäume[JB06]

**Vor- und Nachteile**

Die privaten Schlüssel sind aufgrund der verkürzten Baumhöhe eines einzelnen Baumes kleiner und werden schneller generiert als in MSS, sodass die alte Grenze von $2^{26}$ Schlüsselpaaren auf eine effektive Anzahl von $2^{40}$ Schlüsselpaaren

angehoben werden konnte. Weiterhin ist CMSS bzw. MSS das einzige ökonomisch-quantencomputersichere Signaturverfahren. Alle anderen sind entweder unsicher oder benötigen eine äußerst teure Infrastruktur um laufen zu können. CMSS-Signaturen sind nicht viel größer als Signaturen anderer Verfahren, welche unsicher sind und langsamer signieren/verifizieren. Die weiterhin geringe Größe im Bereich unter 10 KByte ist bei heutigen Datenverbindungen zu vernachlässigen.

# McEliece

### Problem Diffie-Hellman

Das TLS-Protokoll benutzt zur Schlüsselübertragung normalerweise das Diffie-Hellman-Verfahren. Da dieses Verfahren allerdings auf dem Problem des Berechnens diskreter Logarithmen basiert, die nachweislich von Quantencomputern effizient berechnet werden können, entschieden wir uns das Verfahren durch das McEliece-Verfahren zu ersetzen.

### Das Verfahren

Das McEliece-Verfahren wurde 1978 als erstes Public-Key Verfahren von Robert McEliece  entwickelt. Es basiert auf der Sicherheit eines binären, fehlerkorrigierenden Codes, in dem Fall des Goppa Codes. Bei der Schlüsselgenerierung wird eine Generatormatrix erzeugt, die das Produkt zweier zufälliger geheimer Matrizen und einer Goppa-Code Matrix ist, welche eine bestimmte Anzahl von t Fehlern zulässt.
Der öffentliche Schlüssel ist nun die Generatormatrix und t, der geheime Schlüssel ist ein effizienter Dekodieralgorithmus für den Goppa Code, der t Fehler korrigieren kann,  und die beiden geheimen Matrizen.
Zum Verschlüsseln multipliziert man die Nachricht mit der Generatormatrix und addiert einen Fehler-Vektor z mit bis zu t Fehlern.
Zum Entschlüsseln korrigiert man erst mit dem Dekodieralgorithmus die Fehler und führt multipliziert dann mit den inversen Matrizen.
Das McEliece Verfahren hat den Nachteil, dass die Chiffre ca.1,6 mal so lang wie der eigentliche Text ist, und die Schlüssel mit 400kB relativ groß ausfallen. Allerdings gehen wir davon aus, dass im PQC-Zeitalter die

Internetverbindungen schnell genug sind, so dass die Schlüsselgröße nicht weiter ins Gewicht fällt.

# Transport Layer Security

**Einführung**
Häufig muss man über eine Verbindung Daten schicken, welche vor dem Zugriff Dritter geschützt werden müssen. Eine Möglichkeit einer sicheren Verbindung ist TLS, der Nachfolger von SSL. Bei TLS werden die Daten verschlüsselt übermittelt, sodass sie vor dem Lesen durch Dritte geschützt sind.

**Aufbau**
Das TLS-Protokoll besteht aus zwei großen Teilen, zuerst dem so genannten Handshake, zu deutsch: Handschlag, bei dem die sichere Verbindung über eine unsichere Verbindung gestartet wird, dann das ChangeCipherSpec-Protokoll, das die Verschlüsselung startet, dies wird durch den Empfang eines Bytes mit dem Wert 1 initiiert. Zur Fehlerübermittlung gibt es auch noch ein Protokoll, dessen Pakete aus 2 Byte bestehen, im ersten steht die Dringlichkeit (fatal: Sofortiger Verbindungsabbruch, warning: Eine Warnung ohne sofortigem Verbindungsabbruch) und im zweiten Byte steht der Inhalt der Warnung. Im letzten Teil, der Nutzphase, werden die Daten der Benutzer verschlüsselt gesendet.

**Der Handshake**
Zuerst sendet der Client, meist als Teil des Browsers, eine Anfrage an einen Server, das so genannte Client-Hello, in dem er seine eigenen Verschlüsselungsmöglichkeiten mitteilt.
Daraufhin sendet der Server sein Hello als Antwort und sagt welche Verschlüsselungsmethode er von den angegebenen benutzen will, außerdem schickt er sein Zertifikat um sich zu authentifizieren, damit der Cient sicher gehen kann, dass er auch mit dem richtigen Server und nicht mit einem Phishing- oder gehacktem Server aus Fernost eine Verbindung aufbaut. Das Zertifikat ist spezifiziert als X.509-Zertifikat. Ein solches Zertifikat besteht aus mehreren Teilen, zuerst der Versions- und Zertifikatsseriennummer, die CertificateAuthority(CA) meist mit Issuer bezeichnet,  dann die Subject Information, also auf wen das Zertifikat ausgestellt ist,und dessen öffentlicher Schlüssel, der zum z.B. für die

Überprüfung von Signaturen verwedet werden kann. Bei neueren Versionen können noch Erweiterungen kommen; als Abschluss kommt eine Signatur, welche das Zertifikat signiert und somit dessen Echtheit beweist. Diese Signatur wird von der CA angefertigt.
An diesem Punkt kann der Server auch von dem Client ein Zertifikat verlangen, was aber unüblich ist. Jetzt übermittelt er das Pre-Master-Secret mittels McEliece, welches verhindert, dass Dritte an die Schlüssel kommen, wobei die Informationen, in unserem Fall mit CMSS, signiert werden, damit sich kein Dritter zwischen Client und Server schalten kann und die ganze Verbindung über ihn läuft.
Jetzt ist der Client wieder an der Reihe und übermittelt falls gefordert sein Zertifikat, und sendet das Pre-Master-Secret an den Server.
Mit den unterschiedlichen Zufallszahlen, die Client und Server am Anfang erzeugt haben, und dem Pre-Master-Secret können beide jetzt das Master-Secret berechnen, mit dem der Datenverkehr der Nutzphase dann verschlüsselt wird.
Zum Abschluss schickt er das ChangeCipherSpec Protokoll und wechselt danach in den Verschlüsselungsmodus. Jetzt schickt der Server verschlüsselt seine Bereitschaftsnachricht, auch im ChangeCipherSpec Protokoll, damit ist der Handshaketeil abgeschlossen und die Nutzphase beginnt, nachdem noch einmal die Prüfsumme abgeglichen ist, um einen potentiellen Angriff zu entdecken.

**Die Nutzphase**
Die Nutzphase ist die eigentliche Datenübertragungsphase des TLS-Protokolls, da hier die Daten übertragen werden können. Dabei werden sie zuerst von dem Absender mit den errechneten Schlüsseln verschlüsselt, an den Empfänger gesendet, welcher diese wiederum mit den von ihm errechneten Schlüsseln entschlüsselt und somit die Daten sicher und wieder unverschlüsselt hat. Diese Daten können Teil eines HTTP-Protokolls sein, das heißt eine Internetverbindung, wie sie zum Beispiel bei dem Online-Banking benutzt wird. Die Daten können über jedes beliebige TCP/IP-Protokoll geschickt werden, da durch TLS die sichere Verbindung aufgebaut und verwaltet wird.

# Die Implementierung (TLS meets Merkle)

Eine von uns in der Einleitung bereits angesprochene Schwachstelle dieses Verfahrens liegt darin, dass die Signaturen nicht quantencomputersicher sind. Wie sollen also z.B. Alice und Bob sicher gehen, dass sie auch wirklich untereinander kommunizieren und nicht mit einer Dritten wie Eve? Denn wie kann sich Alice sicher sein, dass das Zertifikat mit dem Public Key, mit dem sie später die McEliece-Schlüssel-Signaturen überprüft, auch wirklich Bobs ist und nicht Eves?
Alice muss also sicher sein, dass Bobs Zertifikat nicht korrumpiert ist, und dass kann sie nur, wenn sie ausschließen kann, dass Eve mit ihrem Quantencomputer die Signatur des Zertifikats knacken kann.
Deshalb sind wir in die Rolle der beiden geschlüpft und haben dieses Problem folgendermaßen behoben:
Als Vorlage für unsere TLS-Implementierung haben wir MicroTLS von Erik Tews aus dem JavaCryptoProvider Bouncycastle genommen.
Dieser Client hat für uns zwei wichtige Stellen, an denen wir CMSS einfügen mussten. Die Verschlüsselungsmethoden im TLS-Protokoll werden in speziellen Containern geschickt, den so genannten CipherSuites. Damit unser Client nun CMSS benutzt, mussten wir in der Class TLS-CipherSuiteManager eine eigene BlockCipher erstellen, die an die DHE_RSA Cipher mit AES 256 angelehnt ist, aber statt RSA CMSS zum signieren verwendet.

```
case TLS_MCELIECE_CMSS_WITH_AES_256_CBC_SHA:
return new TlsBlockCipherCipherSuite(new CBCBlockCipher(new
AESFastEngine()), new CBCBlockCipher(new AESFastEngine()), new
SHA1Digest(), new SHA1Digest(), 32, TlsCipherSuite.KE_MCELIECE_CMSS);
```

JuFo.tlsClient.TlsCipherSuiteManager(64-65)

Der verwendete KeyExchange MCELIECE_CMSS wurde von uns ebenso wie TLS_MCELIECE_CMSS_AES_256_CBC_SHA mit einem eigenen Wert versehen, da es keine offizielle Spezifikation dafür gibt.

Die nächste Änderung, die wir im Code vornehmen mussten, waren die richtige Auswertung des Zertifikats, besser gesagt des Public Keys, da dieser für den späteren McEliece-Schlüsselaustausch benötigt wird.

```
Security.addProvider(new FlexiPQCProvider());
Security.addProvider(new FlexiCoreProvider());
```

JuFo.tlsClient.TlsProtocolHandler(327f)

Damit wir die CMSS Algorithmen überhaupt benutzen, und somit den Key instanzieren können, müssen wir den FlexiProvider der TU Darmstadt, in dem die CMSS Methoden/Klassen enthalten sind, laden.
Wir müssen allerdings, bevor wir irgendetwas mit dem Zertifikat machen, erst einmal feststellen, ob das Zertifikat auch überhaupt authentisch ist.
Dazu gibt es normaler Weise komplexe Verfahren, ein Zertifikat zu überprüfen. Der Einfachheit halber haben wir es vorerst durch unsere Klasse MerkleVerifyer im Package JuFo.cert realisiert, die überprüft, ob sie den Issuer des Zertifikats kennt, und ob dessen Signatur stimmt(siehe nächster Codeauszug). Auch darf das Zertifikat nicht selbstsigniert sein. Sollte eine dieser nicht-erlaubten Fälle auftreten, wird ein Error ausgegeben und das Programm beendet sich.

```
if(caCerts[i].getTBSCertificate().getIssuer().equals(issuer)){
      publicKeySpec = new
      X509EncodedKeySpec(caCerts[i].getTBSCertificate().
      getSubjectPublicKeyInfo().getDEREncoded());

      keyFactory = KeyFactory.getInstance("CMSS","FlexiPQC");
      PublicKey caKey = keyFactory.generatePublic(publicKeySpec);
//We only support Merkle-Certificates so let's hope this is one

      sig = Signature.getInstance(
      "CMSSwithSHA1andWinternitzOTSandSHA1PRNG_4","FlexiPQC");
      sig.initVerify(caKey);
      sig.update(cert.getTBSCertificate().getDEREncoded());
      if(!sig.verify(cert.getSignature().getBytes())){
            System.out.println("Wrong Signature!");
            throw new JuFo.tlsClient.TlsRuntimeException("Signature
            wrong");
       }
return true;
```

JuFo.cert.MerkleVerifyer(62ff)

Die Zertifikate werden übrigens mit dem CertificateGenerator in JuFo.cert.CertificateGenerator erstellt. Dieser ist nicht nur eine Art Interface für den X509V3CertificateGenerator aus Bouncycastle, sondern er enthält auch Methoden

zum laden und speichern von Keys und Zertifikaten, auch kann man mit ihm CMSS-Keys erstellen, da er auch eine Art Interface für den Flexiprovider ist.

Hier wird aus der PublicKeySpec des Zertifikats mittels einer CertificateFactory der PublicKey erstellt. Falls es sich nicht um einen CMSS-Key handelt, gibt es einen Error, und die Verbindung bricht mit einem fatal_Error ab.

```
try{
KeySpec publicKeySpec = new
X509EncodedKeySpec(cert.certs[0].getTBSCertificate().getSubjectPublicKey
Info().getDEREncoded());
KeyFactory keyFactory = KeyFactory.getInstance("CMSS","FlexiPQC");
serverKey = keyFactory.generatePublic(publicKeySpec);
}
```

JuFo.tlsClient.TlsProtocolHandler(329ff)

Der Zweite Teil bei dem Merkle zum tragen kommt ist die Signierung und Verifizierung des McEliece-Schlüssels, die anhand der Signierung unseres Servers gezeigt werden soll, den wir eigenständig als Gegenstück zu MicroTLS entwickelt haben. Die oben beschriebenen Vorgänge laufen in ihm genauso ab, wie beim Client.

```
CMSSSignature csig=new
CMSSSignature.CMSSwithSHA1andWinternitzOTSandSHA1PRNG_4();
csig.initSign(serverKey);
mcegen=new McElieceGen();
mcegen.generate();

csig.update(mcegen.getPublicKey().getEncoded());
```

JuFo.tlsServer.TlsServer(594ff)

Hier werden die einzelnen Parameter gehasht und der Hash dann signiert (dies geschieht innerhalb des Flexiproviders). Dieses ByteArray wird dann an den Client gesendet, damit dieser die Echtheit der Parameter überprüfen kann.

Um das Verfahren nun auch einmal praktisch anzuwenden, haben wir einen HTTP-Server und einen kleinen Browser geschrieben, der unser Verfahren zur sicheren Dateiübertragung nutzt.

```
TlsInputStream in=(TlsInputStream)tlsS.getInputStream();
TlsOutputStream out= (TlsOutputStream)tlsS.getOutputStream();
```

JuFo.TLSHttpd(23f)

Hier ist gut zu sehen, dass unsere TLS-Implementierung mit Streams arbeitet, mit denen man ganz normal, wie mit anderen Streams in Java auch, arbeiten kann.

```
if(TLS(url)){
      String address=getAddress(url);
      int port= getPort(url);

      try{
            if(!(this.address.equals(address) &&
      this.port==port)){
                  tlsSocket= new Socket(address,port);
                  tls = new
                  TlsProtocolHandler(tlsSocket
                  .getInputStream(),tlsSocket.getOutputStream()
                  ,"/Developer/");
                  tls.connect(new AlwaysValidVerifyer());
                  in = (TlsInputStream)tls.getInputStream();
                  out=(TlsOutputStream)tls.getOutputStream();
                  this.address=address;
                  this.port=port;
}
```

JuFo.Browser(70ff)

Zuerst überprüft der Browser, ob es sich um eine „https://“-Adresse handelt, und parst diese falls es so ist. Interessant ist hierbei auch noch, dass der Browser in der Lage ist eine TLS-Verbindung aufrechtzuerhalten, da er am Anfang hiermit überprüft, ob er die Verbindung nicht schon aufgebaut hatte.
In diesem Fall benutzt er die alte Verbindung weiter.
Mit dieser Kombination aus Browser und TLSHttpd ist es uns dann gelungen eine HTML-Seite erfolgreich zu übertragen.

# Diskussion

Wie wir jetzt gezeigt haben, funktioniert die mit CMSS signierte und durch McEliece gesicherte Verbindung und wir sind in der Lage diese produktiv einzusetzen. Das heißt eigentlich, dass Alice und Bob sich nun sicher seien können, dass ihre Zertifikate nicht von Eves Quantencomputer korrumpiert werden können und ihre Verbindung auch nicht belauscht werden kann, wodurch sie somit auch nur untereinander kommunizieren. Nur leider gibt es noch ein paar kleinere Probleme.
CMSS-Signaturen sind auf Grund ihrer Komplexität leider noch etwas größer als standardmäßige RSA-Signaturen, weshalb der entstehende Traffic auch ansteigt. Obgleich der weltweiten Zunahme von Highspeed-Internetsanschlüssen, kann dies durchaus noch ein Problem werden, denn es gibt z.B. in der dritten Welt noch viele langsame Anschlüsse, für deren Benutzer das eine unangenehme, längere Wartezeit zur Folge hätte. Dieses Problem könnte man aber durch eine Komprimierung des Streams mit beispielsweise gzip,tar oder bzip2 erreichen, denn Kompression ist in der offiziellen TLS Spezifikation durchaus möglich.
Ein anderes Problem ist unsere Implementierung selbst, da sie auf Java basiert. Alle gängigen Webbrowser wie Firefox oder der InternetExplorer sind in C oder C++ geschrieben und deshalb lässt sich unsere Implementierung gar nicht ohne Weiteres dort einbauen. Aber wenn ein populärer Browser CMSS als Signaturverfahren verwenden würde, wäre es viel einfacher dieses Verfahren populär zu machen, und somit ein höheres Maß an Sicherheit zu garantieren. Ebenso hat auch nicht jeder Java installiert und könnte unter Umständen unseren Browser nicht nutzen. Auch ist Java für eine Anwendung im großen Stil auf Grund seines Bytecodes noch zu langsam, um damit mehrere Tausend Anfragen in der Minute effizient zu verwalten.
Da unsere Implementierung momentan auch nur ein Proof-of-Concept sein soll, ist der Funktionsumfang des TLSHttpd-Servers noch leider sehr beschränkt.

Aber alles in allem haben wir gezeigt, dass eine sichere Authentifizierung auch noch im Post-Quantum-Zeitalter möglich sein wird, ohne z.B. aufwändige Quantenkryptographische Systeme zu benutzen, die für den normalen Benutzer nicht erschwinglich sind. Alice und Bob können, wenn sie das nächste Mal Daten

austauschen wollen, auch wirklich sicher gehen, dass ihre Zertifikate vor Eves Zugriff geschützt sind, und das Phisher nicht mehr so einfach eine Chance haben. Das Potential von solchen Algorithmen ist enorm, denn, wenn man erst anfängt quantencomputersichere Verfahren zu entwickeln, nachdem es bereits Quantencomputer frei verfügbar auf dem Markt gibt, wäre das ein Disaster, da man nicht mehr sicher Online-Banking nutzen oder wichtige Daten übertragen könnte. Aus diesem Grund ist es wichtig frühzeitig solche Algorithmen nicht nur zu entwickeln, sondern auch zu implementieren und ihre Alltagstauglichkeit zu verbessern. Somit ist dieses Projekt ein Schritt in die richtige Richtung, damit gewöhnliche Personen wie Alice und Bob sicher kommunizieren können.

## Ausblick

Um unsere Implementierung alltagstauglicher zu machen, haben wir vor TLS-Merkle in den Firefox oder einen ähnlichen Browser zu integrieren. Ebenso wollen wir unseren TLSHTTP-Server soweit verbessern, dass er auch den POST und SEND Befehl unterstützt, oder das Verfahren in einen Server wie Tomcat einbauen.
Auch ist ein Experiment angedacht, einen eine mit CMSS signierte TLS-Verbindung über das Internet aufzubauen und darüber Versuchsdaten von einem anderen Projekt zu schicken.

## Danksagung

Wir möchten uns ganz herzlich an dieser Stelle bei Prof. Dr. Johannes Buchmann von der TU Darmstadt für seine Hilfe beim Verstehen von CMSS bedanken. Ebenso bedanken möchten wir uns bei Erik Dahmen und Erik Tews von der TU Darmstadt, die unser Projekt betreut und uns immer mit Rat und Tat zur Seite gestanden haben.
Natürlich möchten wir uns auch bei unserem Physiklehrer und Leiter des Weird Science Clubs Dr. Milan Dlabal für seine Unterstützung bedanken.

## Quellenverweise

[JB06] J. Buchmann, L.C. Coronado García, E. Dahmen, M. Döring, E. Klintsevich,CMSS -- An Improved Merkle Signature Scheme, 7th International Conference on Cryptology in India - Indocrypt'06, LNCS 4392, Springer, 2006, pp. 349-363.

[ETews] Erik Tews , Entwicklung von MicroTLS als sichere Ende-zu-Ende-Verbindung über Bluetooth und TCP/IP, Bachelorarbeit TU-Darmstadt

[BC] Bouncycastle Cryptographic API: http://www.bouncycastle.org