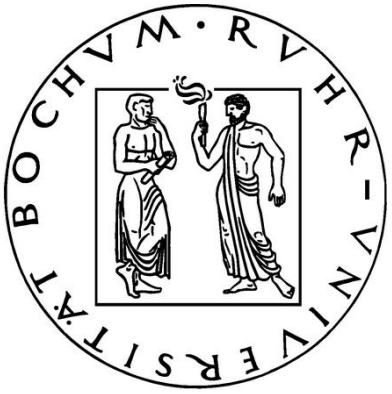

Content Adaptive XML Signatures

Seminararbeit am Lehrstuhl
Netz- und Datensicherheit
Ruhr-Universität Bochum

Jens Riemer

14. Juli 2010

Zusammenfassung

In dieser Arbeit wird ein von Takashi Suzuki et al. entwickeltes Signaturverfahren vorgestellt, welches eine authentifizierte Verteilung adaptiver Multimedia-Inhalte ermöglicht. Das Besondere an diesem Verfahren ist, dass zu bereits signierten Daten durch weitere Parteien Inhalte hinzugefügt oder entfernt werden können, ohne dass hierbei die Originalsignatur zerstört würde. Dadurch kann eine hohe Flexibilität bei der Verteilung authentifizierter, multimedialer Inhalte erreicht werden. Zusätzlich wird ein fast identisches Verfahren von Tan und Deng betrachtet.

Inhaltsverzeichnis

1 Einleitung **4**

2 Technische Grundlagen **5**
2.1 Merkle-Bäume 5
2.1.1 Aufbau 5
2.1.2 Hash-Verifizierung 6
2.2 Trapdoor-Hash-Funktionen 6
2.2.1 Funktionsweise 6
2.2.2 Konstruktion 6

3 Die Architektur **7**

4 Das Signaturschema **8**
4.1 Anforderungen 8
4.2 Platzhalter 9
4.3 Conventional Signatures Scheme 10
4.4 Hash-Sign-Switch 10
4.5 CS vs. HSS 11
4.6 Unerlaubtes Entfernen von Knoten 11

5 Implementierung **11**
5.1 Signierung und Anpassung 12
5.2 Verifikation 14
5.3 Performance 15

6 Sanitizable Signatures **16**
6.1 Aufbau 16
6.2 Realisierung mit Conventional Signatures 17
6.3 Realisierung mit Sanitizable Signatures 17

7 Schlussbemerkungen **18**

1 Einleitung

In den letzten Jahren haben sowohl Anzahl, als auch Leistungsfähigkeit mobiler Endgeräte deutlich zugenommen. Längst schon gehört die Nutzung internetbasierter Dienste für viele Benutzer derartiger Geräte zum Alltag einfach dazu. Dabei spielen heute verstärkt multimediale Inhalte eine Rolle.

Hierbei tut sich für die Anbieter solcher Dienste, wie z.B. für ein Videoportal, die Schwierigkeit auf, die angebotenen Inhalte möglichst unabhängig vom aufrufenden Endgerät in ansprechender Form zu präsentieren. Zu beachten sind hier die üblichen Einschränkungen denen mobile Endgeräte unterliegen (Display-Größe, Übertragungsraten, etc.), als auch eventuell vorhandene Benutzerpräferenzen. Es ist daher zu erwarten, dass die Frage wie diesen Anforderungen Rechnung getragen werden kann, in Zukunft an Bedeutung noch zunehmen wird [2].

Ein besonderes Augenmerk soll hier auf die Schwierigkeit gelegt werden, wie in dem beschriebenen Kontext die Integrität übermittelter Daten gewahrt werden kann. Einerseits erfordert bereits die normale Anpassung der Multimedia-Inhalte an das jeweilige Endgerät Änderungen an den Rohdaten. Darüber hinaus kann eine weitere Anpassung der Daten wünschenswert sein. So könnten beispielsweise andere Unternehmen Werbeverträge mit dem Inhalteanbieter unterhalten, um zielgerichtet in bestimmte Daten ihre Werbeinformationen einfügen zu dürfen. In beiden Fällen würde eine Veränderung der signierten Originaldaten i.d.R. mit einem Ungültigwerden der Signatur einhergehen.

Suzuki et al. begegnen diesem Problem mit dem Einsatz von Trapdoor-Hash-Funktionen in Merkle-Bäumen [3],[4]. Ihr vorgeschlagenes Signaturschema stellt eine recheneffiziente Möglichkeit zur Verfügung, Sekundärinhalte nachträglich in bereits signierte Daten einzufügen. Dies wird mit sogenannten Platzhaltern erreicht. Ebenso können Daten entfernt werden, ohne dass dadurch Signaturen zerstört werden. Die erlaubten Modifikationen können dabei durch die signierende Partei genau gesteuert werden, sodass Änderungen nicht etwa beliebig durchgeführt werden können, sondern nur in dem durch den Signierer vorgegebenen Rahmen. So könnte z.B. ein Anbieter von Multimedia-Inhalten einem Vertragspartner das Einbinden von Werbung gestatten, während Veränderungen durch andere Parteien zu ungültigen Signaturen führen würden.

2 Technische Grundlagen

2.1 Merkle-Bäume

2.1.1 Aufbau

Das vorgestellte Signaturverfahren basiert auf Merkle-Bäumen. Daher sollen diese im Folgenden wenigstens in den Grundzügen erklärt werden. Erfunden wurden diese Hash-Bäume 1979 von Ralph Merkle [4], damals mit der Absicht, eine große Anzahl von Lamport-Signaturen handhaben zu können.

Bei einem Merkle-(Hash)-Baum handelt es sich im Wesentlichen um eine Datenstruktur. Diese enthält in Form eines Baums Informationen über andere Daten (z.B. eine Datei), welche dazu verwendet werden können, diese zu verifizieren. Die Abb. 1 zeigt den Aufbau eines solchen Baums, wie er in der dazugehörigen Patentschrift niedergelegt ist.

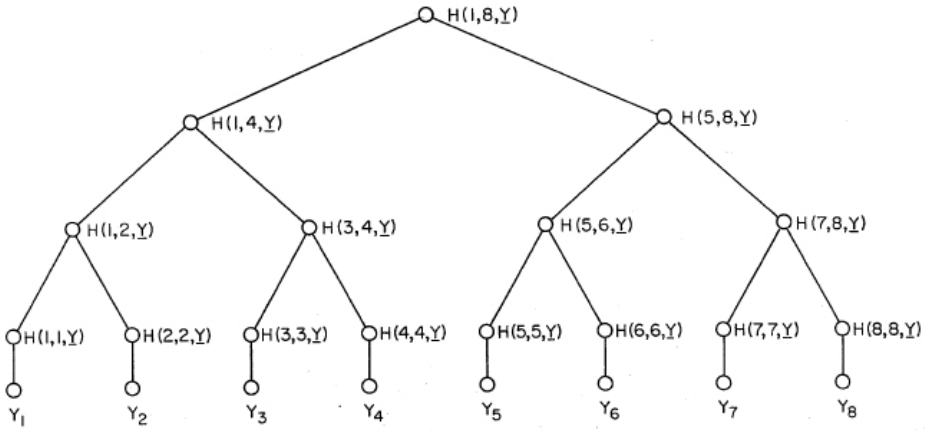

Abbildung 1: Schema eines Merkle-Baums, entnommen aus [4]

Jeder Knoten und jedes Blatt des Baums ist ein Hash-Wert. Dabei kann jeder Knoten, der kein Blatt ist, aus den Hash-Werten seiner Kindknoten berechnet werden. Der Wert an der Wurzel wird *Top Hash* (auch *Root Hash* oder *Master Hash*) genannt. Die Blätter in der untersten Ebene des Baumes enthalten jeweils den Hash-Wert für eine ganz bestimmte Datenmenge. Dies können Teilmengen einer viel umfangreicheren Datei sein.

Ein realistisches Szenario wäre beispielsweise der Austausch einer Datei über ein FileSharing-Netzwerk. Hierbei wird die Datei nicht einfach als Ganzes übertragen, sondern in viele kleinere Einzeldateien „zerschnitten“. Bezogen auf die Abb. 1, würden 8 solcher Einzeldateien vorliegen, im Bild mit $Y_1, \ldots, Y_8$ bezeichnet. Für diese wird jeweils ein Hash-Wert H berechnet, wobei im Folgenden mit $H(i, j, Y)$ der zusammengesetzte Hash-Wert über die Einzeldateien $Y_i, \ldots, Y_j$ gemeint ist.

2.1.2 Hash-Verifizierung

Angenommen die Teildatei Y_5 wird übertragen und soll auf Integrität überprüft werden. Dabei wird vorausgesetzt, dass der Empfänger bereits im Besitz des *Root Hash* ist. Dann benötigt er lediglich $H(6,6,Y)$, $H(7,8,Y)$ und $H(1,4,Y)$, um sich $H(1,8,Y)$ berechnen zu können. Stimmt dieser Wert mit dem bereits bekannten *Root Hash* überein, kann der Empfänger sicher sein, dass Y_5 wirklich Teil der gewünschten Gesamtdatei ist.

2.2 Trapdoor-Hash-Funktionen

2.2.1 Funktionsweise

Ein weiterer wichtiger Bestandteil der hier vorgestellten Verfahren sind die Trapdoor-Hash-Funktionen (TDHFs). Dabei handelt es sich im Wesentlichen um normale Hash-Funktionen, die sowohl deterministisch, als auch kollisionsresistent sind. Damit erfüllen sie die Anforderungen an kryptographisch sichere Hash-Funktionen.

Allerdings besitzen sie einen bedeutsamen Unterschied. Mit Kenntnis eines geheimen Schlüssels, des sogenannten *trapdoor secret*, ist es möglich, effizient Kollisionen zu finden. Normalerweise sollte dies nur dem „Besitzer“ der Funktion möglich sein. Gelangt ein Angreifer jedoch in Besitz dieses Schlüssel, kann er ebenfalls Kollisionen finden. Dagegen ist es mit Hilfe des „öffentlichen Schlüssels“ der TDHF jedem beliebigen Teilnehmer möglich, einen Hash-Wert zu erzeugen, für den der „Besitzer“ nachträglich eine andere Ausgangsnachricht mit identischem Hash-Wert finden kann.

2.2.2 Konstruktion

Für die Konstruktion einer geeigneten TDHF gibt es verschiedene Möglichkeiten. Im Folgenden wird die von Suzuki et al. verwendete Herangehensweise vorgestellt.

- seien p, q Primzahlen, sodass $q|p-1$
- sei g ein Element der Ordnung q in $\mathbb{Z}_p^*$
- wähle Geheimnis $x \xleftarrow{\$} \mathbb{Z}_q^*$ und PubKey $y = g^x \pmod p$
- definiere TDHF als $H_y(m,r) = g^m y^r$ (von jedem berechenbar)
- finde $H(m_1,r_1) = H(m_2,r_2)$ mit $m_1 \neq m_2$ vermöge $r_2 = (m_1 - m_2)x^{-1} + r_1$ (nur mit Kenntnis von x möglich)

Interessant ist hier die Berechnung von r_2. Diese erfordert lediglich eine einzige modulare Multiplikation. Wie noch zu zeigen sein wird, ist es v.a. dieser Tatsache geschuldet, dass durch Verwendung von TDHFs ein deutlicher Performance-Gewinn in den beschriebenen Verfahren erzielt wird.

3 Die Architektur

Suzuki et al. legen bei der Beschreibung ihres Verfahrens ein Multimedia-Streaming-Netzwerk zugrunde, in welchem vor Übertragung der eigentlichen Multimedia-Inhalte zunächst Meta-Informationen gesendet werden, die darüber Aufschluss geben, wie diese Komponenten handzuhaben sind. Diese Unterteilung in Meta-Daten und Inhalte muss bei der Betrachtung berücksichtigt werden.

Eine Anpassung an das jeweilige Endgerät kann in solch einem System auf beiden Ebenen geschehen. So kann auf der Meta-Ebene die Zusammenstellung von Audio- und Videokomponenten gesteuert werden. Beispielsweise könnte je nach Einstellung für einen Benutzer ein zusätzlicher Audiokommentar in einer Filmszene hörbar sein, während ein anderer Benutzer diesen bewusst unterdrücken kann. Die Anpassung auf Media-Ebene beschäftigt sich hingegen eher mit Komprimierungs- und Codierungstechniken.

In Abb. 2 ist das verwendete Netzwerk dargestellt. In erster Ebene befindet sich der Inhalteanbieter (auch *Tier-1 Provider* genannt, im Folgenden kurz als T_1 bezeichnet). Neben diesem gibt es einen Tier-2 Provider (T_2), z.B. einen Werbevertragspartner. Als dritter wichtiger Teilnehmer des Netzwerks ist schließlich der Endbenutzer (auch *Content User* genannt) anzuführen. Er kann Kunde von T_1 oder T_2 sein.

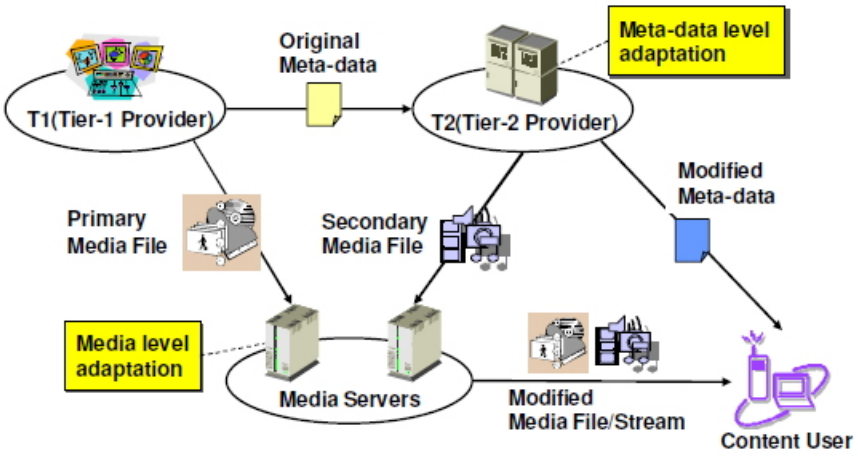

Abbildung 2: Das zugrundeliegende Netzwerk, wie in [1] dargestellt

Die auszuliefernden Media-Dateien werden auf speziell dafür bereitgestellten Media-Servern gelagert. Diese können, müssen jedoch nicht im Besitz des Inhalteanbieters sein. Die Meta-Daten werden hingegen von T_2 angepasst und direkt an den *Content User* weitergegeben. Die einzelnen Media-Dateien kann man sich wie Bausteine vorstellen, aus denen T_2 das zusammensetzt, was der Endbenutzer letztlich zu sehen, bzw. zu hören bekommt. Dabei können sowohl einzelne Komponenten hinzugefügt, als auch entfernt werden. Dies alles geschieht ausschließlich über eine Veränderung der Meta-Informationen. Es ist also nicht notwendig, dass T_2 auf die von T_1 zur Verfügung gestellten Media-Daten Zugriff erhält.

Ein denkbares Beispiel für einen *Tier-1-Provider* wäre ein Videoportal, welches auf eigenen Media-Servern eine große Anzahl von Filmen vorhält und diese anderen Firmen kostenpflichtig zugänglich macht. Der *Tier-2-Provider* könnte in diesem Fall ein Firmenkunde von T_1 sein und als Video-On-Demand-Anbieter auftreten, der jedoch selbst keine eigenen Media-Server unterhält. Der *Content User* wäre nun Kunde von T_2. Wählt er einen kostengünstigen Tarif, könnte T_2 ihm dafür Werbung in die ansonsten ganz normal ablaufenden Filme einblenden. Bei Wahl eines teureren Tarifs würden die Werbeinformationen hingegen nicht erscheinen.

Zwischen den Teilnehmern des Netzwerks kann ein schwaches Vertrauensmodell angenommen werden. Das bedeutet beispielsweise, dass T_2 versuchen könnte, mehr Informationen in die Daten von T_1 einzufügen, als ihm dies vertragsmäßig zustände. Ebenso könnte ein anderer *Tier-2* oder der *Content User* selbst bemüht sein, die durch T_2 hinzugefügten/entfernten Informationen zu unterdrücken, bzw. diese trotzdem einzusehen. Denkbar wäre hier ein lokal installierter Werbeblocker, der die Werbeinformationen herausfiltern könnte.

Zwingend notwendig ist daher zumindest die Annahme einer vertrauenswürdigen Software zur Wiedergabe der übertragenen Inhalte beim Endbenutzer. Das bedeutet, dass sich dieser Media-Player verlässlich an die in den Meta-Daten spezifizierten Regeln halten muss. Diese Software sichert also einerseits T_2 gegenüber dem Endbenutzer ab, da sichergestellt werden kann, dass jener exakt die beabsichtigten Inhalte erhält. Andererseits können so auch T_1 und der *Content User* geschützt werden, da eine nicht erlaubte Veränderung der Daten durch T_2 auffallen würde.

4 Das Signaturschema

4.1 Anforderungen

Aus der im vorangehenden Kapitel gezeigten Netzwerkarchitektur ergeben sich drei Anforderungen an das vorgestellte Signaturschema.

1. T_2 kann gemäß den durch T_1 vorgegebenen Regeln Elemente in die Meta-Daten einfügen, bzw. aus diesen entfernen. Regelverletzungen werden erkannt.

2. T_2 kann Elemente nur an den durch T_1 festgelegten Positionen einfügen. Einfügungen an nicht genehmigten Stellen werden erkannt.

3. Nachdem sich T_2 auf ein einzufügendes Element festgelegt hat, kann dieses nicht unerkannt verändert oder gelöscht werden, d.h. die Signatur wird ungültig.

Anmerkung: T_2 legt sich nicht auf einen genauen Inhalt fest, sondern auf einen Platzhalter, in welchen anschließend Inhalte eingefügt werden können.

4.2 Platzhalter

Um die Idee des Signaturschemas umsetzen zu können, werden zusätzliche Platzhalter-Elemente (*placeholder*) eingeführt. Diese spezifizieren eine Position in den Meta-Daten, in welche eine Entität zusätzliche Meta-Informationen einfügen darf. Der Platzhalter enthält zudem genaue Informationen, um diese Entität exakt zu identifizieren (hier wäre dies T_2).

Das Einfügen der Meta-Daten verläuft dann immer gleich und ist in Abb. 3 skizziert. Zunächst stellt T_2 eine Platzhalteranfrage an T_1. Daraufhin konstruiert T_1 aus den Hash-Werten der Meta-Daten und dem Platzhalter den Merkle-Baum und signiert den Root-Hash. Diese Informationen werden an T_2 zurückgegeben, welcher nun die gewünschten Elemente in den Baum einfügt. Anschließend erhält der Endbenutzer von T_2 die von T_1 erstellte Signatur, die Meta-Daten sowie die noch fehlenden Informationen, um die Signatur verifizieren zu können.

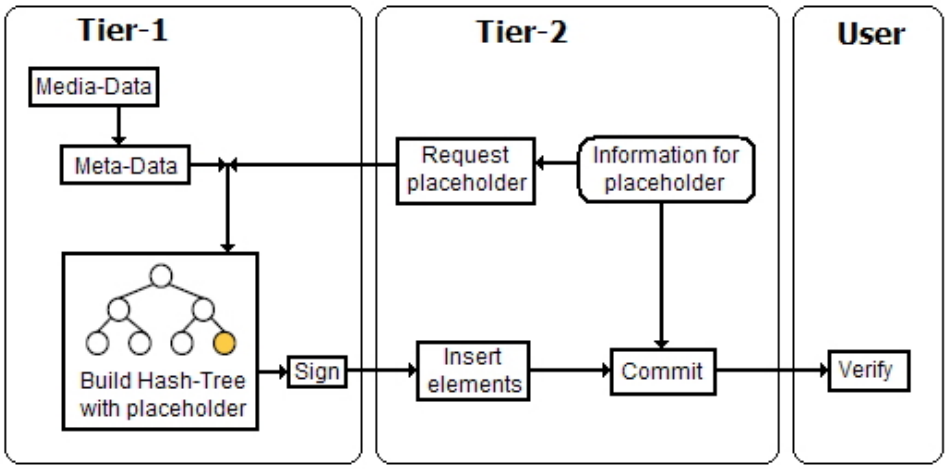

Abbildung 3: Anfordern und Einbinden des Platzhalters nach [1]

Spätestens an dieser Stelle muss man die Frage stellen, wie denn nun der Platzhalter im Nachhinein verändert werden kann, ohne dass dadurch die Signatur zerstört würde. Suzuki et al. ziehen dafür zwei Möglichkeiten in Betracht: Conventional Signature (CS) und Hash-Sign-Switch (HSS) [5].

4.3 Conventional Signatures Scheme

Das CS Schema ist recht simpel und schnell erklärt. Vorausgesetzt wird, dass alle Teilnehmer des Netzwerks ein Public-Key-Verfahren einsetzen. Bei der Konstruktion des Merkle-Baums schreibt T_1 in den Platzhalter den öffentlichen Schlüssel pk_{T2} von T_2 und signiert den Root-Hash. T_2 hängt an die übermittelten Daten die zusätzlichen Informationen einfach hintan und signiert diese separat. Der *Content User* überprüft dann einfach beide Signaturen auf Korrektheit und kann so sicherstellen, dass die zusätzlichen Daten durch T_1 genehmigt waren. Abb. 4 veranschaulicht diesen Zusammenhang.

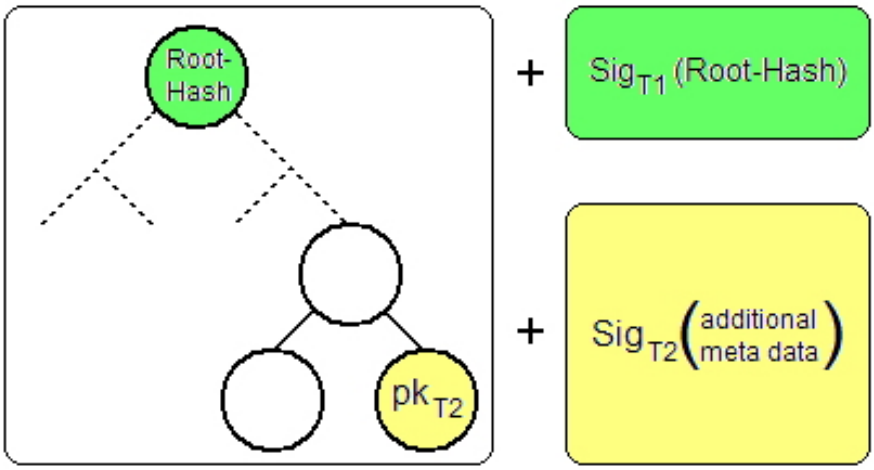

Abbildung 4: Signaturen beim CS Schema

4.4 Hash-Sign-Switch

Der HSS-Ansatz verwendet für die Ergänzung der Platzhalter kryptographisch sichere Trapdoor-Hash-Funktionen. Wie bereits in 2.2.1 erläutert, versteht man unter einer Trapdoor-Hash-Funktion eine Hash-Funktion, für die ein Angreifer nur mit vernachlässigbarer Wahrscheinlichkeit eine Kollision finden kann. Allerdings kennt der legitime Benutzer einen effizienten Algorithmus, der eben solch eine Kollision relativ leicht herbeiführt.

Im vorliegenden Fall erzeugt sich T_2 mittels dieses Algorithmus, passend zu den später einzufügenden Daten, einen (zufälligen) Wert, der den identi-

schen Hash-Wert aufweist; er findet also eine Kollision. Dieser zufällige Wert wird an T_1 geschickt, und von diesem in den Baum integriert und signiert. Im Anschluss daran ist T_2 aufgrund der identischen Hash-Werte dazu in der Lage, die fehlenden Meta-Informationen zu ergänzen, ohne die Signatur von T_1 ungültig werden zu lassen.

4.5 CS vs. HSS

Beim Vergleich der beiden Verfahren erweist sich HSS gegenüber CS als deutlich effizienter. Legt man für die Trapdoor-Hash-Funktion die Diskrete-Logarithmus-Annahme zugrunde, so benötigt HSS lediglich eine Modularmultiplikation. Dem gegenüber kommt CS bei Verwendung von 1024-bit RSA im Schnitt auf 1500 Multiplikationen.

Ein weiterer Vorteil von HSS ist, dass T_2 seine Daten einfach an die Originalsignatur anhängen kann und den Hash-Baum nicht noch einmal neu erstellen muss. Der *Content User* muss hingegen einfach nur den Baum rekonstruieren, um die übertragenen Daten auf nicht authentifizierte Änderungen hin zu überprüfen.

4.6 Unerlaubtes Entfernen von Knoten

Für die Provider ist von besonderem Interesse, dass der Endbenutzer genau die Daten erhält, die für ihn vorgesehen sind. Das heißt insbesondere, dass er nicht in Lage sein soll, z.B. Werbung herauszufiltern. Das Signaturschema muss dieser Tatsache Rechnung tragen.

Das unerlaubte Entfernen von Elementen kann auf zwei Arten verhindert werden. Zum einen bieten sich *aggregate signatures* an, welche sich besonders durch ihre Kompaktheit auszeichnen. Dieser Ansatz war jedoch bei Veröffentlichung des Beitrags von Suzuki et al. noch nicht implementiert. Umgesetzt wurde hingegen der zweite Ansatz, bei dem T_2 einfach jeden Platzhalter signiert, ungeachtet dessen, ob später tatsächlich Inhalte dort eingefügt werden sollen oder nicht. Das bedeutet, dass jeder Platzhalter ohne gültige Signatur von T_2 durch die Media-Player-Software beim *Content User* erkannt wird und diese entsprechend darauf reagieren kann (Fehlermeldung; Verweigern des Abspielens der sonstigen Inhalte). Diese Möglichkeit wurde bereits implementiert und wird im folgenden Kapitel erläutert.

5 Implementierung

Suzuki et al. stellen in ihrem Artikel eine beispielhafte Implementierung für das oben beschriebene Signaturschema vor. Bei diesem verwenden sie SMIL für die Beschreibung der Meta-Daten. SMIL (*Synchronized Multimedia Integration Language*) ist ein vom W3C entwickelter Standard, der als Auszeichnungssprache für zeitsynchronisierte, multimediale Inhalte dient [6].

Die Sprache ist XML-basiert, unterstützt Java und stellt eine einfache Möglichkeit zur Einbindung und Steuerung von Audio-, Video-, Text- und Grafikdaten in Webseiten zur Verfügung.

Zum Schreiben der Nutzungsregeln in das SMIL-Dokument dient hier XACML (*eXtensible Access Control Markup Language*). Dabei handelt es sich um ein XML-Schema, das zur Darstellung und Verarbeitung von Authorisierungsregeln genutzt werden kann. Die Sprache wurde von OASIS standardisiert [7].

Darüber hinaus wurde XML-DSIG um die Möglichkeit erweitert, Merkle-Bäume mit Platzhaltern verarbeiten zu können.

5.1 Signierung und Anpassung

In Abb. 5 ist ein einfaches SMIL-Dokument dargestellt, welches signiert werden soll. Es bindet eine Video- und eine Audiodatei ein und enthält zusätzlich einen Platzhalter für ein später zu ergänzendes Element (farbig markiert). T_2 schickt dieses als SOAP-Nachricht an T_1 und stellt somit seine Platzhalteranfrage.

```
<?xml version="1.0"?>
<smil>
  <head/>
  <body>
    <seq>
      <par>
        <video src="rtsp://tyer-1/video1.rm"/>
        <video src="rtsp://tyer-1/music1.rm"/>
      </par>
      <par>
        <video phid="1"/>
      </par>
    </seq>
  </body>
</smil>
```

Abbildung 5: Original-SMIL-Dokument mit Platzhalter (nach [1], Abb. 3)

T_1 reagiert auf diese Anfrage, indem er die Parameter aus der Nachricht in die Felder ⟨PublicValue⟩ und ⟨TrapdoorHashValue⟩ kopiert. Anschließend berechnet er die Signatur und schickt das Ergebnis zurück an T_2. In Abb. 6, die das Dokument nach dem Signieren zeigt, fallen die beiden Elemente ⟨HashTreeConstruction⟩ und ⟨TrapdoorHashMethod⟩ auf. Diese stellen Erweiterungen von XML-DSIG dar. Ersteres Element ist notwendig, um die Merkle-Bäume integrieren zu können. Das Zweite dient dazu, Parameter der Trapdoor-Hash-Funktion zu übergeben.

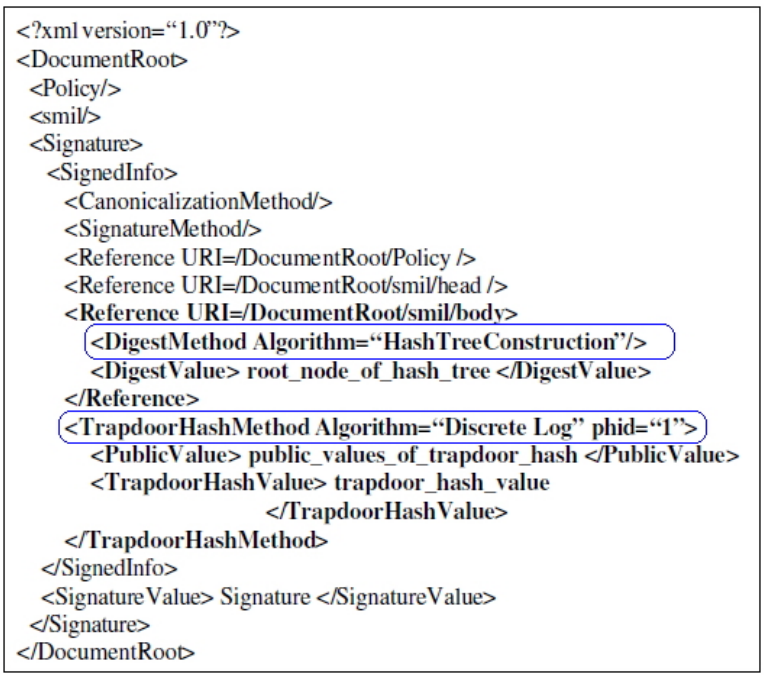

Abbildung 6: Dokument nach dem Signieren (nach [1], Abb. 3)

Nach dem Signieren kann T_2 nun, entsprechend den von T_1 gemachten Vorgaben, das signierte SMIL-Dokument verändern. In Abb. 7 wird ein Element mit Hilfe des Attributwerts „delete“ entfernt und ein weiteres Element mittels „add“ hinzugefügt. Nach der Transformation hat das Dokument die folgende Form:

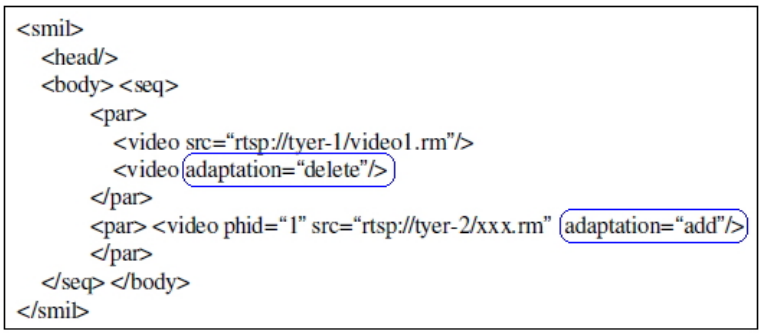

Abbildung 7: Dokument nach der Transformation (nach [1], Abb. 3)

Die transformierten Meta-Daten werden nun nochmals gegen die im Policy-Element hinterlegten Vorgaben geprüft. Für jede erlaubte Änderung wird ein Commitment berechnet und unterhalb des Elements ⟨AdditiveSignature⟩ eingefügt. Wie in der folgenden Abbildung zu sehen ist, wird wiederum das

Attribut *phid* eingesetzt, um die jeweilige Änderung zu referenzieren. Das nun fertige Dokument wird direkt an den Endbenutzer geschickt.

```
<Signature>
  <SignedInfo>
    <CanonicalizationMethod/>
    <SignatureMethod/>
    <Reference URI=/DocumentRoot/Policy />
    <Reference URI=/DocumentRoot/smil/head />
    <Reference URI=/DocumentRoot/smil/body />
    <TrapdoorHashMethod Algorithm=
                        "Discrete Log" phid="1"/>
  </SignedInfo>
  <SignatureValue/>
  <AdditiveSignature phid="1">
    <CommitmentValue>
      Commitment_Value_of_TrapdoorHash
    </CommitmentValue>
  </AdditiveSignature>
</Signature>
```

Abbildung 8: Dokument nach dem Commitment (nach [1], Abb. 3)

5.2 Verifikation

Das Verifikationsmodul beim Endbenutzer wurde als HTTP-Proxy implementiert. Dieser evaluiert die signierten Daten und gibt bei Erfolg ein SMIL-Dokument an den vom Benutzer eingesetzten SMIL-fähigen Media-Player aus. Im Versuchsaufbau wurde hierfür der RealOne Player gewählt [8]. Die Verifikation unterteilt sich in drei Schritte: Validierung der Signatur, Policy Compliance Check und Transformation. Diese sind im Folgenden ausführlicher beschrieben.

1. *Validierung der Signatur:*

   - Zu diesem Zweck wird der Hash-Baum von den Blättern aus zur Wurzel rekonstruiert. Enthält ein Element das Attribut „add“, wird der dazugehörige Hash-Wert aus ⟨TrapdoorHashMethod⟩ ausgelesen. Die Referenzierung erfolgt über das Attribut ⟨phid⟩.
   - Der Commitment-Wert für hinzugefügte Elemente wird separat ermittelt. Dafür wird der Trapdoor-Hash-Wert aus den zugefügten Daten und dem Commitment berechnet und mit dem Wert von ⟨TrapdoorHashValue⟩ im ⟨Signature⟩-Element verglichen. Falls die Werte übereinstimmen, wird mit dem nächsten Schritt fortgefahren.

2. *Policy Compliance Check:* Hier wird nochmals überprüft, ob die von T_2 vorgenommenen Änderungen in dieser Form genehmigt sind. Bei erfolgreicher Überprüfung, wird das Dokument transformiert.

3. *Transformation:* Das signierte Dokument wird schließlich in das standardisierte SMIL-Format überführt. Während dieses Vorgangs werden systemspezifische Elemente und Attribute gelöscht. Anschließend wird das Ergebnis an den Media-Player weitergegeben und kann vom Endbenutzer betrachtet, bzw. gehört werden.

Stellt das Verifikationsmodul bei einem dieser Schritte einen Fehler fest, wird der Vorgang sofort abgebrochen und eine Fehlermeldung an den Media-Player gesendet.

5.3 Performance

Das von Suzuki et al. eingesetzte Testsystem verwendet für T_2 einen Pentium 4 mit 3,06 GHz und 1 GB Arbeitsspeicher unter Redhat Linux 2.4.20. Das Modul für den *Content User* wird auf einer etwas langsameren Maschine mit 866 MHz Pentium 3 und 512 MB RAM unter Windows XP realisiert. Für beide Schemata, CS und HSS wurde 1024 Bit RSA-SHA1 mit XML-DSIG gewählt und für HSS eine Trapdoor-Hash-Funktion mit 1024 Bit Modulus. Das SMIL-Dokument umfasst 5 Elemente.

In Abb. 9 sind die Messergebnisse für T_2 und den Endbenutzer gegenübergestellt. Ohne im Detail auf die einzelnen Zeiten einzugehen, fallen sofort drei Fakten auf.

1. Der Rechenaufwand für den *Content User* ist durchgängig höher, als für T_2.

2. HSS zeigt bei T_2 eine signifikant bessere Performance gegenüber CS, während sich der Aufwand für den *Content User* leicht erhöht.

3. Der Aufwand für die Signaturverifizierung ist für den *Content User* in etwa konstant.

Die Verwendung von HSS bietet für T_2 einen Performance-Gewinn von 95% im Vergleich zu CS und ist daher zu bevorzugen, da T_2 in der Praxis die Anfragen vieler Kunden managen müsste. Die leichte Aufwandserhöhung für den Endbenutzer ist vertretbar, da sich für ihn nur eine unerheblich längere Wartezeit ergibt. Der Geschwindigkeitsgewinn für T_2 hat in diesem Falle Vorrang.

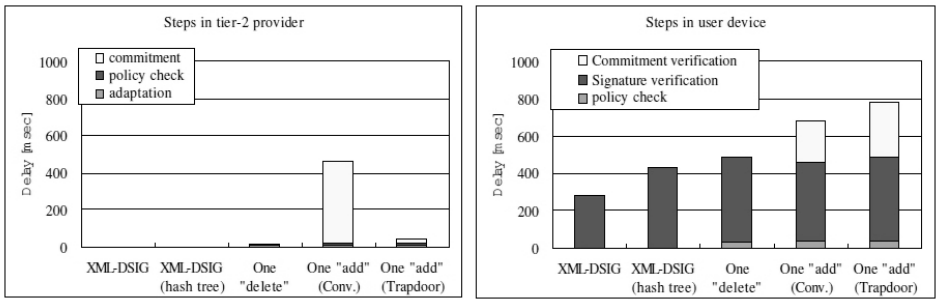

Abbildung 9: Verarbeitungverzögerung in Millisekunden (aus [1])

6 Sanitizable Signatures

Die bisher behandelten modifizierbaren Signaturen sollen nun ergänzend in einem anderen, wenngleich sehr ähnlichem Kontext betrachtet werden. Dazu wird eine von Tan und Deng veröffentlichte Arbeit herangezogen, die sich mit der Anwendung derartiger Signaturen in web-service-basierten Geschäftsprozessen auseinandersetzt [9].

6.1 Aufbau

Tan und Deng verwenden bei ihren Betrachtungen folgende Teilnehmer, die miteinander interagieren müssen und dabei auf gesicherte (hier im Sinne von authentifizierte) Kommunikation angewiesen sind. Folgende Parteien sind in den Prozess eingebunden:

1. Distributor: ein Versandhandel, der Produkte an den Retailer verkauft
2. Deliverer: übernimmt die komplette Logistik für (1)
3. Manufacturer: beliefert (1) mit den benötigten Produkten
4. Retailer: Geschäftskunde, der Produkte von (1) kauft

Der Ablauf stellt sich nun wie folgt dar. Zunächst gibt der *Retailer* beim *Distributor* eine Bestellung in Auftrag. Dieser hat die angeforderten Produkte ursprünglich vom *Manufacturer* erhalten und nun zwischengelagert. Der *Distributor* nimmt die Auslieferung der bestellten Ware nicht selbst vor, sondern übergibt diese an den Geschäftspartner *Deliverer*, da die Logistik vollständig ausgelagert wurde. Der Kunde (*Retailer*) erhält nun über den *Deliverer* seine Bestellung.

Parallel zu diesem Vorgang der physischen Auslieferung, verschickt der *Distributor* ein elektronisches Dokument, welches verschiedenste Informationen zum Bestellvorgang enthält. Er selbst versieht dieses Dokument beispielsweise mit Kundeninformationen und dem geforderten Gesamtpreis der

Waren. Das Dokument wird an den *Deliverer* übergeben, welcher zum reinen Warenwert die Transportkosten aufschlägt. Dies geschieht in Absprache mit dem *Distributor*. Das Dokument wird anschließend an den *Manufacturer* weitergereicht, der es um ergänzende Produktinformationen, wie z.B. Betriebsanleitung oder Garantiehinweise ergänzt. Abschließend geht das mehrfach veränderte Dokument an den *Retailer*. Der beschriebene Aufbau ist der Anschaulichkeit halber in Abb. 10 wiedergegeben.

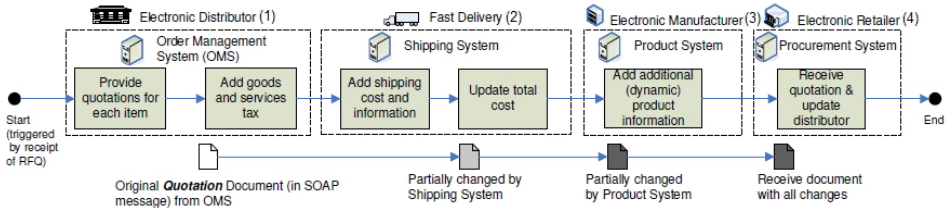

Abbildung 10: Aufbau der Geschäftsprozesse (aus [9])

6.2 Realisierung mit Conventional Signatures

Entlang der gesamten Wegstrecke, die das vom *Distributor* verschickte Dokument nimmt, ist es von großem Interesse, dass die Veränderungen authentifiziert erfolgen. Dies lässt sich theoretisch auch mit „normalen“ XML-Signaturen realisieren.

Dafür signiert jeder Teilnehmer die von ihm modifizierten Bereiche und hängt seine Signatur an die übermittelten Daten mit an. Daraus ergeben sich jedoch zwei Nachteile. Zum einen wird für jeden Teilnehmer, der Daten verändern möchte, eine zusätzliche Signatur erzeugt. Dies führt dazu, dass der *Retailer* bei i die Daten abändernden Teilnehmern schlussendlich i Signaturen verifizieren muss. Zum anderen kann es sein, dass bestimmte Bereiche von mehreren Teilnehmern verändert werden müssen, zum z.B. im beschriebenen Kontext die Gesamtkosten.

6.3 Realisierung mit Sanitizable Signatures

Die Verwendung von Sanitizable Signatures gleicht die erwähnten Nachteile konventioneller Signaturen vollständig aus. Entgegen einem wiederholten Anhängen zusätzlicher Signaturen genügt es, wenn der *Distributor* allen Parteien, die Daten verändern müssen, vorberechnete Trapdoor-Werte (*random coins* genannt) übergibt, mit deren Hilfe sie die Signaturen „updaten“ können. Die *random coins* entsprechen hierbei dem Wert r_2 aus 2.2.2.

7 Schlussbemerkungen

Modifizierbare Signaturen, realisiert durch Trapdoor-Hash-Funktionen, stellen eine wertvolle Bereicherung für web-basierte, verteilte Dienst dar. In den betrachteten Szenarien konnte gezeigt werden, das diese Gruppe von Signaturen die Dezentralisierung von Geschäftsprozessen unterstützt, indem notwendige Änderungen flexibel und authentifiziert an den signierten Daten vorgenommen werden können.

Der größte Vorteil gegenüber konventionellen Signaturen liegt allerdings in der erheblich verbesserten Performance. Diese kommt v.a. bei Updates bereits bestehender Signaturen zum Tragen. Da diese höchstwahrscheinlich auf zentralen Geschäftsrechnern ausgeführt werden, welche i.d.R viele Anfragen zu verarbeiten haben, ist dies besonders vorteilhaft.

Literatur

[1] **Takashi Suzuki et al.:** A system for end-to-end authentication of adaptive multimedia content. *IFIP International Federation for Information Processing*, 175/2005, Springer-Verlag. ISBN 978-0-387-24485-3

[2] **M. Etoh, S. Sekiguchi:** MPEG-7 enabled digest video streaming over G3 mobile network.

[3] **R. Merkle:** Protocols for Public Key Cryptosystems. *Proc. of the IEEE Symposium on Security and Privacy*, S. 122-134, 1980.

[4] **United States Patent 4,309,569:** Method of Providing Digital Signatures, 1982.

[5] **A. Shamir, Y. Tauman:** Improved Online/Offline Signature Schemes. *Proc. of Crypto 2001*, S. 355-367

[6] **W3C:** Synchronized Multimedia, `www.w3.org/AudioVideo` (Stand: 2010-05-30)

[7] **OASIS Committee:** eXtensible Access Control Markup Language, `www.w3.org/AudioVideo` (Stand: 2010-06-01)

[8] **RealOne Player:** `http://de.real.com/realplayer` (Stand: 2010-06-01)

[9] **Kar Way Tan, Robert H. Deng:** Applying Sanitizable Signature to Web-Service-Enabled Business Processes: Going Beyond Integrity Protection, *icws, pp.67-74, 2009 IEEE International Conference on Web Services*, 2009