Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

Fraunhofer Institut für
Graphische Datenverarbeitung
Darmstadt

Entwicklung eines Systems zur dezentralen Online-Ticketerstellung und -kontrolle

Diplomarbeit

Leipzig, November 1998

vorgelegt von
Sven Türpe

Vorwort

»If you can't explain it in plain English, you probably don't understand it yourself.«
Richard Feynman

Formeln werden Sie in dieser Arbeit nur wenige finden, Sätze und Beweise überhaupt nicht. Den Durst des Lesers nach geheimnisvollen, streng formalen Zeichenfolgen sollen statt dessen Quelltexte im Anhang stillen. Warum?

Ich habe keine aufregenden neuen Algorithmen gesucht und gefunden, sondern vorhandene zusammengesetzt.

Die Aufgabe umfaßt die Implementierung eines Prototypen. Die Quelltexte sind einfach da. Wo sonst sollte man sie festhalten als auf bedrucktem Papier? Papier ist nach wie vor der einzige zugleich handliche und haltbare Datenträger. Es erfordert keine besondere Dekodiervorschrift und ist selbst ohne Strom bei Kerzenschein noch lesbar.

Programme werden nicht nur für den Compiler geschrieben. Programmiersprachen sind die formalen Ausdrucksmittel des Informatikers. Zum Schreiben gehört das Lesen. Nur wer wieder und wieder liest, kann selbst zum Autor werden. Ich habe viel aus anderer Leute Quelltext gelernt.

Des Programmierens muß man sich nicht schämen. Gelegentlich treffe ich auf Informatiker, die von einer eigentümlichen Phobie befallen sind. Sie zieren sich, Programmtext anzufassen, als sei das etwas Schmutziges. Allenfalls bei leerer Kasse quälen sie sich mit Editoren und Compilern, aber eigentlich haben sie das nicht nötig. Der Informatiker programmiere nicht, meinen sie, denn dafür habe er Knechte. Hoffentlich gibt es keine Mediziner mit ähnlicher Einstellung zu ihrem Fach.

Und schließlich geht es in der Arbeit vor allem um Sicherheit. Sie lässt sich nur dann in Formalitäten pressen, wenn man sie in einer idealen Welt betrachtet, in der Menschen nicht vorkommen. Zwar spielt sich Kryptographie in der Literatur oft zwischen Alice und Bob ab, doch im richtigen Leben ist Bob ein freundlicher Geldautomat, Alice seine Kundin – und Bösewicht Mallory ein Handtaschenräuber, der Alice beim Eintippen ihrer Geheimnummer über die Schulter schaut. Und schon ist das Konto leer, trotz $M = D_K(E_K(M))$.

Um am Ende den Bogen zum einleitenden Zitat zu spannen: »plain English« oder, wie hier, nacktes Deutsch, ist nicht nur Mittel zur Selbstkontrolle, sondern auch eine Pflicht, die gern vernachlässigt wird. Was mögen Eltern denken, die als Ergebnis eines Vierteljahrhunderts des Durchfütterns nichts als scheinbar sinnlose Zeichenfolgen sehen?

Inhalt

1 Worum es geht

Anforderungsprofil Was das ist bzw. sein könnte, erklärt unnachahmlich die *Frankfurter Rundschau* in ihrer Beilage »Wer will was werden«: »Das Anforderungsprofil eines Berufes besteht aus mehreren Einzelmerkmalen. Diese sind untereinander nicht starr, sondern recht flexibel. Manche können sich gegenseitig ersetzen oder ausgleichen (Kompensation). So ist fehlende Farbsicherheit teilweise durch logisches Denken ausgleichbar (Bei der Verkehrsampel ist Rot immer unten!)«
(Eckhard Henscheid: Dummdeutsch)

1.1 Reality Check

Eintrittskarten Eine Eintrittskarte des Kabaretts *Leipziger Pfeffermühle* berechtigt zum Besuch einer bestimmten Vorstellung. Dafür sind Datum, Uhrzeit und Veranstaltungsort festgelegt. Für jeden Besucher ist ein Sitzplatz reserviert; Reihe und Platznummer werden auf der Karte angegeben. Die einzelnen Plätze sind unterschiedlichen Preisstufen zugeordnet. Beim Einlaß zur Veranstaltung werden die Karten kontrolliert und durch Abreißen eines vorbereiteten Abschnittes entwertet. Studenten und andere mutmaßlich Mittellose können gegen Vorlage eines Aussweises ermäßigte Eintrittskarten erwerben.

Genauso verhält es sich bei einer Eintrittskarte zu einem Konzert im Gewandhaus zu Leipzig. Hier gilt die Karte jedoch zusätzlich vor und nach dem Konzert als Fahrschein in öffentlichen Verkehrsmitteln.

Zum jährlich stattfindenden Chaos Communication Congress gibt es Tages- und Dauerkarten. Sie werden mit einem Photo des Besuchers personalisiert und müssen am Eingang vorgezeigt sowie während des gesamten Besuchs sichtbar getragen werden.

Fahrscheine im Nahverkehr Eine Monatskarte der Leipziger Verkehrsbetriebe berechtigt den Inhaber, beliebig viele Fahrten innerhalb eines geographischen Gebietes zu unternehmen. Dazu kann er alle Verkehrsmittel der LVB nutzen. Die Karte besteht aus zwei Teilen, einem Kundenpaß und der eigentlichen Fahrkarte. Der Kundenpaß bestimmt das Gebiet, in dem die Karte gültig ist. Er enthält außerdem Namen und Adresse des Inhabers sowie eine Nummer. Der Kundenpaß wird kostenlos und ohne Formalitäten ausgestellt; er ist kostenlos und ohne Zeitbegrenzung gültig.

Die Karte ist an Automaten und in Verkaufsstellen erhältlich. Sie gilt vom Tage der ersten Benutzung an bis zum Monatsende. Zur ersten Benutzung muß die Fahrkarte durch einen Stempelaufdruck entwertet werden. Entwerter befinden sich in allen Fahrzeugen der LVB; sie werden ohnehin für Einzelfahrscheine benötigt. Der Inhaber trägt vor der ersten

Abb. 1.1: *Eintrittskarte*

Benutzung seine Kundenpaß-Nummer in ein Feld der Fahrkarte ein. So wird die Karte an den Kundenpaß und damit an den Inhaber gebunden, sie kann also nicht nacheinander von mehreren Personen verwendet werden.

Im Gegensatz dazu kann die MobiCard des Verkehrsverbundes Großraum Nürnberg nacheinander von mehreren Personen benutzt werden. Sie gilt 7 oder 31 Tage von einem beliebigen Datum an, das beim Kauf festgelegt und aufgedruckt wird.

Bahnfahrkarten Eine Fahrkarte für eine Fahrt von Leipzig nach Darmstadt gilt vier Tage lang. Der erste Geltungstag wird beim Kauf festgelegt. Innerhalb dieser vier Tage kann der Reisende die Fahrt beliebig oft unterbrechen. Bestimmte Umwege sind erlaubt. So ist beispielsweise die Fahrt über Halle/Saale möglich. Intercity-Züge kann der Reisende nur benutzen, wenn er zum Fahrschein eine Zuschlagkarte gelöst hat. Für ICE-Züge gilt ein besonderer Fahrpreis. Beim Kauf des Fahrscheins wird festgelegt, auf welchen Teilstrecken diese Züge benutzt werden können. Ein Fahrschein mit ICE-Berechtigung für eine beliebig kurze Teilstrecke kann auf der gesamten Strecke ohne Zuschlag in Intercity-Zügen benutzt werden.

Fahrscheine werden für die erste oder für die zweite Wagenklasse ausgestellt. Mit einer Fahrkarte für die zweite Klasse dürfen die Wagen der ersten Klasse nicht benutzt werden. Eine Sitzplatzreservierung gehört nicht zum Fahrschein, kann aber zusätzlich gekauft werden. Der Preis dafür ermäßigt sich, wenn die Reservierung zusammen mit dem Fahrkartenkauf vorgenommen wird. Inhaber einer Bahncard können Fahrscheine, nicht jedoch Zuschlagkarten und Reservierungen, zum halben Preis erwerben. Im Zug muß

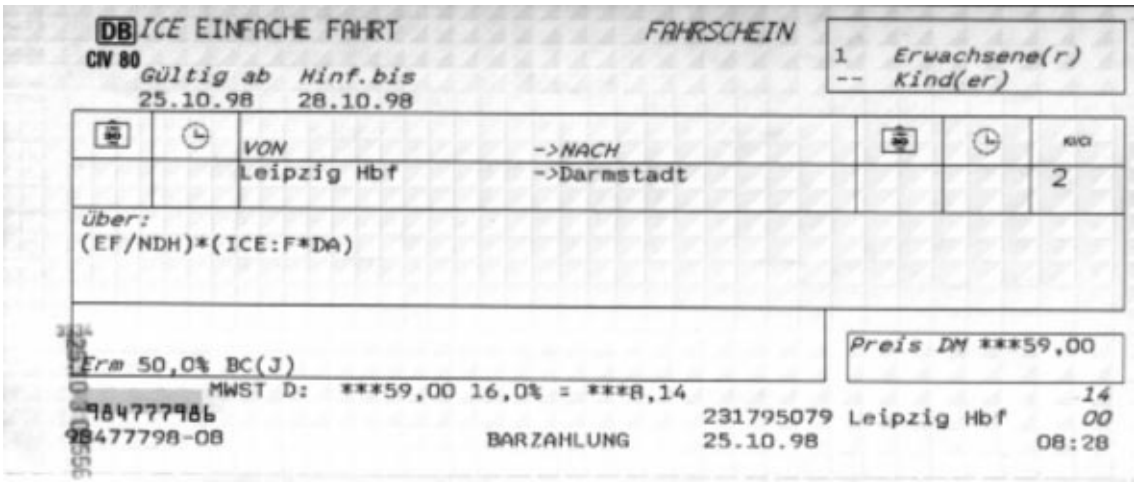
DB ICE EINFACHE FAHRT FAHRSCHEIN
CIV 80
Gültig ab Hinf.bis
25.10.98 28.10.98
1 Erwachsene(r)
-- Kind(er)
VON ->NACH
Leipzig Hbf ->Darmstadt 2
über:
(EF/NDH)*(ICE:F*DA)
Erm 50,0% BC(J)
Preis DM ***59,00
MWST D: ***59,00 16,0% = ***8,14
14
98477798b
231795079 Leipzig Hbf 00
98477798-08 BARZAHLUNG 25.10.98 08:28

Abb. 1.2: *Fahrkarte*

dann neben dem Fahrschein auch die Bahncard vorgezeigt werden. Fahrkarten sind an Bahnhöfen und in Reisebüros erhältlich, sie können jedoch auch im Zug erworben werden. Ebenso gegen Zahlung des Aufpreises Fahrscheine verändert werden, zum Beispiel für eine Verlängerung der Fahrstrecke oder zum Übergang von der zweiten in die erste Klasse.

Die Fahrscheine sind nicht übertragbar, es gibt jedoch keine Möglichkeit, nach einer Fahrtunterbrechung festzustellen, ob sie noch von derselben Person benutzt werden wie vorher. Das ist aber auch nicht nötig, denn jede Teilstrecke kann trotzdem nur einmal von einer Person befahren werden.

Die Fahrscheine werden in jedem Zug einmal oder mehrfach kontrolliert und durch einen Stempelabdruck entwertet. Hat ein Fahrgast seine Karte nicht oder nur teilweise genutzt, kann er sich das zuviel gezahlte Geld erstatten lassen.

Allgemein Ein Ticket ist ein Gutschein für eine vorausbezahlte Leistung. Ein Kunde schließt mit dem Anbieter einen Vertrag und erfüllt seine Pflicht aus diesem Vertrag – die Bezahlung – sofort. Im Gegenzug erhält er zunächst nur eine Zusicherung des Anbieters, daß jener seine Verpflichtungen zu einem späteren Zeitpunkt ebenfalls erfüllen werde. Als Nachweis bekommt der Kunde eine Eintritts- oder Fahrkarte. Sie kann er später gegen die Leistung einlösen, aber nur unter gewissen Randbedingungen, zum Beispiel zu einer bestimmten Zeit an einem bestimmten Ort.

Das funktioniert prächtig, wie ein jeder im Alltag wieder und wieder erfahren kann. Tickets sind kein Thema für eine Diplomarbeit, möchte man meinen.

1.2 Eine dramatische Problembeschreibung ...

Nur einen kleinen Schönheitsfehler gibt es und der soll behoben werden. Fahr- oder Eintrittskarten können für die Kunden eine recht unbequeme Angelegenheit sein.

Man kann sie rechtzeitig vorher kaufen. Dazu setzt man sich in sein Auto, staut sich eine Dreiviertelstunde durch die Stadt zur Verkaufsstelle, kauft, und staut sich eine weitere Dreiviertelstunde zurück. Dann hat man sein Ticket in der Tasche, drei Liter Benzin weniger im Tank, zwei neue Beulen im Blech und die Gewißheit, in ein paar Tagen Post vom Ordnungsamt zu bekommen, weil kein legaler Parkplatz zu finden war. Nie wieder.

Beim nächsten Mal möchte man es besser machen. Vorverkauf ist ein Angebot, das man nutzen kann, aber nicht muß. Man hat sich also für die abendliche Theatervorstellung fein angezogen und fährt zum Veranstaltungsort. Dort wird man vom Ende einer längeren Schlange Wartender empfangen. Schnell angestellt! Langsam, aber stetig kommt man der Kasse näher. Die Füße tun ein wenig weh in den neuen Schuhen, aber was ist das schon im Vergleich zu den Vorverkaufsstrapazen. Außerdem ist man gleich dran und dann kann man sich setzen. Denkste. Nur noch zwei Kunden stehen vor einem, als die Kassiererin das Schild »Ausverkauft« aufhängt.

Ein dritter Versuch. Diesmal soll es ins Kino gehen. Die Eintrittskarten hat man telefonisch vorbestellt. Jetzt kann nichts mehr schiefgehen. Frohen Mutes läuft man also zur Straßenbahn. Während man dort noch mit dem Fahrkartenautomaten kämpft, sieht man im Augenwinkel erst die Scheinwerfer und später die Rücklichter der ankommenden und weiterfahrenden Bahn. Einfach hineinspringen konnte man nicht, denn seit diese Automaten an jeder Haltestelle stehen, verkauft der Fahrer keine Fahrscheine mehr.[1] Die nächste Bahn kommt zwanzig Minuten später. Aber das reicht gerade noch, um rechtzeitig zur Vorstellung anzukommen. Wirklich? Nein, denn vorbestellte Karten sind spätestens eine halbe Stunde vor Beginn der Werbung abzuholen.

Wurde nicht schon vor fünfzehn Jahren [Bie84, Bre85] eine wunderschöne neue Konsumwelt angekündigt, in der man einfach von zu Hause aus einkaufen kann? Zu Hause steht inzwischen ein Computer mit Internetzugang, am Arbeitsplatz sowieso. Elektronisches Geld zum Einkaufen im Netz hat man auch. Wie schön wäre es doch, könnte man seine Tickets im Netz nicht nur bestellen und bezahlen, sondern auch gleich geliefert bekommen, ohne herumfahren oder zeitiger als rechtzeitig am Veranstaltungsort sein zu müssen.

1.3 ...und eine weniger dramatische

Gesucht werden Möglichkeiten, Tickets zu digitalisieren, um sie ihrem Käufer unmittelbar während des Verkaufsvorganges im Netz übermitteln zu können. Der Käufer muß das erworbene Ticket auf einem physischen Träger speichern und diesen zur später Kontrolle – etwa am Eingang oder im Zug – vorlegen können. Das bringt zwei grundlegende Probleme mit sich. Zum einen wird ein geeigneter Datenträger benötigt. Zum anderen ist der Käufer a priori nicht vertrauenswürdig – und digitale Daten, die ihm übermittelt werden, ohne Qualitätsverlust beliebig oft kopierbar.

Zunächst soll die Aufgabe noch respektvoll in gehöriger Entfernung umrundet und dabei aus verschiedenen Blickwinkeln betrachtet werden.

[1] In Darmstadt ist das seit einigen Monaten tatsächlich so.

Wertrepräsentation Als Gutschein repräsentiert eine Fahr- oder Eintrittskarte einen finanziellen Wert, nämlich ihren Kaufpreis. Der Besitzer oder, in manchen Fällen, eine namentlich festgelegte Person kann sie gegen eine Leistung in diesem Wert eintauschen.

Diese Möglichkeit ist jedoch an eine Reihe von Bedingungen geknüpft. Ein Fahrschein gilt nur auf bestimmten Strecken und in bestimmten Verkehrsmitteln. Eine Kinokarte kann nur für die Vorstellung verwendet werden, für die sie gekauft wurde. Die Gültigkeit hängt von *Parametern* ab.

Anders als Geld sind Tickets keine allgemeinen Zahlungsmittel. Sie können nicht zur Begleichung einer beliebigen Schuld verwendet werden. Die Einlösung gegen die bezahlte Leistung ist nur an festgelegten Stellen möglich, in der Regel bei derjenigen Einrichtung, von der oder in deren Auftrag sie verkauft werden. Ein Ticket ist nur *lokal* gültig. Wenn der Verkäufer das unterstützt, können unbenutzte Tickets jedoch auch wieder in Geld umgetauscht werden.

Der Wert einer Eintritts- oder Fahrkarte entspricht genau dem Wert der Leistung, für die sie gekauft wurde (und die durch die Parameter näher bestimmt wird). Es ist deshalb nicht nötig, diesen Wert in kleinere Einheiten aufzuteilen und mehrere solche Einheiten zu einem anderen Wert zusammenzufügen. Tickets sind in der Regel nicht *teilbar*, und sollen es auch nicht sein. Die Einlösung kann jedoch in mehreren Etappen erfolgen. So darf etwa eine Bahnfahrt unterbrochen und später fortgesetzt werden.

Genau wie (Bar-)Geld sind Tickets meist *anonym* und vor der Benutzung beliebig übertragbar. Gelegentlich ist aber auch ausdrücklich die Bindung an eine Person gewünscht. Das kommt vor allem bei Zeitkarten vor, die innerhalb eines längeren Zeitraumes beliebig oft, aber nur von einer Person benutzt werden dürfen, zum Beispiel bei Monatskarten im Nahverkehr. Ermäßigte Tickets sind zwar nicht unbedingt an eine Person gebunden, aber an eine Eigenschaft des Benutzers, etwa den Besitz einer Bahncard[2] oder die Eigenschaft, Student zu sein.

Da eine Fahrkarte oder Eintrittskarte einen Wert repräsentiert, darf sie *nicht leicht kopierbar* sein. Sonst könnte sie jeder mit wenig Aufwand vervielfältigen und so aus dem Nichts Werte erschaffen.

Zu guter Letzt ist der Wert eines Tickets nicht beständig. Im Gegenteil, oft ist von Anfang an ein *Verfallsdatum* festgelegt. Eisenbahnfahrkarten besitzen einen aufgedruckten Geltungszeitraum; Eintrittskarten für eine Veranstaltung werden spätestens mit dem Ende dieser Veranstaltung wertlos und Fahrscheine für den Nahverkehr gelten höchstens bis zur nächsten Preiserhöhung, auch wenn der Zeitpunkt dafür noch gar nicht bekannt ist.

Nachrichtenübermittlung Ein Ticket kann als Nachricht betrachtet werden. *Sender* ist die Verkaufsstelle. Sie erzeugt ein Objekt, das, in natürlicher Sprache ausgeschrieben, zum Beispiel den folgenden Inhalt haben könnte:

> Für die Vorführung des Films »Akte X – Der Film« am 12. September 1998 um 20:15 Uhr im Kino »Hollywood« hat jemand den Eintrittspreis für einen

[2] Für Automobilisten: Das ist ein Halbpreispaß der Deutschen Bahn AG.

> Platz in Reihe 5-7 bezahlt. Wer auch immer diese Eintrittskarte kurz vor diesem Zeitpunkt am angegebenen Ort vorlegt, ist hereinzulassen.

Empfänger ist die Kontrollstelle am Eingang oder der Kontrolleur im Zug. Anhand der Nachricht müssen sie entscheiden, ob ein Besucher zum Betreten des Veranstaltungsortes und ein Fahrgast zur Nutzung eines Verkehrsmittels berechtigt ist oder nicht.

Als *Transportkanal* für die Nachricht wird ein physischer Träger genutzt, den der Kunde von der Verkaufs- zur Kontrollstelle transportiert. Das ist naheliegend, denn der Kunde legt diesen Weg ohnehin zurück und die Nachricht bezieht sich auf ihn und niemanden sonst.

In der Regel ist die Nachricht so kodiert, dass sie von einem Menschen leicht und schnell erfaßt werden kann. Vereinzelt sind aber auch Systeme im Einsatz, die den menschlichen Einlasser ersetzen sollen. In diesem Falle wird die Nachricht maschinenlesbar gespeichert. Ein solches System benutzt zum Beispiel die Leipziger Messe auf dem neuen Messegelände. Die Eintrittskarten enthalten dort einen Magnetstreifen, auf dem Daten gespeichert sind. Am Eingang findet der Besucher Drehkreuze vor, die ihm den Weg versperren. Dort steckt er seine Karte in ein geheimnisvolles Kästchen, worauf das Drehkreuz ihm und nur ihm den Weg freigibt.

Die Sicht auf die Nachrichtenübermittlung ist in etwa die des Veranstalters. Er benötigt am Eingang oder im Fahrzeug Informationen, um eine Entscheidung fällen zu können. Das Ticket als Nachricht liefert ihm diese Informationen.

Authentifizierung Von der Nachrichtenübermittlung ist es nur ein kleiner Schritt zur Authentifizierung. Sie ist aus Sicht des Kunden bedeutsam, der eine Leistung bereits bezahlt hat und nun in Anspruch nehmen möchte. Dazu muß er die Bezahlung *nachweisen*.

Sein Ticket ermöglicht ihm diesen Nachweis. Voraussetzung dafür ist, daß Tickets nur durch ordnungsgemäßen Kauf erworben werden können. Der Besitz einer Eintritts- oder Fahrkarte zeigt zweifelsfrei, daß zumindest irgend jemand diese Karte gekauft und bezahlt hat. Das muß nicht derjenige sein, der sie vorlegt; sie könnte beispielsweise auch dem rechtmäßigen Besitzer gestohlen worden sein. In diesem Fall hat jener einfach Pech. Sein Nachweis ist weg und er kann nicht ins Kino. Dem Veranstalter kann das gleichgültig sein, solange er nur die Sicherheit hat, daß zu jedem vorgelegten Ticket ein Verkaufsvorgang stattgefunden und Geld in seine Kasse gebracht hat.

Sobald sich jemand ohne Kauf in den Besitz scheinbar gültiger Tickets bringen kann, funktioniert das nicht mehr. Der Besitz einer Eintrittskarte ist dann nicht mehr exklusiv denen vorbehalten, die eine bezahlt haben. Damit wird dieser Besitz zum Nachweis der Bezahlung ungeeignet.

Die Authentifikation erfolgt *anonym*. Der Kunde möchte nicht seine Identität nachweisen, sondern lediglich die Tatsache, daß er zu einem früheren Zeitpunkt eine bestimmte Handlung, eben den Kauf, vollzogen hat.

1.4 Anforderungen

Nun ist es an der Zeit, das Ziel konkret zu beschreiben und Forderungen an die gesuchten Verfahren aufzuzählen. Zunächst seien einige Voraussetzungen genannt.

Zwei Parteien sind am Verkauf und der Benutzung von Tickets beteiligt, der Anbieter, bei dem die Tickets später auch wieder eingelöst werden, und der Kunde, der sie kauft. Der Verkauf durch Dritte erscheint im Zusammenhang mit dem Internet wenig sinnvoll und soll daher nicht berücksichtigt werden.

Vorausgesetzt wird auf der Anbieterseite ein System zur Verkaufsdurchführung im Netz sowie die Bereitschaft, neue Prozeduren und neue Technik für die Ticketkontrolle einzuführen. Das Verkaufssystem muß sich nicht am Veranstaltungsort befinden, es kann bei einem Internet-Dienstanbieter untergebracht sein.

Der Kunde möge über eine beliebige EDV-Anlage mit Internet-Anschluß und einem halbwegs aktuellen Web-Browser verfügen. Die Anlage muß nicht seine eigene sein. Es kann sich auch um ein Gerät am Arbeitsplatz oder in einem Internet-Café handeln. Eine bestimmte Hard- oder Softwareplattform wird nicht verlangt.

Forderungen an die gesuchten Verfahren sollen in drei Klassen eingeteilt werden: Notwendig, wünschenswert und nice to have.[3] Kann ein Verfahren eine der *notwendigen* Forderungen nicht erfüllen, ist es unbrauchbar. Auf *wünschenswerte* Eigenschaften kann notfalls verzichtet werden, wenn es dafür gute Gründe gibt. Sie sind jedoch ausdrücklich als Entwurfsziele genannt und müssen gebührend berücksichtigt werden. Der Klasse *nice to have* schließlich werden Forderungen zugeordnet, die keine größere Bedeutung haben und die getrost unerfüllt bleiben können, wenn sich nicht im Laufe des Entwurfs eine Lösung anbietet. Zudem sollten derlei Eigenschaften nicht nachträglich implementiert werden. Das führt allzu leicht zu »creeping featurism«,[4] dem Problem der schleichenden Einführung von immer mehr Features, die eine Software größer, aber nicht unbedingt besser machen.

Außerhalb dieser drei Kategorien werden noch Regeln aufgeführt, nach denen Kinobetreiber mit Filmverleihern abrechnen. Sie sind nicht Gottes Gesetz und lassen sich zweifellos an neue Techniken anpassen, weshalb sie hier nur erwähnt werden sollen.

1.4.1 Notwendige Eigenschaften

Eintrittskarten zu Veranstaltungen und Fahrkarten sollen im Internet verkauft werden. Dort soll neben dem Verkaufsvorgang – diese Problem ist im Grunde gelöst – insbesondere auch die Distribution erfolgen. Der Rechner des Käufers sowie die Datenübertragung zwischen diesem und dem Verkaufssystem ist dabei nicht vertrauenswürdig, das heißt

[3] Die Frage nach Übersetzungen im Bekanntenkreis erbrachte zwei Vorschläge: »Bonus-Funktionen« und »sinnlose Funktionen«.

[4] »1. Describes a systematic tendency to load more chrome and features onto systems at the expense of whatever elegance they may have possessed when originally designed. (...) 2. More generally, the tendency for anything complicated to become even more complicated because people keep saying „Gee, it would be even better if it had this feature too“. The result is usually a patchwork because it grew one ad-hoc step at a time, rather than being planned.« [Ray96, Ray98]

die übermittelten Daten können vom Käufer sowie von Dritten manipuliert und kopiert werden.

Betrugssicherheit Zu den wichtigsten Entwurfszielen gehört deshalb die Betrugssicherheit. Der Kunde oder Dritte dürfen nicht in der Lage sein, sich durch einfache Manipulationen unberechtigt in den Besitz von Werten zu bringen. Andererseits muß auch der Kunde ausreichend vor Betrug geschützt sein, sonst wird er seine Tickets weiter auf herkömmliche Weise erwerben.

Prüfung und Unterscheidbarkeit Der Veranstalter muß die verkauften Fahr- oder Eintrittskarten auf *Echtheit* und *Gültigkeit* prüfen können. Werden Eintrittskarten für verschiedene Veranstaltungen, Fahrkarten für unterschiedliche Strecken oder auf andere Weise Tickets mit unterschiedlichem Geltungsbereich verkauft, so muß eindeutig feststellbar sein, welches Ticket wofür gilt. Der Anbieter muß eigene Tickets von denen anderer Anbieter unterscheiden können.

Entwertung Es muß möglich sein, benutzte Tickets zu entwerten. Dabei ist die Kopierbarkeit digitaler Daten zu berücksichtigen. Weiterhin muß die Möglichkeit bestehen, Tickets mit dem Ablauf einer gewählten Frist *automatisch ungültig* werden zu lassen, ohne daß sie dem Veranstalter vorgelegt werden.

Reservierung Eintrittskarten reservieren häufig gleichzeitig einen Sitzplatz. Das sollen sie auch tun, wenn sie im Netz gekauft werden.

Anonymität Herkömmliche Fahr- und Eintrittskarten sind bis auf wenige Ausnahmen anonym. Niemand kann feststellen, wer wann welches Ticket gekauft hat. Bei nicht anonymen Tickets bleibt immerhin noch die automatisierte Erfassung der Käufer und Benutzer unmöglich, so daß nicht im großen Stil Nutzerprofile verfertigt werden können. Es gibt keinen Grund, daran etwas zu ändern.

Dieser Punkt verdient besondere Beachtung, da bei der Benutzung von Computern und Netzdiensten leicht personenbezogene Daten in großem Umfang anfallen, wenn man sich nicht ausdrücklich um Vermeidung bemüht. Fallen Daten erst einmal an, wird sie auch jemand speichern und verwenden wollen. Zieht man die Möglichkeit böswilligen Mißbrauchs in Betracht, helfen dagegen auch keine Gesetze (die zudem ständigem Wandel unterliegen). Bereits der Systementwurf sollte deshalb jede unnötige Speicherung und Verarbeitung personenbezogener Daten vermeiden.

Robustheit und Zuverlässigkeit Der Benutzer eines herkömmlichen Tickets kann nicht viel falsch machen. Das Billett kann verloren gehen, in der Waschmaschine landen oder zu Hause vergessen werden. Wenn der Kunde es aber zum richtigen Zeitpunkt ungewaschen mit sich führt, kann er sich sicher sein, daß es akzeptiert wird. Selbst ein beschädigtes Ticket wird unter Umständen noch anerkannt. Eine kleine Portion Sorgfalt genügt, um den Verlust von Werten zu vermeiden. Das darf sich nicht ändern, wenn

das Ticket online gekauft wird. Dem Käufer darf kein Verlust (oder jedenfalls kein höheres Verlustrisiko als bisher) entstehen, wenn er nur die gleiche kleine Portion Sorgfalt aufwendet.

Der Anbieter andererseits muß unter allen Umständen in der Lage sein, vorgelegte Fahr- oder Eintrittskarten zu prüfen. Ist ihm das aus irgendwelchen Gründen nicht möglich, bleibt ihm nur die Wahl, entweder alle Kunden – die bereits bezahlt haben! – abzuweisen oder aber jeden ohne Kontrolle einzulassen. Letzteres ist im Falle einer einmaligen kurzzeitigen Störung vertretbar, bietet bei häufigem Auftreten jedoch eine Betrugsmöglichkeit.

Benutzbarkeit und Ergonomie Auf Kundenseite darf der Ticketkauf im Netz keine besonderen Handlungen erfordern, die weit über die Auswahl der gewünschten Karten und einige Mausklicks zum Bezahlen hinausgehen. Insbesondere ist es nicht zumutbar, vor dem Kauf besondere Software zu installieren, die dann vielleicht noch von Anbieter zu Anbieter wechselt. Auch darf keine besondere Anstrengung nötig sein, die gekaufte Bitfolge auf einen physischen Träger zu bannen und darauf zum Einsatzort zu transportieren.

Beim Anbieter ist vor allem der Kontrolle Aufmerksamkeit zu widmen. Der Verkauf erfolgt ja automatisiert durch einen Computer, so daß dabei keine allzu großen Probleme zu erwarten sind. Die Kontrolle aber muß schnell und einfach erfolgen; sie darf keine langwierigen und fehlerträchtigen Prozeduren erfordern. Am Eingang eines Kinos wird die Eintrittskarte kurz angeschaut und der Abrißabschnitt entfernt. Der Schaffner im Zug liest kurz die Angaben der Fahrkarte und benutzt seine Stempelzange zum Entwerten. Das darf nicht komplizierter werden.

1.4.2 Wünschenswerte Eigenschaften

Unabhängigkeit vom Datenträger Ein geeigneter Träger, auf dem die online gekauften Tickets bis zur Benutzung gespeichert und dann zur Kontrolle vorgelegt werden, ist noch nicht bestimmt. Falls dabei mehrere Möglichkeiten in Betracht kommen, sollte jede von ihnen prinzipiell einsetzbar sein.

Offline-Betrieb Datenkommunikation ist nach wie vor teuer und außerdem störanfällig. Das gilt erst recht, wenn als einziger Übertragungsweg Funkverbindungen zur Verfügung stehen, etwa in fahrenden Zügen oder an ungewöhnlichen Veranstaltungsorten. Die Ticketkontrolle sollte deshalb offline möglich sein, das heißt ohne Kommunikationsverbindung zur Verkaufsstelle oder woandershin. Das ist auch im stationären Einsatz wichtig. Allenfalls große Veranstalter sind in der Lage, das Verkaufssystem selbst zu betreiben. Alle anderen werden dafür die Dienste eines Internet-Providers in Anspruch nehmen und deshalb keinen ständigen Zugriff auf das Verkaufssystem haben.

Gegen gelegentlichen kurzen Datenaustausch, zum Beispiel mittels einer ISDN-Wählleitung, ist nichts einzuwenden.

Personalisierbarkeit Wie die einführenden Beispiele (Seite 9) zeigen, möchte man zuweilen ein Ticket an bestimmten Nutzer binden. Nur diese Person darf es dann benutzen. Das sollte auch für online verkaufte Fahr- und Eintrittskarten möglich sein.

HTML und HTTP, sonst nichts Java ist in. Jeder will es, jeder nutzt es. Scheinbar. Applets kosten Zeit und Geld, und zwar die Zeit und das Geld des Nutzers. Sie zu übertragen dauert und ihre Ausführung auf dem Rechner ruft auch selten einen Geschwindigkeitsrausch hervor. Zudem ist nach wie vor nicht für jeden Browser und jedes Betriebssystem eine brauchbare Java-Implementierung vorhanden. So kann beispielsweise der recht beliebte Browser Opera der norwegischen Firma Opera Software A/S keine Applets ausführen. Gerade dieses Produkt bietet aber besondere Unterstützung für Behinderte Benutzer. [Ope98] Nach Möglichkeit sollte der Ticketverkauf deshalb nicht nur ohne besondere, vom Käufer zu installierende Software, sondern auch ohne Java-Applets auf Kundenseite auskommen.

1.4.3 Nice to have

Mehrfachtickets Unter den Beispielen in Abschnitt 1.1 findet sich eines, in dem die Eintrittskarte zur einer Veranstaltung gleichzeitig Fahrschein für den Weg zum und vom Veranstaltungsort ist. Das ist eine gute Idee, bereitet aber bereits jetzt absehbare Probleme. Im angeführten Beispiel müsste das Nahverkersunternehmen dieselben Kontrollprozeduren ausführen können wie der Konzertveranstalter. Praktisch liefe das darauf hinaus, daß beide zur gleichen Zeit den elektronischen Ticketverkauf einführen müssen, wenn sie diesen Service anbieten möchten.

Eine Karte für mehrere Personen Wenn man im Internet eine Bitfolge kaufen kann, mit der man später ins Kino kommt, warum sollte man dann nicht auch eine kaufen können, die der ganzen Familie den Besuch ermöglicht? Fünfmal den gleichen Kaufvorgang zu vollziehen ist eine langweilige Angelegenheit und ohnehin möchte man ja auch fünf zusammenhängende Plätze buchen.

Wiederherstellung Im Laufe des Verkaufsvorganges erhält der Kunde irgendwann eine Bitfolge, die das Gekaufte darstellt. Sie wird bis zur Verwendung auf einem transportablen Träger abgelegt. Daß sich beliebig viele Kopien dieser Bitfolge herstellen lassen, ist nicht nur ein Problem hinsichtlich der Betrugssicherheit, sondern bietet auch die Möglichkeit, beschädigte Träger zu ersetzen und erneut mit der gekauften Bitfolge zu versehen. Im Verlustfall wird das nicht funktionieren, wenn das System betrugssicher ist und ein anderer das verlorene Billett findet und benutzt, doch gegen versehentliche Beschädigung ist man so gewappnet.

Wenn das Verfahren also die einfache Herstellung von Sicherheitskopien erlaubt, kann dies dem Käufer ausdrücklich angeboten werden.

Rückgabe und Erstattung Veranstalter können ihren Kunden den Kaufpreis für unbenutzte Tickets erstatten. Das tut beispielsweise die Deutsche Bahn AG für ihre Fahrkarten. Es währe angenehm, legte der Online-Handel dem keine Steine in den Weg.

1.4.4 Abrechnungskontrolle der Filmverleiher

Die folgenden Ausführungen orientieren sich an einem Text über die Abrechnungskontrolle des Verbandes der Verleiher e.V. [WFN]

Kinobetreiber erhalten die Filme zur Vorführung von einem Verleih. Welchen Betrag sie dafür jeweils an den Verleiher zu entrichten haben, hängt von der Zahl der Vorführungen und der Besucher ab. Sie müssen folglich mit dem Verleih abrechnen; diese Abrechnung wird vom Verband der Verleiher in Stichproben überprüft.

Die Abrechnung erfolgt tageweise. Für jeden Film und jeden Spieltag sind neben Datum und Spielort die Vorstellungszeiten, die Besucherzahlen und die Bruttoeinnahmen anzugeben.

Damit Abrechnungskontrollen überhaupt möglich sind, müssen die verkauften Eintrittskarten fortlaufend numeriert sein (mindestens von 1 bis 100.000; für Logenplätze mindestens 1 bis 20.000) sowie einige festgelegte Merkmale aufweisen.

Eine ordnungsgemäße Kinokarte besteht aus einem Stammabschnitt und dem Abrißteil. Der Stammabschnitt enthält den Namen des Kinos, den Ort, die Platzkategorie, die Firmenanschrift der Druckerei, die Kartennummer sowie ein Siegel der Spitzenorganisation der Filmwirtschaft (SPIO).

Der Abrißteil muß ebenfalls die Kartennummer und ein Siegel der SPIO enthalten. Weiterhin ist der Aufdruck »Abriß, als Eintrittsausweis ungültig« gefordert.

Zusätzlich zur Abrechnung führt der Kinobetreiber einen Tageskassenrapport, der zehn Jahre aufbewahrt und gegebenenfalls kontrolliert wird. Dort werden täglich zu jedem Filmtitel die Uhrzeiten, zu denen der Film aufgeführt worden ist sowie der Verkauf der Eintrittskarten notiert. Der Verkauf wird nach Platzgruppen, Nummern, Stückzahlen und Einnahmebeträgen spezifiziert.

Zur Abrechnungskontrolle kauft ein Kontrolleur incognito eine Eintrittskarte. Er besucht die Vorstellung und zählt dort die Zuschauer. Danach meldet er die Zuschauerzahl und die Nummer seiner Eintrittskarte an den entsprechenden Verleih. Ist in der Abrechnung des Kinos die Kartennummer vermerkt und die Zuschauerzahl korrekt angegeben, gibt es nichts zu beanstanden. Tauchen hingegen Unstimmigkeiten auf, muß der Kinobetreiber mit einer Revision rechnen, bei der sämtliche Unterlagen überprüft werden.

1.5 Verwandte Probleme

1.5.1 Elektronisches Geld

Bereits lange vor dem aktuellen Internet-Hype beschäftigten sich Kryptologen mit der Frage, ob und wie sich Geld digitalisieren läßt.

Ideales digitales Geld hat sechs Eigenschaften: [OkOh91, Schn96]

- Unabhängigkeit von physischen Orten, Transferierbarkeit über Datennetze

- Sicherheit gegen Kopieren und Wiederverwendung
- Schutz der Privatsphäre des Benutzers
- Offline-Fähigkeit, d.h. Bezahlen ohne Netzanschluß
- Transferierbarkeit zwischen den Nutzern
- Teilbarkeit der digitalen Geldeinheiten

Tatsächlich eingesetzte Systeme bieten jeweils nur eine Auswahl daraus. Rivest stellt gar Thesen zur Diskussion, nach denen solch ein ideales System gar nicht notwendig sei und sich nicht durchsetzen werde. [Riv97] So sei etwa die Übertragbarkeit zwischen den Nutzern verzichtbar, wenn von überall problemlos eine Datenverbindung zur Bank aufgebaut werden könne.

Auf einige der genannten Eigenschaften kann man allerdings unter keinen Umständen verzichten. In jedem Falle ist Betrugssicherheit notwendig; ein Zahlungssystem, das jedem die Herstellung beliebiger Geldmengen erlaubt, ist keins. Wünschenswert ist der Schutz der Privatsphäre, denn Konsumentenprofile, die gar nicht erst entstehen, kann auch niemand nutzen. Die Möglichkeit, ohne Zahlungen ohne Netzverbindung zu einer Bank abzuwickeln, ist für Zahlungssysteme im richtigen Leben interessant, für den Handel im Internet hingegen offensichtlich unnötig.

An einer digitalen Zahlung sind mindestens drei Parteien beteiligt: Eine Bank, ein Händler und ein Kunde. Der Kunde besitzt anfangs Geld auf einem Konto, das während des Verkaufsvorganges auf irgendeine Weise auf ein Konto des Händlers übertragen werden muß. Die Bank soll dabei nichts über die Einkaufsgewohnheiten des Kunden erfahren und dem Händler soll der Kunde seine Identität nicht offenbaren müssen, jedenfalls nicht solange er ehrlich ist.

Derzeit sind im wesentlichen vier Konzepte für elektronische Geldübertragungen bekannt. [FuWr96]

Am einfachsten sind Systeme auf der Basis von *Überweisungen*. Sie nutzen im wesentlichen eine schon länger existierende Möglichkeit des Geldtransfers. Für Zahlungen im Internet muß der Kunde lediglich in der Lage sein, seiner Bank über das Netz einen Überweisungsauftrag zu erteilen. Ist das auf ausreichend sichere Weise möglich, steht der Zahlung nichts im Wege. Dabei erfährt die Bank jedoch - genau wie bei herkömmlichen Überweisungen - genau, wann der Kunde wem welchen Betrag zahlt. Auch gegenüber dem Händler bleibt der Kunde nicht anonym.

Auch *scheckbasierte Systeme* (zu dieser Klasse gehören auch solche für *Kreditkartenzahlungen*) beruhen auf einem bereits seit langem benutzen Prinzip, das den Bedürfnissen der digitalen Konsumwelt angepaßt wird. Der Kunde übergibt dem Händler einen Schuldschein. An die Stelle der Unterschrift tritt eine digitale Signatur; die Bank garantiert die Deckung bis zu einem gewissen Höchstbetrag. Der Händler reicht den Scheck bei seiner Bank ein. Die schreibt seinem Konto den entsprechenden Betrag gut. Genau wie bei den Überweisungssystemen bleibt die Privatspäre des Kunden praktisch ungeschützt.

Münzsysteme orientieren sich am Bargeld, bilden es aber nicht vollständig nach. Der Kunde besorgt sich von der Bank eine elektronische Münze. Das ist ein Datensatz, der

einen gewissen Wert repräsentiert und im dem Computer des Kunden oder auf einer Chipkarte gespeichert werden kann. Die Bank bucht den entsprechenden Betrag vom Konto des Kunden ab. Beim Kauf übergibt der Kunde dem Händler eine oder mehrere digitale Münzen im gewünschten Wert. Der Händler reicht diese bei seiner Bank ein und erhält ihren Wert auf seinem Konto gutgeschrieben. Diese Verfahren bieten elektronisches Geld im engeren Sinne. Eine Münze muß nicht an den Kunden gebunden sein, so daß jener gegenüber dem Händler anonym bleiben und die Bank aus den rücklaufenden Münzen keine Schlüsse auf Einkäufe des Kunden ziehen kann.

Die vierte Klasse schließlich bilden Zahlungssysteme mit einem *sicheren Zähler*. Sie benötigen manipulationssichere Hardware, die zwar im Besitz des Kunden ist, von ihm aber nicht anders als vorgesehen benutzt werden kann. Dieser Ansatz wird in Abschnitt 1.5.4 gesondert behandelt.

Ein Problem, das alle Verfahren zu lösen haben, ergibt sich aus der einfachen Kopierbarkeit digitaler Daten. Es ist zu verhindern, daß Werte vervielfältigt und für mehrere Zahlungen eingesetzt werden. Dafür gibt es im wesentlichen vier Ansätze. Nichtanonyme Verfahren, die sich an herkömmlichen Zahlungsmethoden orientieren, machen Betrug mit (eigenem) Geld einfach dadurch unmöglich, daß die Identität des Zahlenden bekannt ist. Online-Verfahren erfordern beim Bezahlen Kommunikation mit einer Bank, die die mehrfache Verwendung von Werteinheiten durch Abgleich mit einer Datenbank sofort erkennt. Anonyme Offline-Verfahren schließlich arbeiten entweder mit manipulationssicheren Geräten beim Bezahlenden oder mit kryptographischen Protokollen, die die Identität von Betrügern offenbaren, ehrliche Nutzer aber anonym bleiben lassen. [FuWr96, Schn96, Wai88, Bra98]

1.5.2 Vorausbezahlter Strom

Wasser, Strom und Gas kommen hierzulande aus der Wand. Bezahlt wird später. Den Verbrauch ermitteln Zähler, die vor Manipulationen geschützt sind und regelmäßig abgelesen werden. Anhand der abgelesenen Werte stellt das Versorgungsunternehmen eine Rechnung, die vom Kunden bezahlt wird. Oder auch nicht. Kann oder will ein Kunde nicht zahlen, so bleibt lediglich die Möglichkeit, ihn von der weiteren Versorgung auszuschließen, damit der Schaden nicht noch größer wird (und um Druck auf ihn auszuüben).

Eine andere Möglichkeit, beispielsweise Elektrizität zu verkaufen, sind Systeme mit Vorauszahlung. Der Kunde zahlt zunächst und erhält dafür das Recht, dem Netz die bezahlte Energie zu entnehmen. Ist der gezahlte Betrag verbraucht, wird die Versorgung vom Zähler unterbrochen.

Dazu könnte man Münzautomaten benutzen, aber sie zu leeren wäre ebenso aufwendig wie Zählerablesungen. Zudem provozierte das Vorhandensein von Geld Angriffe auf Technik und Personal. Die Zähler sollen deshalb nicht selbst Geld einnehmen, sondern lediglich darüber informiert werden, wieviel Strom der Kunde dem Netz entnehmen darf.

Anderson und Bezuidenhoudt haben die Einführung eines derartigen Systems in Südafrika untersucht. [AnBe96] Schwerpunkt der Betrachtung sind die Probleme, die im Praxiseinsatz in puncto Sicherheit und Zuverlässigkeit aufgetreten sind.

Die Kunden erwerben in einer Verkaufsstelle einen digitalen Gutschein. Das ist letzt-

lich nichts als eine Zahl, die der Zähler als Anweisung interpretiert. Zur Übermittlung an den Zähler wird diese Zahl in einem handlichen EEPROM-Gerät, einer Chipkarte oder auf einer Magnetstreifenkarte gespeichert oder aber einfach vom Kunden über eine Tastatur eingegeben.

Die Sicherheitsanforderungen sind leicht zu erkennen. Die elektronischen Gutscheine dürfen nicht fälschbar sein und nicht mehrfach verwendbar, sei es in einem Zähler oder in mehreren verschiedenen. Sie müssen also entweder manipulationssicher oder jeweils einmalig sein. Als Problem erweist sich dabei der Informationsfluß zwischen Verkaufssystem und Zähler. er kann nur in einer Richtung erfolgen, denn dafür stehen als Kanal lediglich die elektronischen Gutscheine zur Verfügung. Eine Rückfrage oder gar umfangreiche Interaktion, wie sie in vielen kryptographischen Protokollen auftritt, ist nicht möglich.

Die Lösung ist dennoch einfach. Jeder Zähler erhält eine eindeutige Nummer. Verkaufsstelle und Zähler sind außerdem im Besitz eines gemeinsamen geheimen Schlüssels. Er kann auf der Verkäuferseite aus der Zählernummer und einem Händlerschlüssel abgeleitet werden. Der Verkäufer benutzt den gemeinsamen Schlüssel, um Anweisungen – zum Beispiel: »Erhöhe den Zähler für die bezahlten Kilowattstunden um den Wert 10« – an den Zähler zu verschlüsseln. Sie sind dann nur für das Gerät lesbar, für das sie verschlüsselt wurden. Der Zähler selbst sorgt dafür, daß ihm ein und derselbe Gutschein nicht mehrfach angeboten werden kann.

Die kryptographische Sicherung wird durch weitere Sicherheitsfunktionen ergänzt, unter anderem durch Methoden zum Wechsel von Schlüsseln und zur Betrugserkennung.

Das beschriebene Verfahren erwies sich als tauglich. Gleichwohl zeigten sich bei der Einführung Sicherheitsprobleme. Ursache waren meist Details im Design und der Implementierung der einzelnen Geräte. Sie wurden von den Kunden zufällig entdeckt und ausgenutzt.

Mit längerem Einsatz traten weitere Probleme zu Tage. Neben Funktionsfehlern der Technik gehörten dazu Schwierigkeiten im Verhalten der Kunden – einige meldeten einen Defekt, wenn der gezahlte Betrag verbraucht war – und im Systembetrieb, beispielsweise beim Vergleich von verbrauchten und verkauften Mengen zur Betrugserkennung.

1.5.3 Elektronische Briefmarken

Die amerikanische Post (U.S. Postal Service) testet seit Ende März 1998 ein System der Firma E-Stamp[5], das den Verkauf von Briefporto über das Internet ermöglichen soll [USPS98]. Der USA-weite Einsatz ist noch für dieses Jahr geplant.

Postkunden, die das Verfahren nutzen wollen, müssen zunächst eine besondere Software erwerben, die derzeit nur für Betriebssysteme der Windows-Familie angeboten wird. Zusammen mit der Software wird ein manipulationssicheres Gerät geliefert, das an den PC des Nutzers anzuschließen ist. Dieses Gerät speichert Informationen über bezahltes und verbrauchtes Porto. Es kann über das WWW-Angebot der Firma E-Stamp mit bis zu 500 Dollar aufgeladen werden. Gezahlt wird mit Kreditkarte, Abbuchung vom Konto oder im Voraus mit einem Scheck.

[5] http://www.e-stamp.com/

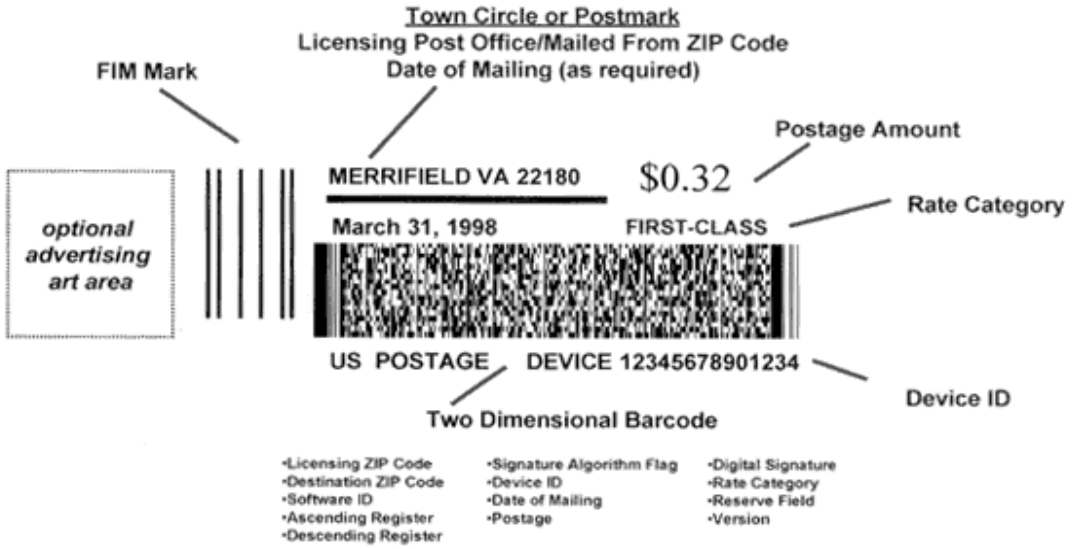

Abb. 1.3: *Maschinenlesbare Briefmarke. Quelle: [USPS98]*

Der gespeicherte Betrag kann nun verwendet werden, um Briefe zu frankieren. Die Software erzeugt dazu mit Hilfe des Sicherheitsgerätes ein Etikett, das direkt auf einen Briefumschlag gedruckt werden kann. Es besteht im wesentlichen aus einem PDF417-Barcode, der maschinell gelesen werden kann. Darin sind alle Daten kodiert, die zur Prüfung der Echtheit benötigt werden, unter anderem: [ESta98]

- der Portobetrag
- das Erzeugungsdatum
- die Seriennummer des Sicherheitsgerätes
- die Adressen von Absender und Empfänger

Weiter ist eine digitale Signatur enthalten, die Veränderungen erkennbar macht und verhindert, daß elektronische Briefmarken ohne Mitwirkung des Sicherheitsgerätes erzeugt werden.

Einzelheiten und Kontrollverfahren sind in [TYH96] beschrieben. Ausgangspunkt der Entwicklung war nicht die Briefmarke, sondern Frankiermaschinen und die Betrugsfälle, die im Zusammenhang damit auftreten.

Neben der Fälschung von Wertzeichen muß angesichts der leichten Kopierbarkeit auch die mehrfache Verwendung verhindert werden oder wenigstens erkennbar sein. Dafür sorgen die Parameter, die im maschinenlesbaren Symbol kodiert sind. Anhand der Seriennummer des Sicherheitsgerätes ist im Betrugsfall die Quelle ermittelbar. Die Angaben über Absender und Empfänger erlauben die Benutzung von Kopien allenfalls für

Sendungen, die vom gleichen Absender zum gleichen Empfänger geschickt werden. Das Erzeugungsdatum ermöglicht die Begrenzung des Gültigkeitszeitraumes.

Um Betrug zu erschweren, müssen nicht alle Briefe überprüft werden. Bereits die Kontrolle von Stichproben kann das Entdeckungsrisiko für Betrüger so weit erhöhen, daß sich der Betrug nicht lohnt.

1.5.4 Telefonkarten

Seit Ende der achtziger Jahre wurden viele Münzfernsprecher in der Bundesrepublik und in anderen Ländern durch Kartentelefone ersetzt. Die verbrauchten Telefongebühren werden bei diesen Geräten von einer Chipkarte abgebucht. Es handelt sich um ein Zahlungssystem mit sicherem Zähler, das bereits in Abschnitt 1.5.1 erwähnt wurde.

Die Vorteile für den Anbieter liegen auf der Hand. Im Gegensatz zu Münzfernsprechern enthält ein Kartentelefon kein Geld und bietet damit weniger Anreiz zu Vandalismus. Der Kunde hat eher Nachteile, denn er gibt der Telefongesellschaft beim Kartenkauf praktisch einen zinslosen Kredit und muß, will er jedes öffentliche Telefon benutzen können, zwei verschiedene Zahlungsmittel mit sich herumtragen.

Der Chip auf der Karte enthält einen Zähler, dessen Wert bei einer neuen Karte dem Kaufpreis entspricht. Wird die Karte benutzt, verringert das Telefon den Wert des Zählers um den zu zahlenden Betrag. Sobald der gesamte Kaufpreis verbraucht ist, wird die Karte wertlos [FRW98].

Der Zähler der Karte kann nicht nur von Telefonapparaten verändert werden, sondern von jedem, der über einen Computer mit Chipkartenschnittstelle verfügt. Die Schaltung des Chips sorgt jedoch dafür, daß Veränderungen nur in einer Richtung möglich sind. Der Wert des Zählers läßt sich nur verringern. Das genügt zum Schutz vor Betrug, denn wer den Wert seiner Telefonkarte eigenmächtig verringert, fügt nur sich selbst Schaden zu.

Lange haben die Telefonkarten der Deutschen Telekom Angriffen widerstanden. Aber Betrüger sind hartnäckig und so gelang es schlißlich doch, einen lange nur theoretisch diskutierten Angriff auf das System umzusetzen. Er ist ebenso einfach wie das Funktionsrinzip der Karte: Man hält sich nicht mit Manipulationsversuchen an gekauften KarteTelefonkarte auf, sondern stellt einfach seine eigenen her. Das vertraut darauf, daß die Kartenlogik Manipulationen verhindert. Diese Bedingung wird von Kartensimulatoren verletzt und es ist keine Möglichkeit vorgesehen, die Echtheit der verwendeten Telefonkarten im Apparat zu prüfen.

Die Umsetzung des Angriffs ist allerdings immer noch recht kompliziert und lohnt sich für den normalen Telefonierer kaum.

2 Ansätze

Ideenproduktion Ramschvokabel aus der Nebelwelt der Zeitungsredaktionen, Werbeagenturen und PR-Gaunereien. Zentrum der Ideenproduktion ist das Brainstorming, wo alle Mann hoch beieinandersitzen und von Platons wie von Hegels Idealismus aber trotzdem keine Ahnung haben. Gott sei Dank.
(Eckhard Henscheid: Dummdeutsch)

Der Vorbereitungen sind nun genug getan und die Arbeit kann beginnen. Zunächst soll eine ideale Welt angenommen werden, in der alles geht, was theoretisch möglich ist. Am Ende des Kapitels werden wir wissen, welche Konzepte unter welchen Voraussetzungen einsetzbar sind. Ob diese Voraussetzungen in der Realität erfüllbar sind, wird später untersucht.

2.1 Beteiligte und ihre Rollen

Die folgenden Begriffserklärungen stellen keine Definitionen im mathematischen Sinn dar. Das ist Absicht, denn die Entwicklung einer Theorie der Kinokasse ist nicht Thema dieser Arbeit.

Veranstalter Der Veranstalter bietet gegen Entgelt Leistungen an. Kunden, die eine Leistung in Anspruch nehmen möchten, müssen im Voraus zahlen und erhalten dafür ein Ticket. Der Veranstalter betreibt also eine Verkaufsstelle.

Die Verkaufsstelle ist der Ort, an dem Tickets erzeugt werden. Sie liegt im Einflußbereich des Veranstalters. Die Interaktion der Verkaufsstelle mit der Außenwelt, das heißt mit jedem anderen als dem Veranstalter, beschränkt sich auf Verkaufsvorgänge. Da die Verkaufsstelle in unserem Fall ein Computer ist, soll sie auch als *Verkaufssystem* bezeichnet werden.

Verkaufsvorgang heiß der Prozeß, in dessen Verlauf ein Ticket erzeugt und an den Kunden übergeben wird. Er soll über das Internet erfolgen. Die Existenz geeigneter Zahlungsverfahren wird vorausgesetzt. Während des Verkaufsvorganges können zwischen dem Rechner des Käufers und dem Verkaufssystem in beiden Richtungen mehrfach Daten ausgetauscht werden. Das Verkaufssystem kann dabei Daten speichern.

Am Ende der Verkaufsprozedur besitzt der Käufer das Billett, das ihm vom Verkaufssystem übermittelt wurde.

Das Ticket kann nur eine Bitfolge sein, denn nichts anderes kann das Netz übertragen. Sie wird auf einem leicht zu transportierenden Datenträger gespeichert, etwa einer Chipkarte oder einem Stück Papier.

Nach einem geeigneten Träger werden wir später suchen. Er wird in jedem Falle maschinenlesbar sein. *Aktive* Träger, zum Beispiel Chipkarten, können darüber hinaus Berechnungen ausführen, während ein *passiver* Träger lediglich einmal mit Daten gefüllt und danach nur noch gelesen werden kann.

Der Veranstalter tritt noch in einer zweiten Rolle auf. Er betreibt eine oder mehrere Kontrollstationen, zum Beispiel an Veranstaltungsorten oder in Fahrzeugen.

Eine Kontrollstation prüft Tickets auf Echtheit und Gültigkeit. Fallen beide Prüfungen positiv aus, wird die entsprechende Leistung gewährt. Andernfalls wird der Überbringer des Tickets abgewiesen. Eine Kontrollstation kann mehrere *Kontrollpunkte* umfassen, zum Beispiel mehrere Eingänge zum Veranstaltungsort, und verfügt über einen lokalen Speicher.

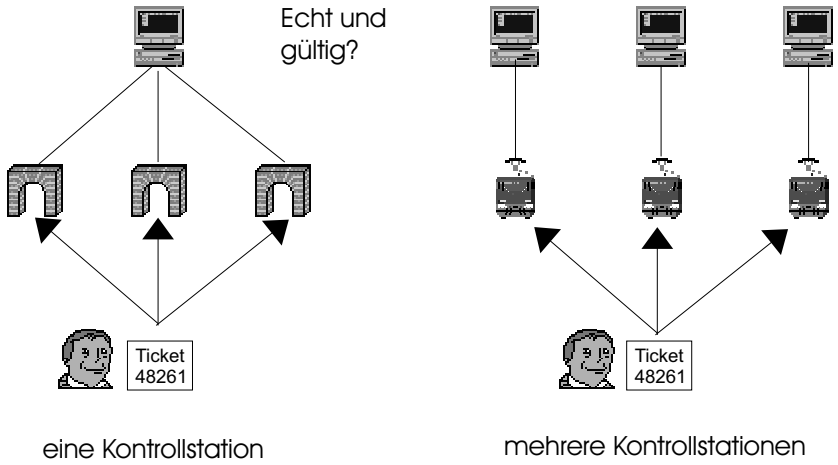

Abb. 2.1: *Kontrollstationen und Kontrollstellen*

Die Kontrollpunkte einer einzelnen Station können also auf einen gemeinsamen Datenbestand zugreifen und auf diese Weise miteinander kommunizieren. Jede Kontrollstation aber möge autonom arbeiten, solange nicht ausdrücklich etwas anderes gefordert ist.

Echt sind Tickets, die von der Verkaufsstelle erzeugt und danach nicht verändert wurden. Eine Prüfung der Echtheit anhand physischer Merkmale ist offensichtlich nicht möglich, denn das Ticket soll ja als Bitfolge im Netz übertragbar sein.

Gültig heißt ein Ticket, wenn es echt ist, noch nicht benutzt wurde und die Ticketparameter, die den Geltungsbereich festlegen, vorgegebenen Forderungen genügen. Die Kontrollstation muß in der Lage sein, ein vorgelegtes Ticket als benutzt zu kennzeichenen und es damit zu entwerten.

Kunde Der Kunde nimmt, wie der Veranstalter, nacheinander zwei verschiedene Rollen ein.

Als Käufer baut er eine Verbindung zum Verkaufssystem auf und erwirbt ein Ticket. Das bewahrt er sorgsam auf, bis er es zu einem späteren Zeitpunkt als

Besucher einer Kontrollstation vorlegt und, sofern es als gültig erkannt wird, die bezahlte Leistung erhält. Als Besucher ist der Kunde ein potentieller Betrüger (möchte aber nicht als solcher behandelt werden!)

Die Aufgabe Gesucht sind Verfahren, nach denen eine Kontrollstation über Echtheit und Gültigkeit eines vorgelegten Billetts entscheiden kann. Dabei sind fünf Probleme zu lösen.

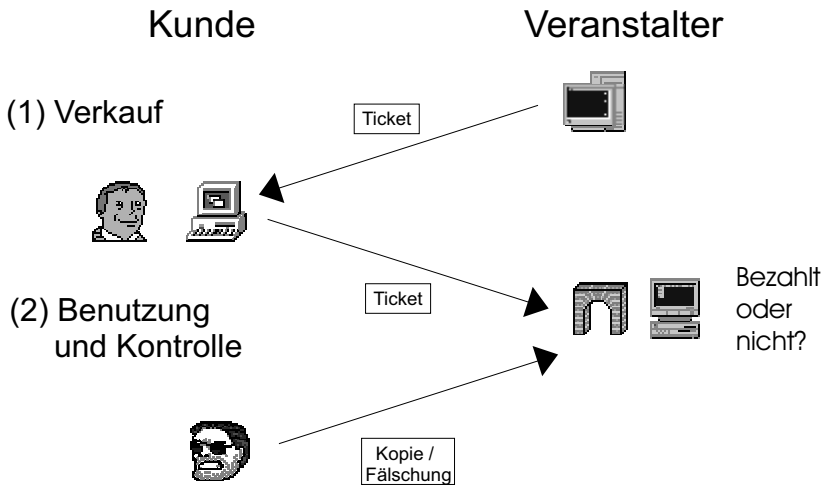

Abb. 2.2: *Rollenverteilung*

Die *Echtheit* umfaßt Authentizität und Integrität. [Beu94] Für jedes Ticket muß festgestellt werden, ob es von der Verkaufsstelle des Veranstalters erzeugt und ob es danach nicht verändert wurde. Bei Verwendung eines aktiven Datenträgers ist statt dessen die Authentizität des Trägers festzustellen. Herkömmliche Tickets sind in jedem Fall an ihren Träger gebunden. Die Echtheit wird durch Sicherheitsmerkmale garantiert, die oft recht simpel sind (z.B. Papierfarbe und -art, Gestaltung des Aufdrucks), sich aber im Verhältnis zum Kaufpreis genügend schwer fälschen lassen.

Die Gültigkeit eines Tickets ist durch Parameter bestimmt. Für jede Anwendung sind geeignete Parameter zu suchen. Auf ein herkömmliches Billett sind sie in der Regel aufgedruckt und werden von einem Menschen geprüft.

Falls kein manipulationssicheres Gerät das Kopieren verhindert, muß die Verwendung mehrerer Kopien eines Tickets erkennbar gemacht werden. Dieses Problem ist neu.

Tickets sollen bei der Kontrolle entwertet werden. Bei kopierbaren Tickets muß die Entwertung für alle existierenden Kopien wirksam werden, sobald nur eine einzige von

ihnen einer Kontrollstation vorgelegen hat. Traditionellen Eintrittskarten verfügen dafür über einen Abrißabschnitt; zur Entwertung von Fahrkarten benutzt man häufig einen Stempelaufdruck.

Bei alldem sollen die Kunden anonym bleiben können, wenigstens solange sie ehrlich sind.

2.2 Ein naiver Vorschlag

Weil er fast jedem, dem man die Idee des elektronischen Billettverkaufs unterbreitet, spontan in den Sinn kommt, sei hier zunächst der naivste aller Ansätze vorgestellt.

Jedes Ticket erhält beim Verkauf eine eindeutige Nummer. Sie wird so gewählt, daß sie von Dritten nur schwer erraten werden kann.[6] Das Verkaufssystem legt alle Gültigkeitsparameter (zum Beispiel Veranstaltung, Datum, Uhrzeit) unter dieser Nummer in einer Datenbank ab. Der Kunde erhält als Ticket diese Nummer in irgendeiner handhabbaren Form. Dafür genügt ein passiver Datenträger. Die Kontrollstation greift mittels der Nummer auf die Datenbank zu und prüft die dort abgelegten Parameter. Liegen sie in vorgegebenen Wertebereichen, wird das Ticket akzeptiert. Gleichzeitig wird das Ticket entwertet, indem seine Benutzung in der Datenbank vermerkt wird. Der Datensatz könnte an dieser Stelle auch einfach gelöscht werden, um alle weiteren Kopien ungültig zu machen.

Dieser Ansatz hat Nachteile. Die Datenübertragung zwischen Verkaufs- und Kontrollsystem erfolgt auf zwei getrennten Wegen. Fällt einer von ihnen aus, kann ein Ticket nicht mehr geprüft werden. Insbesondere kann dem Kunden ohne eigenes Verschulden ein Verlust entsstehen, denn er kontrolliert nur einen der beiden Kanäle.

Der Betrieb der Datenbank ist wegen der Anforderungen zum einen an die Verfügbarkeit und zum anderen an die Sicherheit – wer Zugriff auf die Datenbank hat, kann nach Belieben Tickets erzeugen – recht aufwendig. Hinzu kommen die Kommunikationskosten, wenn der Rechner mit dem Verkaufssystem an einem anderen Ort betrieben wird als die Kontrollstation(en).

Diese Nachteile gilt es zu vermeiden. Die Tickets sollen im wesentlichen den einzigen Kanal bilden, über den Daten von der Verkaufs- und Kontrollstelle übermittelt werden. Die Kontrolle soll auch nicht von Zustandsänderungen abhängig sein, die während des Verkaufs auf der Veranstalterseite eintreten, sondern allein das Ticket als Grundlage nehmen. Dabei hilft die Kryptographie.

2.3 Eine Kontrollstation

Zur Vereinfachung sei zunächst nur eine einzelne Kontrollstation betrachtet. Sie erhält Nachrichten – die Tickets – vom Verkaufssystem, kann selbst aber keine dorthin senden.

Wieder erhält jedes Billett beim Verkauf eine eindeutige Nummer. Sie dient jedoch nicht als Schlüssel zum Datenbankzugriff, sondern wird einfach auf dem Ticket vermerkt.

[6] Dazu muß man lediglich hinreichend lange Zufallszahlen benutzen.

Nur Kopien ein und desselben Tickets tragen die gleiche Nummer. Hinzu kommen die Parameter, die den Geltungsbereich bestimmen.

Diesen ersten Schritt könnte jeder auch ohne ordnungsgemäßen Kauf vollziehen. Damit nur das Verkaufssystem Tickets erzeugen kann, werden die Daten mittels eines kryptographischen Algorithmus um eine Prüfsumme ergänzt, die nur vom Verkaufssystem erzeugt und wenigstens von der Kontrollstation (eventuell auch von jedermann) verifiziert werden kann. Die Kryptographie hält dafür geeignete Algorithmen bereit; die Einzelheiten sollen im Moment nicht weiter interessieren.

Die Prüfsumme hängt sowohl von der Nachricht ab, als auch von einem Schlüssel, der nur dem Veranstalter bekannt ist. Damit ist neben der Authentizität auch die Integrität des Tickets geschützt – Manipulationen an der Nachricht machen die Prüfsumme ungültig.

Die Kontrollstation kann damit die Echtheit eines vorgelegten Tickets feststellen. Mit dem Billett erhält sie auch alle Gültigkeitsparameter zur Prüfung. Die eindeutige Nummer ermöglicht die Entwertung. Dazu müssen lediglich die Nummern aller kontrollierten Tickets lokal in einer Liste gespeichert werden. Ist eine Nummer bereits in der Liste enthalten, handelt es sich beim vorgelegten Ticket um ein bereits benutztes. Die Entwertung umfaßt sämtliche Kopien, denn sie tragen ja alle die gleiche Nummer.

Das Verfahren ermöglicht den Kunden Anonymität, denn zu keinem Zeitpunkt müssen sie ihre Identität offenbaren. Darüber hinaus ist der Verkaufsvorgang einfach, denn dabei wird, abgesehen von der Zahlung, lediglich eine Nachricht zum Käufer übertragen und dort auf einem (passiven) Datenträger abgelegt.

Ohne Schwierigkeiten können auch personengebundene Tickets (vgl. 1.1) hergestellt werden. Dazu führt man einfach einen numerierten Kundenpaß ein, wie er zum Beispiel von Zeitkarten im öffentlichen Nahverkehr bekannt ist. Seine Nummer wird beim Kauf angegeben und vom Verkaufssystem zusammen mit den anderen Daten auf dem Ticket vermerkt. Die Kontrollstation kann die Kundenpaßnummer anzeigen oder, wenn der Paß ebenfalls maschinenlesbar ist, selbsttätig vergleichen.

Dieser Ansatz ist einfach und einigermaßen elegant, zumal er sich eng an traditionelle Verfahrensweisen anlehnt. Wenn wir einen geeigneten Datenträger finden und die kryptographischen Verfahren keine unerwarteten Schwierigkeiten bereiten, wird er vorzüglich funktionieren. Die Kontrollstation arbeitet autonom; die nötige Liste der kontrollierten Tickets ist bezüglich des Aufwandes nicht mit der Datenbank aus dem vorigen Abschnitt vergleichbar.

Mit einer einzelnen Kontrollstation kommen wir überall dort aus, wo nur wenige, nah beieinander liegende Kontrollstellen nötig sind, zum Beispiel in einem Kino (das auch mehrere Sääle haben kann) oder auf einem Ausstellungsgelände. Die Kommunikationskosten innerhalb der Station sind dann gering; mehr als einige Meter Kabel braucht man nicht.

2.4 Mehrere Kontrollstationen

Mit mehr als einer Kontrollstation haben wir es zu tun, sobald ein Billett an einem beliebigen von mehreren möglichen Orten eingelöst werden kann und die Kommunikation

zwischen diesen Orten aufwendig und teuer ist. Das beste Beispiel dafür ist die Eisenbahn.

Wollte man eine Kontrollstation über mehr als einen Zug erstrecken, brauchte man eine Funkverbindung zu einem Zentralrechner, womit man wieder bei der naiven Datenbanklösung aus dem vorletzten Abschnitt angelangt wäre. Wenn aber zwischen den Zügen kein Datenaustausch möglich ist, wie verhindert man dann, daß eine vierköpfige Familie statt vier verschiedener Fahrkarten vier verschiedene Zugverbindungen für die Fahrt zum Urlaubsort benutzt?

Bisher haben wir nur passive Datenträger benötigt, die Daten transportieren können und sonst nichts. Für sie ist die Antwort einfach: Es geht nicht. Das Problem ist gut erforscht, denn es tritt auch in elektronischen Zahlungssystemen auf. Dort wird ein finanzieller Wert durch eine Bitfolge repräsentiert und man muß verhindern, daß diese Folge kopiert und mehrfach bei verschiedenen Händlern zum Bezahlen eingesetzt wird.

Drei Lösungen sind dafür bekannt. Erstens kann man bei jedem Bezahlvorgang eine Verbindung zu einer Bank aufbauen, wo über die Verwendung der (eindeutig gekennzeichneten) digitalen Zahlungsmittel Buch geführt wird. Das wollen wir hier gerade nicht. Die zweite Möglichkeit sind manipulationssichere Geräte wie etwa die Geldkarte, die seit einiger Zeit von deutschen Banken angeboten wird. Sie hindern den Besitzer am Kopieren des digitalen Geldes.[7] Drittens schließlich können komplizierte kryptographische Protokolle eingesetzt werden, die Betrug zwar nicht verhindern, aber erkennbar machen und die Identität des Betrügers offenbaren. Diese Idee geht auf Chaum, Fiat und Naor zurück [CFN90].

Ein passiver Datenträger genügt für keinen der drei Ansätze. Da das Problem für elektronisches Geld und unsere Fahrkarten im wesentlichen dasselbe ist, können wir guten Gewissens davon ausgehen, daß es keine Lösung gibt, die ohne eine Chipkarte oder einen anderen transportablen Rechner auskommt.

Wann immer jemand sagt: »Geht nicht!« regt sich Protest. Man könnte doch ... Ja, man könnte den Geltungsbereich von Fahrkarten so weit einschränken, daß kein sinnvoller Gebrauch kopierter Tickets möglich ist. Möchte man aber nicht. Das hieße letztendlich – man denke an die vierköpfige Familie – sich auf genau eine Zugverbindung festzulegen. Die spontane Entscheidung, etwas eher oder später zu fahren würde damit ebenso zum Problem wie jeder verpaßte Anschluß. Ja, man könnte den Geltungsbereich ein wenig großzügiger gestalten, wenn man persönliche Fahrscheine ausstellt. Möchte man aber nicht, wenn man sich vor Augen führt, daß man mit dem maschinenlesbaren Reisenden ein viel größeres Problem schafft als man mit dem Fahrkartenverkauf via Internet löst.

Bei Nahverkehrsfahrscheinen stellt sich ein weiteres Problem. Sie werden nur in Stichproben kontrolliert. Angesichts der Seltenheit von Kontrollen in Bus und Straßenbahn hätte der Benutzer etwa einer kopierten Monatskarte gute Chancen, unentdeckt zu bleiben, sofern er sich sein Ticket nicht mit zu vielen anderen teilt. Da die Kontrolle schon bei herkömmlichen Fahrscheinen problematisch ist und sich gewöhnliches Schwarzfahren durchaus lohnen kann, Favorisieren die Verkehrsunternehmen die automatische Bezahlung mittels kontaktloser Chipkarten [Mei97].

[7] Die Geldkarte ist ein schlechtes Beispiel, denn sie basiert auf einem Schattenkonto, das außerhalb des Kartenchips geführt wird. [Sel97]

Die elektronische Fahrkarte trägt einen Chip. Das allein ist ein Grund, sie nicht zu benutzen; noch eine Karte und noch eine und noch eine möchte niemand, und eine Universalkarte ist nicht in Sicht [Riv97]. Auch stellt sich die Frage, wozu eine Fahrkartenkarte überhaupt noch nützlich ist, wenn man auch richtiges Geld auf dem Chip speichern kann. Nur für die Platzkarte? Weitere Nachteile der Chipkarte sind auf Seite 66 aufgezählt. Es sind keine grundsätzlichen Hindernisse. Zieht man aber in Betracht, daß Reisende bei der Deutschen Bahn und vielen Nahverkehrsunternehmen sorglos ihre Fahrt antreten können, ohne sich vorher um eine Fahrkarte zu bemühen, erscheint die Suche nach dem elektronischen Ticket als zweifelhaftes Vergnügen.

2.5 Nachtgedanken

Hastig eingefügt und nicht ganz passend noch einige Gedanken zur digitalen Fahrkarte.

Mit einer Chipkarte als Helfer läßt sich Kopiersicherheit bei gleichzeitiger Anonymität auf ähnliche Weise erreichen wie beim digitalen Geld. Entweder man vertraut darauf, daß die Chipkarte das Kopieren verhindert. Dann ist einzig der Verkaufsvorgang problematisch. Oder aber man betrachtet die Chipkarte lediglich als handlichen Computer und nutzt ihre Fähigkeit, jederzeit Berechnungen ausführen zu können.

Will man den Verkauf gegen unerwünschte Eingriffe absichern, müssen Chipkarte und Verkaufssystem direkt miteinander kommunizieren. Der Computer des Käufers spielt in diesem Fall nur noch als Stromversorgung und Internet-Anschluß der Karte eine Rolle. Verkaufssystem und Karte authentisieren sich zunächst beim jeweils anderen. Das ist mit Public-Key-Kryptographie kein größeres Problem. Das Verkaufssystem verkauft nur an echte Karten und die Karte nimmt nur Tickets von echten Verkaufssystemen an.

Danach können beide Seiten entweder einen Schlüssel für ein symmetrisches Kryptosystem aushandeln oder aber direkt mit dem öffentlichen Schlüssel des Kommunikationspartners arbeiten. Wichtig ist nur, daß sämtliche übertragenen Informationen auch vor dem Debugger eines böswilligen Käufers geschützt sind.

Sind die Ticketinformationen erst auf der Karte, droht ihnen keine Gefahr mehr - es sei denn, der Karteninhaber kann eine Kontrollstation simulieren und so an die Daten gelangen. Auch bei der Kontrolle ist deshalb eine gegenseitige Authentisierung nötig. Die Kontrollstation akzeptiert nur echte Chipkarten und die Karte läßt sich nur von einer echten Kontrollstation prüfen.

Statt auf die Manipulationssicherheit einer Chipkarte zu setzen, kann man auch versuchen, im Betrugsfall den Betrüger dingfest zu machen. Für elektronisches Geld ist dieses Problem gelöst. Das Protokoll von Chaum, Fiat und Naor [CFN90, Wob97] läßt ehrliche Kunden anonym, entlarvt jedoch Betrüger, die ihr elektronisches Geld mehrfach ausgeben, mit hoher Wahrscheinlichkeit. Dadurch ist es offline-fähig, das heißt beim Bezahlen ist keine Datenverbindung zur Bank notwendig. Der Datenaustausch fällt zwar nicht weg, wird aber auf einen späteren Zeitpunkt verlagert.

Unverändert übernehmen läßt sich dieses Protokoll nicht. Es garantiert lediglich, daß die Bank keine Verbindung zwischen einem ausgezahlten Betrag und dem damit getätigten Kauf herstellen kann. Die Information, *daß* der Kunde den Betrag X abgehoben hat,

ist nicht geschützt. Natürlich nicht, denn die Bank muß X vom Kontostand des Kunden abziehen. Das ist beim Geld kein Problem, beim Fahrkartenverkauf schon.

Nutzte man das Protokoll von Chaum/Fiat/Naor für Fahrkarten, so träten an die Stelle des Geldbetrages die Gültigkeitsparameter des Tickets. So erhielte der digitale Verkäufer die Information, daß der Kunde soundso eine Fahrkarte von A nach B gekauft hat, die ab übermorgen gilt. Wann die Reise mit welchem Zug angetreten wirde, könnte niemand feststellen, aber das wäre fast schon ein uninteressantes Detail – der Anbieter erführe von allen seinen Kunden, von wo nach wo sie fahren, und könnte das Reisedatum auch recht weit eingrenzen!

Dem praktischen Einsatz dieses Protokolls steht ein weiteres Hindernis im Wege. Immer noch wären eindeutige Nummern und eine Liste bereits benutzter Tickets nötig. Zwar fiele die Online-Verbindung zur Überprüfung weg, doch die Datenbank selbst ist schon ein Problem. Ein nicht ganz kleines Rechenzentrum nur für die Fahrkartenkontrolle? Das kauft niemand.

Legen wir das Plastik zu den Akten und wenden uns ganz und gar den Eintrittskarten zu, die uns noch genügend Arbeit bereiten werden.

2.6 Parameter

Ein Baustein fehlt in diesem Kapitel noch. Bisher war nur ganz unkonkret von Parametern die Rede, die den Gültigkeitsbereich bestimmen. Wie sie gewählt sein müssen, damit die Kontrollstation über die Gültigkeit befinden kann, ist das Thema dieses Abschnitts.

Die Gültigkeitsprüfung sollte automatisch erfolgen und als Ergebnis eine Ja/Nein-Entscheidung, ein einziges Bit, liefern. Das kostet den Billettabreißer den Arbeitsplatz, denn dieses eine Bit genügt, um eine mechanische Sperre zu öffnen oder gegebenenfalls eben nicht. Die Alternative ist aber auch nicht besser. Werden die Parameter des Tickets lediglich auf einem Bildschirm angezeigt und von einem Menschen geprüft, müßte dessen Aufmerksamkeit abwechselnd dem Bildschirm und dem Eingang gelten. Abzureißen gibt es nichts mehr und so hätte der Kontrolleur einzig die Aufgabe, auf den Schirm zu starren. Nur Betrugsversuche brächten etwas Abwechslung in diese stupide Tätigkeit, indem sie ihn zwängen aufzuspringen den Bösewicht zu ergreifen.

Welche Angaben enthält eine herkömmliche Eintrittskarte? In der Regel:

- den Namen des Veranstalters,
- die Bezeichnung der Veranstaltung,
- Datum und Uhrzeit des Beginns,
- den Preis der Karte oder die Preisklasse sowie
- die Nummer des reservierten Sitzplatzes.

Hinzu kommen:

- verbale Hinweise und

- Gestaltungselemente (z.B. ein Logo des Veranstalters).

Sind alle Angaben vorhanden, kann die Gültigkeit offensichtlich geprüft werden. Werden sie alle benötigt, wenn eine Maschine die Kontrolle übernehmen soll? Nein, denn dazu muß lediglich die Veranstaltung, für die das Ticket gilt, eindeutig bestimmbar sein. Das kann man schon mit einer eindeutigen Veranstaltungsnummer erreichen. Ohnehin muß sich regelmäßig jemand hinsetzen und alle Veranstaltungen in das Verkaufssystem eingeben. Eindeutige Nummern lassen sich dabei problemlos durch die Software erzeugen.

Außerdem kann es für eine Veranstaltung Eintrittskarten zu unterschiedlichen Preisen geben. Die Berechtigung, eine Ermäßigung in Anspruch zu nehmen, kann nur am Veranstaltungsort geprüft werden. Die Preisklasse oder der Kaufpreis muß deshalb ebenfalls angegeben sein. Der Veranstalter steht implizit fest, sobald die Echtheit des Tickets erfolgreich überprüft wurde.

Damit ist ein Parametersatz gefunden, der für die Kontrolle genügt. Das Ticket enthält seine eindeutige Nummer, eine Veranstaltungsnummer sowie eine Kennzeichnung der Preisklasse. Da in praktisch jedem Fall nur wenige verschiedene Preise für eine Veranstaltung vorkommen, genügt die Einteilung in Preisklassen. Der tatsächliche Kaufpreis muß nicht angegeben werden; der Maschine ist egal, welche Zahlen sie vergleicht.

Bei falscher Anwendung ist diese Variante anfällig gegen Replay-Angriffe, das heiß die Wiederverwendung eines alten, schon einmal benutzten Tickets. Die Liste der kontrollierten Tickets in der Kontrollstation kann eigentlich nach der Veranstaltung gelöscht werden. Tut man das aber, und gibt alsbald einer anderen Veranstaltung die gleiche Nummer, können alte Tickets noch einmal verwendet werden.

Die Minimallösung erlaubt zwar die automatische Kontrolle, nicht aber den weitgehend automatischen Betrieb des Gesamtsystems. Für jede Veranstaltung muß vom Verkaufssystem eine Nummer festgelegt und der Kontrollstation mitgeteilt werden. Läßt sich dieser Datenaustausch vermeiden? Dazu müßen beide Seiten, Verkaufssystem und Kontrollstation, unabhängig voneinander einer Veranstaltung denselben Zahlenwert zuordnen.

Das ist einfacher als es sich liest. Man benötigt lediglich eine Uhr auf der Kontrollseite. Beim Verkauf ist der Zeitpunkt des Veranstaltungsbeginns bekannt, denn er wird wenigstens zur Kundeninformation gebraucht. Man kann ihn also ohne Schwierigkeiten als Parameter in das Billett aufnehmen. Die Kontrollstation mit Uhr kennt die aktuelle Zeit. Gültig ist ein Ticket, wenn die aktuelle Zeit innerhalb eines fest gewählten Intervalls um die Zeitangabe des Tickets liegt, etwa eine Stunde vorher bis eine halbe Stunde danach. Der Kontrollstation muß nun nicht mehr mitgeteilt werden, welche Veranstaltungsnummer sie jetzt gerade als gültig ansehen soll. Auch die Dateneingabe für den Verkauf vereinfacht sich, denn jetzt genügen dabei Angaben wie »täglich 20:15 Uhr«.

Die Uhr muß unter allen Umständen die korrekte Zeit liefern, sonst werden echte, gültige Tickets fälschlich zurückgewiesen. Die genaue Uhrzeit steht aber überall zur Verfügung, wo es einen nicht allzu teueren DCF77-Empfänger gibt. Darüber hinaus ist die Lösung unflexibel. Selbst bei großzügig gewählten Intervallen kann eine unvorhergesehene Verschiebung des Beginns Probleme bereiten. Für solche Fälle sollte es möglich sein, der Station von Hand einen Zeitpunkt mitzuteilen, den sie statt der aktuellen Zeit zur

Kontrolle verwendet.

Beginnen, beispielsweise in einem Kino mit mehreren Säälen, mehrere Veranstaltungen zur gleichen Zeit, muß zusätzlich der Veranstaltungsort angegeben sein. Das ist jedoch ein fester Parameter, der bei der Installation für jeden Kontrollpunkt eingestellt werden kann. Auch an dieser Stelle kann man die kryptographische Echtheitsprüfung zurückgreifen, indem man für jeden Ort einen anderen Schlüssel benutzt.

Zu guter Letzt sei noch eine »geschwätzige« Variante diskutiert. Wie anfangs festgestellt, ist die Gültigkeitskontrolle möglich, wenn alle Parameter auf dem Ticket enthalten sind, die auch auf einer traditionellen Eintrittskarte vorkommen. Das ändert sich auch nicht, wenn die Tickets übers Internet vertrieben werden. Neben der Ticketnummer nimmt man die Veranstaltung und der Ort, den Preis oder die Preisklasse, die Anfangszeit sowie die reservierte Platznummer in das Ticket auf.

Veranstaltung und Ort können statt numerisch auch als ASCII-Zeichenfolge kodiert werden. Die Anfangszeit kann man noch um Informationen zum Zeitintervall ergänzen, in dem die Eintrittskarte gültig sein soll. Die Kontrollstation prüft dann Ort und Veranstaltung auf dem Ticket durch Vergleich mit vorgegebenen Werten und die Zeitparameter anhand ihrer Uhr. Weiter kann geprüft werden, ob die Platznummer zur Preisklasse beziehungsweise zum Preis paßt. Bei ermäßigten Tickets signalisiert die Station dem letzten verbliebenen Kontrolleur, daß noch die Ermäßigungsberechtigung zu prüfen ist.

Diese Barockversion der digitalen Eintrittskarte dürfte sich an nahezu jede denkbare Anwendung anpassen lassen, indem man nicht benötigte Parameter wegläßt.

2.7 Binär oder Text?

Auf welche Weise sollen die Parameter in eine Bitfolge verwandelt werden? Grundsätzlich gibt es dafür zwei Möglichkeiten, nämlich »flache« ASCII-Dateien auf der einen und Binärformate auf der anderen Seite [Gan95].

In einem Binärformat werden die Daten im wesentlichen so abgelegt, wie sie auch während der Verabreitung im Hauptspeicher vorliegen. Die Bedeutung einzelner Bits und Bytes ergibt sich, genau wie bei Datenstrukturen im Speicher, implizit aus der Position in der Datei oder im Datenblock. Das spart Platz auf dem Datenträger und Bandbreite bei der Übertragung. Die so abgelegten oder übertragenen Daten lassen sich leicht weiterverarbeiten; im Idealfall können sie in einem (C-)Programm einfach in den Speicher geladen und dort mit den Datenstrukturen des Programms identifiziert werden. Das Speichern oder Senden ist natürlich genauso einfach.

Problematisch wird dieser Ansatz, sobald verschiedene Rechnerplattformen oder auch nur verschiedene Compiler beteiligt sind. Insbesondere Zahlen können auf unterschiedliche Weise im Speicher - und bei naiver Herangehensweise also auch in Dateien - abgelegt werden. Die Programmiersprache C definiert noch nicht einmal die Zahl der Bits, die dafür verwendet werden, ganz zu schweigen von der Abbildung komplexer Datenstrukturen im Speicher. Ganz so einfach ist die Sache denn also doch nicht, aber bei sauberer Formatdefinition und Programmierung sind die Hürden überwindbar. Das Internet funktioniert immerhin.

Binärformate haben einen weiteren Nachteil. Ohne die Formatspezifikation und entsprechenden Code zum Lesen der Daten sind sie weitgehend nutzlos. Man kann sie nicht mit einem gewöhnlichen Editor bearbeiten und auch nicht mit den vielen kleinen Werkzeugen, die die Unix-Welt zu einem digitalen Schlaraffenland machen.

Das ist bei ASCII-Formaten anders. Hier werden Zahlen und andere Daten in lesbaren ASCII-Text umgewandelt und dann gespeichert oder übertragen. Das kann in Form von Name-Wert-Paaren erfolgen (z.B. »From: stuerpe@igd.fhg.de«) oder in starreren Strukturen, zum Beispiel mit Zeilen, die durch ein besonderes Zeichen in eine Anzahl von Feldern aufgeteilt sind. Solche Formate sind leicht zu erzeugen, aber das Lesen erfordert einigen Programmieraufwand. Doch der lohnt sich, denn man erhält ein portables Datenformat, das zudem auch von Menschen gelesen werden kann.

Beispielhaft sind dafür einige Internet-Protokolle der Anwendungsschicht (HTTP, SMTP und andere), die vollständig von einem Menschen mit der Spezifikation unterm Arm abgewickelt werden können. ASCII-Formate benötigen mehr Speicherplatz und Bandbreite, aber das spielt heute nur noch bei wenigen Anwendungen eine Rolle. Sie können mit vielerlei Unix-Werkzeugen bearbeitet werden.

Beispiele für Binärformate sind IP-Pakete und das GIF-Format für Bilddateien, ASCII-Formate findet man bei der E-Mail und XBM (X BitMap).

Die Datenportabilität spielt für unsere Anwendung keine allzu große Rolle. Zwar muss das Verkaufssystem Tickets erzeugen, die die Kontrollstation richtig interpretiert, doch die Datenmenge ist überschaubar. Für Menschen lesbar sein müssen die Daten im maschinenlesbaren Teil des Tickets nicht unbedingt. Der Kunde soll sie ohnehin nicht bearbeiten. Die kryptographischen Methoden aus dem nächsten Kapitel, die ihn an Manipulationen hindern sollen, erzeugen ohnehin Binärdaten. Die kann man zwar auch mit dem Zeichenvorrat[8] des ASCII darstellen, aber dabei vergrößert sich ihr Umfang. Wie wir in Kapitel 4 noch sehen werden, soll das Ticket möglichst wenige Bytes beanspruchen. Aus diesen Gründen werden wir ein Binärformat verwenden.

2.8 Zusammenfassung

Tickets, die nur an einem Ort gültig sind, lassen sich recht einfach digitalisieren, sofern ein geeigneter Datenträger existiert und die Echtheit mit kryptologischen Verfahren gesichert werden kann. Der Datenträger kann in diesem Fall ein passiver sein, denn eine einfache Nachrichtenübertragung genügt. Damit wird auch der Verkaufsvorgang sehr einfach. Nach dem Bezahlen wird das Ticket per HTTP dem Käufer übermittelt. Kryptographische Protokolle sind dafür nicht erforderlich.

Tickets, die an mehreren unabhängigen Kontrollstationen benutzt werden können, benötigen manipulationssichere Geräte oder wenigstens einen aktiven Datenträger. Sie bleiben in den übrigen Kapiteln unberücksichtigt.

Die Rückgabe gekaufter Tickets gegen Erstattung des Kaufpreises ist übrigens nur mit Einschränkungen möglich. Die Entwertung der Tickets erfolgt durch Speicherung ihrer Nummer in der Kontrollstation. Dort muß auch die Rückgabe vermerkt werden, sonst

[8] Eine wunderbare Einführung in die Welt der Zeichensätze und -kodierungen bietet [Kor98].

blieben Kopien des zurückgegebene Tickets weiter benutzbar. Da die Kontrollstation unabhängig vom Verkaufssystem arbeiten soll, ist eine Online-Rückgabe kaum schmerzfrei zu implementieren. Bemüht sich der Kunde hingegen zum Veranstaltungsort, steht der Erstattung des Kaufpreises nichts im Wege.

3 Sicherheit

Schadensbegrenzung Wird immer dann heftig versucht, wenn eh nichts mehr zu retten ist. In letzter Zeit war besonders im Fall Tiedge viel von Schadensbegrenzung zu hören, nachdem der Mann jahrelang rumgesoffen und sich schließlich in die DDR abgesetzt hatte. Besonders perfide war die Verwendung des Wortes im Zusammenhang mit der US-Bombardierung Libyens im April 1986, als nach Hunderten von Toten und Verletzten, nach zerstörten Häusern und Besitz noch irgendein »Schaden begrenzt« werden sollte.
(Eckhard Henscheid: Dummdeutsch)

Die eben diskutierten Ansätze setzen allesamt voraus, daß nur der Verkäufer in der Lage ist, Tickets zu erzeugen, und daß nachträgliche Veränderungen nicht unbemerkt möglich sind. Wie das erreicht werden kann, untersucht dieses Kapitel. Eine Möglichkeit sind manipulationssichere Geräte. Bis auf die Frage, wie ein Ticket übers Netz in das Gerät kommt, sind sie uninteressant, denn sie verhindern per definitionem unerlaubte Zugriffe. Zudem genügt für Eintrittskarten offenbar ein passiver Datenträger, der lediglich eine Nachricht übermittelt und zu keiner Interaktion fähig ist. Aus diesem Grund werden hier nur Verfahren zur Nachrichtenauthentifikation betrachtet.

3.1 Authentifikation

> »In the broadest sense, authentication is concerned with establishing the integrity of information purely on the basis of the internal structure of the information itself, irrespective of the source of that information.« [Sim92]

Die Kryptologie führt die Sicherheit von Nachrichten, sei es vor unbefugter Kenntnisnahme oder oder vor Veränderung, darauf zurück, daß ein kleines Geheimnis bewahrt wird, der kryptographische Schlüssel [MOV96]. Die Techniken der Kryptologie benutzen dieses Geheimnis, um die Welt in zwei ungleiche Hälften zu spalten. Die Besitzer des Schlüssels können mit dessen Hilfe Operationen ausführen, die allen anderen versagt bleiben, etwa den Klartext einer Nachricht lesen oder eine Nachricht erzeugen, die von anderen als authentisch erkannt wird.

Für Anwendungen im richtigen Leben genügt praktische Sicherheit: Der Weg, eine kryptographische Operation ohne den zugehörigen Schlüssel auszuführen, ist wohlbekannt, wird aber durch den Rechenaufwand versperrt, der ohne Schlüssel nicht zu bewältigen ist.

Symmetrische Verfahren setzen voraus, daß Sender und Empfänger einer Nachricht denselben Schlüssel kennen. Irgendwie müssen sie es schaffen, sich auf einen zu einigen, ohne daß er Dritten zur Kenntnis gelangt. (Auch dafür liefert die Kryptologie interessante Methoden, zum Beispiel das Diffie-Hellmann-Protokoll zum Schlüsselaustausch. [Beu94])

Da niemand sonst den Schlüssel besitzen soll, ist der Sender sicher, daß seine Nachrichten nur von ihm selbst und vom beabsichtigten Empfänger gelesen werden können, und der Empfänger weiß, daß nur er oder der Sender eine Nachricht erzeugt haben kann.

Dagegen verlangen *asymmetrische* Verfahren die Geheimhaltung nur auf einer Seite. Schlüssel treten paarweise auf und bestehen jeweils aus einem geheimen und einem zugehörigen öffentlichen Teil. Der Empfänger und niemand sonst kann mit dem geheimen Schlüssel Nachrichten lesen, die der Sender mit dem dazu passenden öffentlichen Schlüssel verschlüsselt hat. Nur der Sender ist mit seinem geheimen Schlüssel in der Lage, einer Nachricht Authentizitätsmerkmale hinzuzufügen, die der Empfänger mit dem zugehörigen öffentlichen Schlüssel prüft. Zwar hängen die Schlüssel voneinander ab, doch aus dem öffentlichen Teil kann der geheime praktisch nicht ermittelt werden, so daß das Geheimnis gewahrt bleibt.

Umfassende wie tiefschürfende Literatur zur Kryptologie gibt es reichlich. Empfohlen seien hier die allgemeinverständliche Einführung von Beutelspacher [Beu94], das Schneiers Kompendium [Schn96] sowie das Handbuch von Menezes, van Oorschot und Vanstone [MOV96]. Einen vorzüglichen Überblick über Authentifizierungstechniken liefert Simmons [Sim92], und mit dem Brechen von Kryptosystemen, der Kryptoanalyse, beschäftigt sich das Werk von Wobst [Wob97]. Allgemeine Ausführungen zur Kryptographie und ihren Verfahren, die nicht mit einer Quellenangabe versehen sind, haben vermutlich aus einem dieser Werke ihren Weg ins Hirn des Autors gefunden.

3.1.1 Der Betrüger

Bevor wir uns den drei Lösungsmöglichkeiten widmen, die die Kryptologie für unser Authentifikationsproblem bereithält, sind einige Überlegungen über den Gegner nötig. Der ist eine unehrliche Person mit dem Ziel, in der Rolle des Besuchers eine Leistung zu erhalten, die sie so nicht bezahlt hat. Der Aufwand, den er für seinen Betrug treiben wird, ist schwer zu schätzen.

Der Ticketpreis setzt dem finanziellen Aufwand eine Grenze [Bra98], denn ein Betrug, der teurer ist als Ehrlichkeit, lohnt sich allenfalls des sportlichen Ehrgeizes wegen. Das ist jedoch mit einer gehörigen Portion Vorsicht zu genießen, denn unser Betrüger hat möglicherweise gar nicht den finanziellen Vorteil im Auge, sondern möchte zum Beispiel unbedingt das Endspiel einer Fußball-WM sehen, für das er keine Karte mehr bekommen hat. Es darf auch gern den dreifachen Eintrittspreis kosten, wenn er nur irgendwie ins Stadion kommt.

Der finanzielle Aufwand wird dadurch relativiert, daß dem Betrüger unter Umständen Ressourcen kostenlos zur Verfügung stehen, obwohl sie teuer sind. Wer moderne Kryptosysteme angreift, braucht vor allem Rechenleistung und Speicher. Beides bekommt heute jeder Student in großem Umfang für exakt 0,00 Euro. Der Autor sitzt, während er diese Sätze schreibt, in einem Raum mit 15 leistungsfähigen Computern, die alle nichts tun als auf Benutzer zu warten. Über das unscheinbare Kabel, das seinen Arbeitsplatz mit dem Rest der Welt verbindet, hat er Zugang zu einer dreistellige Zahl weiterer Geräte. Zu einem Parallelrechner verbunden bieten sie ihm eine Rechenleistung, von der er vor einigen Semestern noch nicht mal geträumt hat.

Mehr noch als der finanzielle sperrt sich der Zeitaufwand gegen Quantifizierungsversuche. Zeiten kann man angeben, aber man kann sie kaum bewerten. Vier Wochen sind viel, rechnet man sie nach üblichen Stundensätzen in Geld um. Sie sind vernachlässigbar, wenn sich ein Student damit die Semesterferien vertreibt und das auch noch als Chance zum Lernen nimmt. Das wiegt schwer, weil fehlende Prozessorleistung in Grenzen durch größeren Zeitaufwand ersetzbar ist. Bei Angriffen auf kryptographische Schlüssel muß die Zeit zudem nur einmal investiert werden, während das Ergebnis mehrfach genutzt werden kann [Bra98], sofern die Schlüssel nicht sehr häufig geändert werden.

Angriffen auf kryptographische Schlüssel setzt die Nutzungsdauer des Schlüssels eine zeitliche Grenze, solchen auf einzelne Tickets der Zeitpunkt der Veranstaltung.

Obgleich sie nicht wörtlich zitiert sind, haben sich schon andere solche Gedanken gemacht. Auf der sicheren Seite ist man nach derzeitigem Forschungsstand mit Schlüssellängen von 90 Bits und mehr [Bla96, Schn96], wenn der benutzte Algorithmus keine größeren Schwächen aufweist. Der Aufwand eines Brute-Force-Angriffs, bei dem säemtliche Schlüssel durchprobiert werden, bis der richtige gefunden ist, liegt dann in einer Größenordnung, die nicht nur Eintrittskartenfälscher vor Probleme stellt.

Daß der Gedankengang hier dennoch dargestellt ist, hat einen politischen Hintergrund. Er ist ein Argument für starke Kryptographie und vor der hat mancher Angst. So gibt es seit einigen Jahren Forderungen, dem Normalbürger Kryptographie nur noch dann zu erlauben, wenn sie entweder nichts taugt, oder die geheimen Schlüssel bei staatlichen Stellen hinterlegt werden, was ebenfalls bedeutet, daß die Kryptographie nichts taugt, denn die geheimen Schlüssel müssen genau das sein: geheim. Für sich selbst und seine Geheimdienste soll der Staat selbstverständlich eine Ausnahme machen. Bis jetzt gelten in der Bundesrepublik keine Einschränkungen und auch konkrete Gesetzesvorlagen sind noch nicht in der Öffentlichkeit aufgetaucht. Die Gefahr ist jedoch latent vorhanden, worauf der Autor ausdrücklich hinweisen möchte. Für die weitere Beschäftigung mit dem Thema bietet www.crypto.de [Rei98] mit zahlreichen Hyperlinks einen guten Ausgangspunkt.

Zurück zum Betrüger. Er investiert also vielleicht deutlich mehr in seinen Angriff, als man bei flüchtiger Betrachtung meint, und starke Kryptographie bietet auch dagegen Schutz. Was ist noch über ihn bekannt? Dem Angreifer wird unterstellt, daß er alles kennt bis auf die Schlüssel, die ausdrücklich geheimgehalten werden:

»**Das Prinzip von Kerkhoffs.** Die Sicherheit eines Kryptosystems darf nicht von der Geheimhaltung des Algorithmus abhängen. Die Sicherheit gründet sich nur auf die Geheimhaltung des Schlüssels.« [Beu94]

Das gilt auch für alle anderen Systemkomponenten, die zusammen mit dem Kryptosystem eingesetzt werden. Warum ist diese Annahme vernünftig? Sie schadet jedenfalls nicht.[9] Weiß der Angreifer weniger als angenommen, macht ihm das die Arbeit jedenfalls nicht leichter. Kann er aber umgekehrt etwas in Erfahrung bringen, von dessen Geheimhaltung die Sicherheit abhängt, muß dieses Etwas ausgetauscht werden, denn das Wissen ist nicht

[9] Famous last words?

rückholbar. Für die kurze Bitfolge des Schlüssels ist das einfach, für andere Komponenten, etwa den Algorithmus oder seine Implementierung, ist es das nicht.

Die Anwendung Eintrittskartenverkauf legt eine weitere Annahme über das Wissen des Betrügers nahe. Er soll auch alle Parameter kennen, die in das Ticket eingehen. Wie sie kodiert sind, ist ihm – Kerkhoffs' Prinzip auf das Gesamtsystem angewandt – ebenfalls bekannt. Auch diese Annahme ist vernünftig. Die meisten Parameter wählt der Kunde ohnehin beim Kauf selbst. Die eindeutige Ticketnummer könnte man geheimhalten, aber dann müßte sie so erzeugt werden, daß sie nicht leicht zu erraten ist.

Eine letzte Annhame über den Betrüger betrifft die Zahl der Tickets, die er sich verschaffen kann. Sie ist von Bedeutung, weil Angriffe auf einen Schlüssel tendenziell umso einfacher werden, je mehr damit gesicherte Nachrichten einem Angreifer in die Hände fallen. Zum einen kann eine Betrüger einzelne Tickets kaufen. Dabei hat er die Möglichkeit, den Inhalt teilweise zu beeinflussen, indem er Parameter wie die Veranstaltung oder das Datum wählt. Allzu viele Nachrichten wird er sich auf diese Weise nicht verschaffen, denn eigentlich möchte er betrügen und nicht den Veranstalter reich machen. Auf einem anderen Weg kann er mit weniger Aufwand mehr erreichen: Benutzte Eintrittskarten werden weggeworfen, und das oft in der Nähe des Veranstaltungsortes. Je nach Besucherzahl und seiner Geduld kann ein Betrüger einige hundert, vielleicht auch einige tausend Tickets aus Papierkörben und vom Boden aufsammeln. Vermutlich wird ihm das zu mühsam sein, doch vorsichtshalber sei angenommen, daß er es versucht.

Wie kann der (bis jetzt nur potentielle, aber zu allem entschlossene) Betrüger nun daran gehindert werden, seine Schandtat zu vollbringen?

3.1.2 Symmetrische Verschlüsselung

Ein symmetrisches Kryptosystem besteht aus einer Verschlüsselungsfunktion E_K und einer Entschlüsselungsfunktion D_K (E steht für encrypt, D für decrypt). Beide hängen vom Schlüssel (Key) K ab; jeder Schlüssel liefert ein anderes Funktionspaar. Um eine Nachricht M (wie Message, oft auch als Klartext bezeichnet) zu verschlüsseln, wird wird der Funktionswert $C = E_K(M)$ bestimmt. Das ist der Geheimtext (engl. Ciphertext). Der Empfänger gewinnt daraus durch Anwendung der Entschlüsselungsfunktion die Klartextnachricht $M = D_K(C)$ zurück.

Es gibt zwei Arten symmetrischer Kryptosysteme, *Stromchiffren* und *Blockchiffren*. Stromchiffren behandeln den Klartext und den Geheimtext als eine Folge von Zeichen $M = m_1 m_2 \ldots m_n$ und $C = c_1 c_2 \ldots c_n$, meist einzelnen Bits. Der Schlüssel dient zur Initialisierung eines Pseudozufallszahlengenerators, der eine weitere Zeichenfolge $k_1 k_2 \ldots k_n$ liefert. Zum Verschlüsseln wird diese Folge zeichenweise mit dem Klartext verknüpft, etwa durch Addition modulo 2: $E_K(M) = E_K(m_1 m_2 \ldots m_n) = (m_1 \oplus k_1)(m_2 \oplus k_2) \ldots (m_n \oplus k_n) = c_1 c_2 \ldots c_n = C$. Analog erfolgt die Entschlüsselung $D_K(C) = D_K(c_1 c_2 \ldots c_n) = (c_1 \ominus k_1)(c_2 \ominus k_2) \ldots (c_n \ominus k_n) = m_1 m_2 \ldots m_n = M$ Im Falle der Addition modulo 2 sind die Operationen $\oplus$ und $\ominus$ identisch. Details und Variationen sind im Moment nicht von Bedeutung. [Beu94]

Blockchiffren arbeiten mit Blöcken von mehreren Zeichen; üblich sind 64 Bit, also 8 Bytes [RLab98]. Die Nachricht M wird solche Blöcke unterteilt und auf jeden

Block die Verschlüsselungsfunktion $\overline{E_K}$ angewendet, die einen Klartextblock in einen Geheimtextblock überführt. Ergebnis ist eine Folge von Geheimtextblöcken $E_K(M) = E_K(m_1 m_2 \ldots m_l) = \overline{E_K}(m_1)\overline{E_K}(m_2)\ldots\overline{E_K}(m_l) = c_1 c_2 \ldots c_l = C$. Der Empfänger verfährt mit diesem Geheimtext C und der Blockentschlüsselungsfunktion $\overline{D_K}$ genauso und erhält $D_K(C) = D_K(c_1 c_2 \ldots c_l) = \overline{D_K}(c_1)\overline{D_K}(c_2)\ldots\overline{D_K}(c_l) = m_1 m_2 \ldots m_l = M$. Die Blockverschlüsselungsfunktion $\overline{E_K}$ ist eine Bijektion, sonst wäre keine eindeutige Entschlüsselung möglich. Bei einer Blocklänge von b Bits gibt es $2^b!$ Bijektionen, die Klartextblöcke auf Geheimtextblöcke gleicher Länge abbilden. Jede davon entspricht einem möglichen Schlüssel. In der Praxis liefert ein Algorithmus eine Auswahl von beispielsweise 2^{64} oder 2^{128} Bijektionen und ordnet jeder einen Schlüssel von 64 beziehungsweise 128 Bit Länge zu. [MOV96]

Symmetrische Verschlüsselungsverfahren werden mit dem Ziel konstruiert, Informationen geheimzuhalten. Sie sind nicht dazu gedacht, die Integrität der übermittelten Nachrichten zu sichern. Können sie trotzdem auch dafuer gebraucht werden?

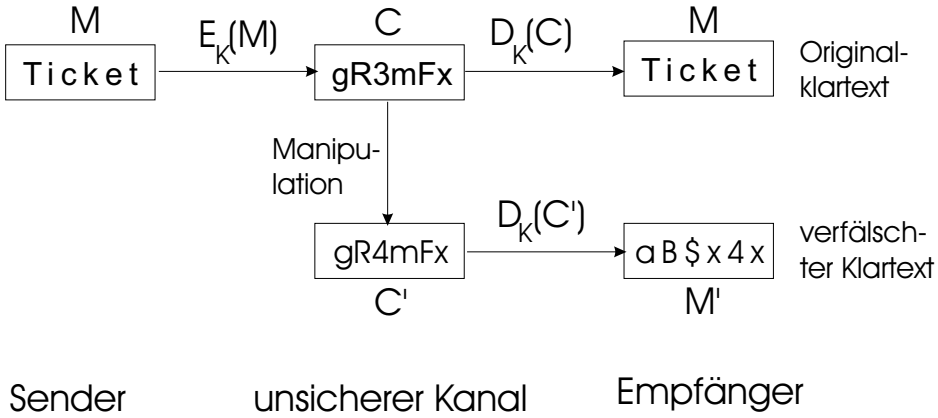

Abb. 3.1: *Bemerkt der Empfänger die Manipulation?*

Augenscheinlich nicht, wenn der Empfänger vor dem Entschlüsseln keinerlei Informationen über den Klartext besitzt. Der Sender verschlüsselt eine Nachricht $E_K(M) = C$ und übermittelt C auf einem unsicheren Kanal. Verändert ein Außenstehender während der Übertragung den Geheimtext C in C', erhält der Empfänger beim Entschlüsseln eine modifizierte Nachricht $M' = D_K(C')$. Das kann er aber nicht bemerken, wenn er jede beliebige Nachricht akzeptiert.

In realen Anwendungen kommen selten beliebige Nachrichten vor.[10] Protokolle definieren Datenstrukturen und Menschen tauschen natürlichsprachige Nachrichten per E-Mail aus. Die Geheimhaltung wird dadurch schwieriger, weil Kryptoanalytiker Anhaltspunkte bekommen, wie das Ergebnis ihrer Arbeit aussehen wird. Dafür gibt es dem

[10] Selten heißt selten und nicht niemals. Der Austausch kryptographischer Schlüssel ist das beste Beispiel für eine Situation, in der der Empfänger Zufallszahlen erwartet. Daß das zum Problem werden kann, wenn Protokolle zum Schlüsselaustausch ungenau spezifiziert sind, zeigen Mao und Boyd anhand einiger ISO-Standards [MaBo94].

Empfänger verschlüsselter Kommunikation die Möglichkeit, Modifikationen zu erkennen, sofern sie die festgelegten Datenstrukturen verändern und dadurch ungültig machen. So könnte die Nachricht etwa eine CRC-Prüfsumme zur Fehlererkennung tragen, die ungezielte Veränderung mit hoher Wahrscheinlichkeit anzeigt. Gezielte Manipulationen, mit denen ein Angreifer jedes einzelne Bit einer Nachricht in einen gewählten Wert ändern kann, sind ohne Schlüssel nicht möglich, denn dann wäre er im Besitz der Funktion E_K und damit des geheimen Schlüssels.

Muß der Angreifer aber überhaupt so gezielt manipulieren? Nein, und unsere Eintrittskarten sind das beste Beispiel dafür! Sie tragen als Parameter eine Nummer, von der nichts verlangt wird als daß sie noch auf keinem vorher kontrollierten Ticket gestanden hat. Ein Betrüger, der eine gekaufte Karte vervielfältigen möchte, muß dazu nur diese Nummer verändern. Er kennt nach Voraussetzung das gesamte System mit Ausnahme der Schlüssel, also weiß er, welche Bits des Klartextes er verändern muß. Bei einer Stromchiffre kann er sie im Geheimtext lokalisieren und bei einem Blockverfahren wenigstens noch den oder die Blöcke ermitteln, die die Ticketnummer oder Teile davon enthalten.

Bei Stromchiffren genügt ihm das. Er ersetzt die betreffenden Bits im Geheimtext einfach durch zufällig gewählte Werte. Beim Entschlüsseln wird daraus eine andere, ebenfalls zufällige Bitfolge und an allen anderen Parametern ändert sich nichts. Mit etwas Glück hat der Betrüger auf diese Weise eine Ticketnummer generiert, die noch nicht in der Liste der Kontrollstation steht – und besitzt nun zwei gültige Eintrittskarten zum Preis von einer. Man kann das ein wenig erschweren, indem man für die Ticketnummer nur die unbedingt notwendige Anzahl von Bits benutzt, aber damit landet man alsbald bei einer Numerierung, die für jede Veranstaltung bei Null beginnt. Das gefällt dem Betrüger, denn wenn er als Erster am Einlaß erscheint, steht seine Zufallszahl sicher noch nicht in der Liste und er kann auch noch zuschauen, wie später ein ehrlicher Kunde des Betrugs bezichtigt wird, weil seine Ticketnummer bereits vom Betrüger »verbraucht« wurde.

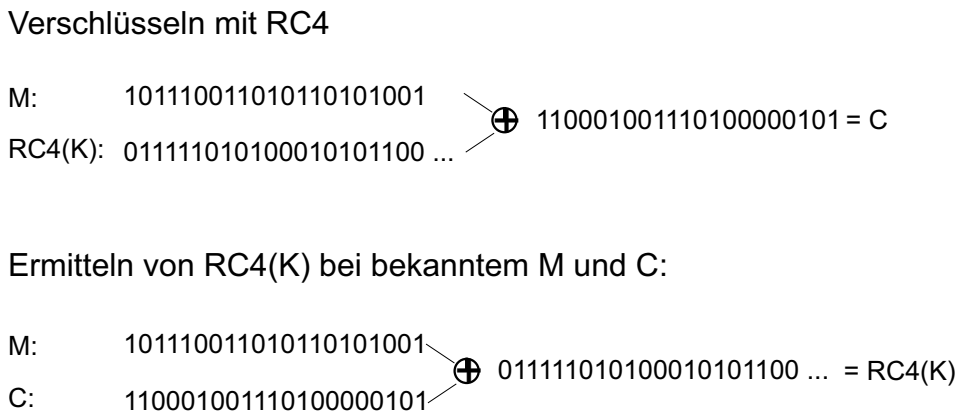

Abb. 3.2: *Stromchiffren schützen nicht vor Manipulation*

Mit einer Stromchiffre geht es also nicht. Zwar kann man sie so gestalten, daß der Wert eines einzelnen Bits beim Ver- oder Entschlüsseln auch die Werte nachfolgender Bits

beeinflußt. Mit der Ticketnummer würde der Betrüger dann gleichzeitig auch nachfolgende Daten ändern. Doch wenn das Verfahren erst modifiziert werden muß, um überhaupt brauchbar zu sein, kann man es getrost beiseite legen, zumal kein Mangel an Alternativen besteht. Auch wurde ein anderer erheblicher Mangel noch gar nicht erwähnt. Häufig wird der Schlüsselstrom durch bitweise Addition modulo 2 mit dem Klartext verknüpft. Ist der Klartext bekannt, läßt sich der benutzte Teil des Schlüsselstroms leicht ermitteln. Das liefert zwar noch nicht den Schlüssel selbst, genügt jedoch, um eine Nachricht gleicher Länge zu verschlüsseln.

Bleiben die Blockchiffren. Da hat es der Angreifer etwas schwerer. Er kann innerhalb des Blocks keine einzelnen Bitpositionen manipulieren. Nichts hindert ihn aber daran, einzelne Blöcke einer Nachricht mit Zufallszahlen zu überschreiben. Er kann auch Blöcke innerhalb der Nachricht vertauschen. Selbst gezielte Veränderungen sind leicht möglich, wenn mehrere Nachrichten mit demselben Schlüssel gesichert wurden. Dann kann der Betrüger einen Block aus einer Nachricht (deren Klartext er in unserem Fall kennt!) durch einen aus einer anderen ersetzen.

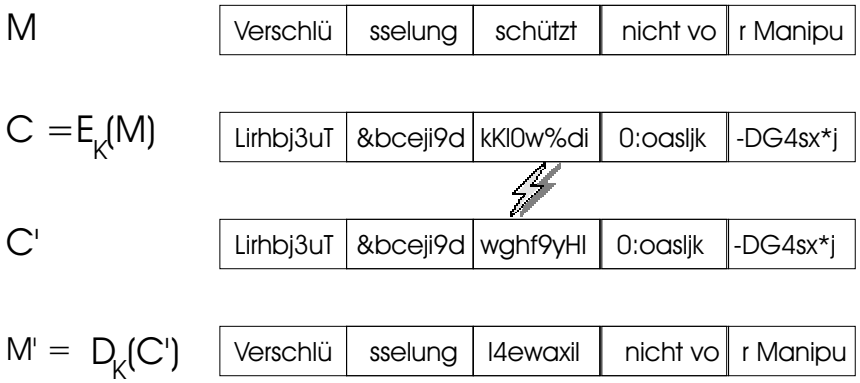

Abb. 3.3: *Veränderung eines Geheimtextblockes in einer Nachricht*

Und was ist mit kurzen Nachrichten, so kurz, daß ein einziger Block ausreicht? Im 2. Kapitel wurde unter anderem ein Parametersatz vorgeschlagen, der neben der Ticketnummer die Anfangszeit der Veranstaltung enthält und vielleicht noch eine Preisklasse. Das paßt gut in acht Bytes. Findet ein Betrüger keine anderen Lücken, kann er es immer noch versuchen, ein Ticket zu erraten. Das macht wenig Arbeit und wenn er falsch geraten hat, muß er nur schnell genug weglaufen.

Mit welcher Wahrscheinlichkeit ist eine zufällig gewählte Nachricht dieser Länge eine gültiges Eintrittskarte? Der Einfachheit halber sei angenommen, daß nur die Ticketnummer und die Anfangszeit verwendet werden. Für Zeitangaben kann man eine Darstellung benutzen, die in der Unix-Welt als `time_t`[11] bekannt ist. Bleiben vier Bytes, die die Ticketnummer enthalten sollen.

[11] Der Typ `time_t` ist unter den gängigen Unices ein 32-Bit-Integer-Wert. Er gibt einen Zeitpunkt in Sekunden an, die seit 1970-01-01 00:00:00Z vergangen sind. Der Wertebereich genügt bei 32 Bit bis zum Januar des Jahres 2038.

Bei einer Blocklänge von 64 Bit gibt es 2^{64} verschiedene Geheimtextblöcke c. Jedem wird durch die Entschlüsselungsfunktion ein anderer Klartextblock $m_c = \overline{D_K}(c)$ zugeordnet. Welcher das ist, hängt vom Schlüssel ab. Der Betrüger wählt einen Block c. Schlimmstenfalls ist die Nummernliste der Kontrollstation leer. Dann wird jede Nummer akzeptiert. Die 32 Bits, die die Ticketnummer enthalten, können also beliebig sein. Das Problem reduziert sich darauf, einen Geheimtextblock c zu finden, der beim Dechiffrieren in den übrigen 32 Bit den richtigen Wert liefert. Wäre dafür genau eine Bitfolge zugelassen, fände der Betrüger mit einer Wahrscheinlichkeit von $1 : 2^{32} = 1 : 4294967296$ durch bloßes Raten einen geeigneten Geheimtextblock.

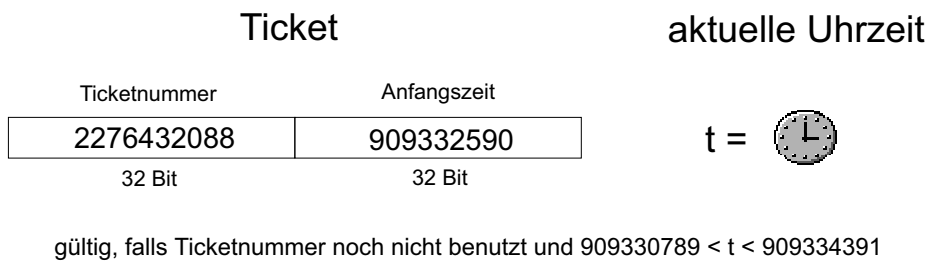

Abb. 3.4: *Zufällig gewähltes Ticket*

Die vorgeschlagene Kontrollprozedur für solche Tickets fordert lediglich, daß die aktuelle Uhrzeit bei der Kontrolle in einem Intervall um den Veranstaltungsbeginn liegt, der auf dem Billett angegeben ist. Ein sinnvolles Zeitintervall kann in einem Kino beispielsweise eine halbe Stunde, 1800 Sekunden, vor Vorstellungsbeginn anfangen und eine halbe Stunde danach enden. Der ratende Betrüger erscheint zu einem bestimmten Zeitpunkt am Veranstaltungsort und muß eine passende Ticketnachricht finden.

Passend ist das Ticket dann, wenn sich die darin enthaltene Anfangszeit um nicht mehr als 1800 Sekunden vom Kontrollzeitpunkt unterscheidet. Das ist mit Sicherheit der Fall, wenn sich die `time_t`-Darstellungen beider Zeiten in wenigstens einem der 21 höchstwertigen Bits unterscheiden, denn dann ist die Differenz auf jeden Fall grösser als $2^{11} - 1 = 2047$. (Der Betrüger rechnet umgekehrt: Unterscheiden sich lediglich die zehn niederwertigen Bits, ist die Differenz kleiner als $2^{10} = 1024$ und er er kann bestimmt ins Kino gehen.)

Nur die 21 höchstwertigen Bits muß der Betrüger also garantiert richtig raten, um sich ein gültiges Ticket zu verschaffen. Bei zufälliger Wahl kann er das mit einer Wahrscheinlichkeit von $1 : 2^{21} = 1 : 2097152$ bewerkstelligen. Mit einfachem Raten kommt er in der vorausgesetzten Konstellation also nicht sehr weit. Die Chance, zu einer gestohlenen EC-Karte die vierstellige Geheimzahl zu erraten[12] ist deutlich besser, und im Erfolgsfall winkt Bargeld statt eines Kinobilletts im Wert von dreizehn Mark.

Die kleine Überschlagsrechnung zeigt zweierlei. Grundsätzlich sind symmetrische Blockchiffren geeignet, eine kurze Nachricht vor Verfälschung zu schützen. Das funktio-

[12] Theoretisch 1:3000 bei drei Versuchen und PINs von 1000 bis 9999, praktisch für ältere Karten 1:682 ohne und 1:150 mit Kenntnis der Daten auf dem Magnetstreifen. [Moe97, Kuhn97]

niert jedoch nur, wenn der Empfänger einige Bits der Nachricht schon kennt. Die Anzahl der bekannten Bits induziert eine obere Schranke der Manipulationssicherheit, indem sie die Erfolgswahrscheinlichkeit eines Angriffs durch einfaches Raten festlegt.

Der Versuch, eine gültige Nachricht zu erraten, ist einfach und läßt sich nicht verhindern. Welches Risiko davon ausgeht, kann man ausrechnen oder schätzen. Bisher haben wir angenommen, daß es keinen einfacheren Angriff gibt. Ist diese Annahme gerechtfertigt?

Die benutzte Blockchiffre muß unanfällig gegen Angriffe mit gewähltem Klartext sein. Ein Betrüger kann beim Ticketkauf die Parameter zumindest teilweise selbst bestimmen. Vermutlich genügt ihm das nicht, aber sicher ist sicher und die Nebenwirkung ist ohnehin erwünscht: gegen Angriffe mit lediglich bekanntem Klartext, die unsere Anwendung ermöglicht, ist das Verfahren dann auf jeden Fall resistent [MOV96]. Moderne Algorithmen sind mit dem Ziel konstruiert, solchen Angriffen standzuhalten; ein starkes Verfahren lässt nur Angriffe zu, die genauso schwer sind wie das Durchsuchen des gesamten Schlüsselraumes [Bla96]. unabhängig vom eingesetzten Verschlüsselungsalgorithmus gilt:

> »It is also believed that forging messages is as difficult as breaking the cipher, and that frequent key replacements increase the security of the cipher, or at least cannot reduce its strength. In this paper we show that all these beliefs are wrong.« – Eli Biham [Bih96].

So schlimm, wie man nach diesem Zitat glauben könnte, ist es allerdings nicht. Bihams Report befaßt sich mit Situationen, in denen derselbe (bekannte) Klartextblock mit vielen verschiedenen Schlüsseln verschlüsselt wird. »Viele« sind für den dort beschriebenen Angriff $2^{k/2}$ Geheimtexte bei einer Schlüssellänge von k Bits. Der Angriff liefert dann *einen* der $2^{k/2}$ Schlüssel, aber nicht sicher, sondern lediglich mit hoher Wahrscheinlichkeit. Dazu sind $2^{k/2}$ Berechnungen nötig. Auch mit weniger Geheimtexten führt er zum Erfolg, aber dann steigt die Komplexität. Stehen umgekehrt noch mehr Geheimtexte zur Verfügung, so werden auch mehr Schlüssel aufgedeckt und der Rechenaufwand je gefundenem Schlüssel verringert sich. Der Angriff besteht einfach darin, daß ein Klartext, etwa ein immer wieder auftretender Kommunikationsheader, mit $2^{k/2}$ zufälligen Schlüsseln verschlüsselt wird. Die so erhaltenen Klartexte speichert der Angreifer zusammen mit den Schlüsseln in einer Tabelle. Jeder abgefangene Geheimtext kann nun einfach in der Tabelle gesucht werden. Ist er darin enthalten, liefert das sofort den zugehörigen Schlüssel.

Für das elektronische Billett stellt der Kollisionsangriff keine wirkliche Gefahr dar. Kryptosysteme mit weniger als 64 Bit Schlüssellänge möchte man ohnehin nicht mehr benutzen[13], so daß ein Betrüger nicht nur 2^{32} Tickets brauchte, sondern auch alle mit gleichem Inhalt, aber mit verschiedenen Schlüsseln verschlüsselt.

Hier aufgeführt ist Bihams Angriff, weil er unabhängig vom benutzten Verschlüsselungsalgorithmus funktioniert. Zudem zeigt er beispielhaft, wie der konkrete Einsatz eines

[13] Im Juli 1998 gelang es der Electronic Frontier Foundation (EFF), mit einem Spezialrechner im Wert von knapp 250.000 Dollar einen 56-Bit-DES-Schlüssel in nur drei Tagen zu finden [EFF98a, EFF98b]. Es wird nicht mehr lange dauern, bis die nötige Rechenleistung jedem zur Verfügung steht.

Kryptosystems – in diesem Fall die Verschlüsselung von immer gleichen Kommunikationsheadern mit verschiedenen Schlüsseln – die Sicherheit des Systems beeinflussen kann. Schlimmstenfalls lassen sich mehrere Schwachstellen gemeinsam ausnutzen und liefern zusammen einen einfachen Angriff.

Algorithmus	Blocklänge	Schlüssellänge	Bemerkungen
DES	64 Bit	56 Bit	zu kurze Schlüssel [EFF98a, EFF98b]
IDEA	64 Bit	128 Bit	patentiert; nichtkommerzielle Nutzung frei
RC5	32, 64 und 128 Bit	bis 2048 Bit	beschrieben in [BaRi96]
Blowfish	64 Bit	bis 448 Bit	beschrieben in [Schn96]

Tab. 3.1: *Einige Blockchiffren*

Abschließend sind in Tabelle 3.1 einige verbreitete Blockchiffren aufgeführt. Einen umfassenderen Überblick enthält [MOV96]. Eine Reihe von Neu- und Weiterentwicklungen bewerben sich derzeit darum, als Advanced Encryption Standard (AES) zum DES-Nachfolger zu werden [NIST98]. Bis der Sieger feststeht, wird noch einige Zeit vergehen.

3.1.3 Codes zur Nachrichtenauthentifikation

Veränderungen an sehr kurzen Nachrichten können mit symmetrischen Blockchiffren erkennbar gemacht werden, wenn die Nachricht genügend Redundanz enthält. Bei längeren Nachrichten scheitert das daran, daß die Blöcke unabhängig voneinander behandelt werden. Das läßt nicht nur Manipulationsmöglichkeiten, sondern gefährdet auch die Geheimhaltung, zu der sie normalerweise eingesetzt werden. Kommt ein Block im Klartext mehrfach vor, enthält auch der Geheimtext an den entsprechenden Positionen identische Blöcke. Ein Angreifer erhält daraus Informationen über den Klartext, bevor er überhaupt richtig mit seiner Arbeit begonnen hat.

Um das zu verhindern, kann ein Blockalgorithmus in verschiedenen Betriebsarten eingesetzt werden [MOV96, Schn96]. Die einfachste, der Electronic Codebook Mode (kurz ECB), wurde bereits im vorigen Abschnitt vorgestellt. Drei weitere sind verbreitet. Der Cipher Feedback Mode (CFB) und der Output Feedback Mode (OFB) machen im wesentlichen eine Stromchiffre aus dem Blockalgorithmus [Schn96, MOV96] und sind damit für die Nachrichtenauthentifizierung uninteressant. Der dritte Modus, das Cipher Block Chaining (CBC), läßt die Blockverschlüsselung selbst unangetastet, verknüpft jedoch den i-ten Klartextblock m_i vorher mit dem $(i-1)$-ten Geheimtextblock c_{i-1}. Beim ersten Block wird statt dessen ein zufälliger Initialisierungsvektor iv benutzt, der nicht geheim bleiben muß. Mit Bezeichnungen wie in 3.1.2 und $\oplus$ als bitweiser Addition modulo 2 verläuft die Verschlüsselung im CBC-Modus gemäß $E_K(M) = E_K(m_1 m_2 \ldots m_l) = \overline{E_K}(m_1 \oplus iv)\overline{E_K}(m_2 \oplus c_1) \ldots \overline{E_K}(m_l \oplus c_{l-1}) = c_1 c_2 \ldots c_l = C$ und das Entschlüsseln entsprechend $D_K(C) = D_K(c_1 c_2 \ldots c_l) = iv \oplus \overline{D_K}(c_1)(c_1 \oplus \overline{D_K}(c_2)) \ldots (c_{l-1} \oplus \overline{D_K}(c_l)) = m_1 m_2 \ldots m_l = M$. Nun hängt jeder Block c_i des Geheimtextes vom i-ten Block m_i des

Klartextes und allen vorausgehenden Klartextblöcken ab. [MOV96]

Das sieht auf den ersten Blick gut aus, genügt aber nicht, um Fälschern das Leben schwer zu machen. Zum einen ist die Abhängigkeit nur in einer Richtung wirksam. Eine Änderung am Geheimtextblock c_i beeinflußt die vorhergehenden Blöcke nicht; der Empfänger erhält beim Entschlüsseln $m_1 \ldots m_{i-1}$ unverändert. Aber auch in der anderen Richtung wirkt sich die Änderung von c_i nur beschränkt aus. Davon sind nämlich nur die Blöcke m_i und m_{i+1} betroffen. Für m_{i+1} kann der Angreifer sogar bestimmen, welche Bitpositionen sich (möglicherweise) ändern, indem er genau diese Stellen in c_i manipuliert, denn m_{i+1} ist ja nichts anderes als $\overline{D_K}(c_{i+1}) \oplus c_i$.

Wie sich Blockchiffren trotzdem zur Nachrichtenauthentifizierung einsetzen lassen, zeigt zum Beispiel Simmons in seinem Überblick [Sim92]. Die gesamte Nachricht wird im CFB-Modus verschlüsselt und letzte Block, der sowohl vom Schlüssel als auch von der gesamten Nachricht abhängt, wird zusammen mit dem *Klartext* übertragen. Der Empfänger wiederholt die Prozedur und vergleicht das Ergebnis mit dem Block, den er zusammen mit der Nachricht erhalten hat. Stimmen beide überein, stammt die Nachricht tatsächlich vom vermuteten Absender und ist unverfälscht.

Dem Fälscher fehlt hier der Geheimtext. Ihn könnte er verändern und so das Ergebnis des Entschlüsselns nicht nach Belieben, aber doch vielleicht gezielt genug für seine Zwecke beeinflussen. Nun berechnet der Empfänger nur noch den letzten Block dieses Geheimtextes als Funktion der empfangenen Nachricht und des geheimen Schlüssels und vergleicht den Funktionswert mit dem, den der Sender auf gleiche Art ermittelt hat. Das ist das Prinzip der Codes zur Nachrichtenauthentifikation, englisch Message Authentication Codes (MAC). Will der Fälscher fälschen, muß er zu seiner Nachricht einen passenden MAC konstruieren. Dazu brauchte er aber, so der Algorithmus sicher ist, den Schlüssel, der in die Berechnung eingeht.

Codes zur Nachrichtenauthentifizierung sind das Gegenstück zu den fehlererkennenden und -korrigierenden Codes der Kodierungstheorie [Sim92]. In beiden Fällen werden die Nachrichten vor der Übermittlung mit zusätzlicher Redundanz versehen. Codes zur Fehlerkorrektur sind so gestaltet, daß die wahrscheinlichsten Übertragungsfehler zu einer Nachricht führen, die im Sinne irgendeines Abstandsmaßes nahe bei der gesendeten Nachricht liegen und als ungültig erkennbar sind. Der Empfänger kann zu jeder empfangenen Nachricht die wahrscheinlich gesendete ermitteln. Zur Authentifizierung hingegen sollen Verfälschungen, die ein Angreifer bewußt vornimmt, eine ungültige Nachricht erzeugen, so sehr er sich auch anstrengt.

Blockchiffren sind nicht die einzige Möglichkeit, Authentifizierungscodes zu erzeugen. Schneier erwähnt eine Reihe von Algorithmen [Schn96], hat aber an fast allen etwas auszusetzen. Hier soll nur ein weiteres Verfahren besprochen werden.

Einen Code zur Nachrichtenauthentifizierung kann man aus jeder kryptographischen Hashfunktion gewinnen. Eine Hashfunktion bildet beliebig lange Nachrichten auf einen Hashwert konstanter Länge ab. Kryptographische Hashfunktionen sollen schwer zu invertieren und kollisionsfrei sein, das heißt es soll praktisch unmöglich sein, zu einem gegebenen Hashwert eine passende Nachricht oder zu einer gegebenen Nachricht eine zweite mit gleichem Hashwert zu finden. Außerdem soll sie für eine gegebene Nachricht leich zu berechnen sein [RLab98]. Eine kryptographische Hashfunktion erzeugt also eine

Art Fingerabdruck, ein Message Digest, das für die Nachricht charakteristisch ist und sich mit der Nachricht verändert; eine zweite Nachricht mit gleichem Fingerabdruck zu erzeugen gelingt praktisch nicht. Um daraus einen Authentifizierungscode zu erhalten, muß man Außenstehende nur noch daran hindern, zu einer veränderten Nachricht selbst einen neuen Hashwert zu bestimmen.

Algorithmus	Hashlänge	Bemerkungen
MD4	16 Bytes	RFC 1320; Sicherheitsprobleme
MD5	16 Bytes	RFC 1321; sicherer als MD4
SHA/SHA-1	20 Bytes	Bestandteil des Digital Signature Standard
RIPEMD-160	20 Bytes	[BDP97]

Tab. 3.2: *Einige Hashfunktionen*

Man könnte den Fingerabdruck einfach mit einem der bereits erwähnten symmetrischen Algorithmen verschlüsseln. Damit handelte man sich jedoch wieder die gleichen Probleme ein, derentwegen man einen MAC überhaupt benutzen möchte. Der bekannte Klartext – ein Angreifer kann den Fingerabdruck der Originalnachricht leicht berechnen – macht Stromchiffren wie RC4 unbenutzbar. In einen einzelnen Block paßt er bei gängigen Blockalgorithmen, zum Beispiel DES oder IDEA, nicht; zudem besitzt er, anders als unsere Tickets, keine Redundanz, die der Empfänger nutzen könnte, um Manipulationen am MAC zu erkennen.

Ein geeigneters Verfahren, einen geheimen Schlüssel in die Berechnung einfließen zu lassen, ist in RFC 2104 beschrieben. Es erfordert neben einer Hashfunktion $H(x)$ und dem Schlüssel K zwei Bytefolgen p_i und p_o, die vorgegeben sind und in der Spezifikation `ipad` bzw. `opad` genannt werden. Bezeichne $\oplus$ wieder die bitweise Addition modulo 2 und $\odot$ die Verkettung zweier Bytefolgen. Den MAC, der mit der Nachricht M versandt wird, erhält man durch $HMAC(M) = H((K \oplus p_o) \odot H((K \oplus p_i) \odot M))$. Die Länge des resultierenden Codes hängt von der Hashfunktion H ab. Üblich sind 16 (MD5) oder 20 (SHA, RIPEMD-160) Bytes. Das HMAC-Verfahren ist dank seiner Veröffentlichung als RFC – darin ist eine Referenzimplementierung in C enthalten – recht weit verbreitet. Andererseits ist es noch jung und deshalb wenig untersucht. Betrachtungen zur Sicherheit der HMAC-Konstruktion finden sich in [KaRo95] und [BCK96].

3.1.4 Digitale Signaturen

Verschlüsselung schützt kurze, ein Authentifizierungscode auch lange Nachrichten vor unbemerkter Manipulation. Das ist schön, macht uns aber noch nicht glücklich. Beide Verfahren arbeiten symmetrisch. Sender und Empfänger brauchen einen Schlüssel, der jedem der beiden und niemandem sonst bekannt ist. Sie können ihn konspirativ austauschen oder mit einem geeigneten Protokoll vor aller Augen vereinbaren, ohne daß ihn hinterher ein anderer kennt.

Ist das nicht genug? Ist es nicht. Kopien des Schlüssels werden an zwei Orten aufbewahrt. Wer ihn stehlen möchte, kann sich das schlechter bewachte Ziel aussuchen. Für die digtale Eintrittskarte bedeutet das, daß die Kontrollstation genauso gut unbefugtem Zugriff geschützt werden muß wie das Verkaufssystem. Das ist aufwendig, denn während das Verkaufssystem für Außenstehende nur über eine Netzverbindung erreichbar ist, laufen Tag für Tag Tausende von Besuchern an den Kontrollstellen vorbei. Jedes Sender-Empfänger-Paar braucht einen eigenen Schlüssel. Der Sender kann nur dem Empfänger die Echtheit einer Nachricht nachweisen, aber keinem Dritten.

Digitale Signaturen lösen all diese Probleme auf – aus heutiger Sicht – naheliegende Weise. Sender und Empfänger teilen sich nicht mehr ein gemeinsames Geheimnis, sondern nur der Sender besitzt einen geheimen Schlüssel K_s (s wie secret). Dessen öffentliches Gegenstück K_p (p für public) darf hingegen nicht nur der Empfänger kennen, sondern auch sonst jeder. Der Sender kann mit seinem geheimen Schlüssel eine Signierfunktion S_{K_s} ausführen und jeder, der den öffentlichen Schlüssel besitzt, die Verifikationsfunktion $V_{K_p} = S_{K_s}^{-1}$. Die Funktionen müssen so beschaffen sein, daß bei Kenntnis des öffentlichen Schlüssels K_p der geheime K_s des Senders praktisch nicht berechnet werden kann. Geeignete Funktionen können zum Beispiel aus der Faktorisierung großer Zahlen oder dem diskreten Logarithmus gewonnen werden [MPW92].

Wie schon bei den MACs wird die Nachricht im Klartext übertragen und um Redundanz erweitert, die dem Empfänger die Echtheitsprüfung gestattet. Dazu sind Hashfunktionen gut geeignet, sofern man nur Angreifer daran hindern kann, den Hashwert durch einen selbst berechneten zu ersetzen [MPW92]. Authentifizierungscodes, die aus einer Hashfunktion erzeugt werden, nehmen dafür den geheimen Schlüssel zur Eingabe hinzu.

Für eine digitale Signatur wird statt dessen die Signierfunktion auf einen Fingerabdruck der Nachricht angewandt. Ist H eine kryptographische Hashfunktion und M die zu übertragende Nachricht, so ermittelt der Sender zunächst den Fingerabdruck $F = H(M)$. Dann berechnet er für F den Wert seiner Signierfunktion und erhält $A = S_{K_s}(F) = S_{K_s}(H(M))$. Zusammen mit der Nachricht M schickt er A an den Empfänger. Jener wendet die Verifikationsfunktion auf A an und erhält $V_{K_p}(A) = S_{K_s}^{-1}(A) = F$. Weiter berechnet der Empfänger selbst einen Fingerabdruck $F' = H(M)$ der erhaltenen Nachricht. Stimmen F und F' überein, hat er die Nachricht unverfälscht empfangen und kann sicher sein, daß sie vom vermuteten Sender stammt.

Da der Empfänger kein Geheimnis mehr braucht, um die Echtheit einer Nachricht zu prüfen, kann jeder in seine Rolle schlüpfen. Dadurch ist es möglich, Dritten die Herkunft einer Nachricht zu beweisen. Der geheime Schlüssel muß nur noch an einem Ort vor Neugierigen geschützt werden. Es ist nur noch ein Schlüssel pro Sender nötig und nicht einer für jede Kommunikationsbeziehung.

Digitale Signaturen werden derzeit am häufigsten mittels des RSA-Kryptosystems erzeugt [RLab98]. RSA ermöglicht nicht nur die Herstellung von Signaturen, sondern auch asymmetrisches Verschlüsseln. Sender und Empfänger tauschen dabei ihre Rollen, was den Besitz von Geheimnissen anbelangt. RSA ist nicht nur weit verbreitet, sondern auch fast so alt wie die Idee, daß es überhaupt Kryptographie mit öffentlichem Schlüssel geben könnte. Der RSA-Algorithmus gehört deshalb zu den am besten untersuchten

Algorithmus	Signaturlänge	Schlüssellänge (Modulus)	Grundlage	Bemerkungen
RSA	gleich der Schlüssellänge	ab 512 Bit	Faktorisierung großer Zahlen	kürzere Schlüssel sind möglich, aber unsicher
DSA	320 Bit	bis 1024 Bit	diskreter Logarithmus	längere Schlüssel möglich, aber in DSS nicht vorgesehen

Tab. 3.3: *Verbreitete Signaturverfahren*

asymmetrischen Verfahren.

Neben asymmetrischen Kryptosystemen, die wie RSA sowohl das Verschlüsseln als auch das Signieren erlauben, gibt es auch solche, die nur eins von beidem ermöglichen. Als Vertreter dieser Gruppe hat in jüngerer Zeit vor allem der Digital Signature Algorithm (DSA) Verbreitung gefunden, der Bestandteil des amerikanischen Digital Signature Standard (DSS) ist [RLab98].

Asymmetrische Kryptographie macht den Schlüsselaustausch einfacher, wirft aber gerade dadurch ein neues Problem auf. Die öffentlichen Schlüssel darf zwar jeder kennen, aber wenn sie völlig ungesichert übermittelt werden, ist ein Man-in-the-middle-Angriff möglich [Schn96]. Der Angreifer fängt dabei den öffentlichen Schlüssel während der Übertragung ab und schickt statt seiner einen anderen weiter, zu dem er den zugehörigen geheimen Schlüssel besitzt. Der Empfänger des falschen Schlüssels wird nun die Signaturen des Angreifers für echt halten oder Nachrichten mit dessen öffentlichem Schlüssel verschlüsseln. Solange der Angreifer die gesamte Kommunikation zwischen den beiden Beteiligten abfangen kann, ist er so in der Lage, nach Belieben mitzulesen und Signaturen zu fälschen.

Öffentliche Schlüssel dürfen also zwar von jedem gelesen werden, brauchen aber selbst einen Echtheitsnachweis, wenn sie über unsichere Kanäle verbreitet werden. Das Echtheitszertifikat kann mittels einer digitalen Signatur an den Schlüssel gebunden werden, aber um sie zu prüfen, braucht man wieder einen sicher echten öffentlichen Schlüssel der Einrichtung, die das Zertifikat ausgestellt hat. Wie man es auch dreht und wendet, wenigstens einmal ist Kommunikation auf einem manipulationssicheren Kanal nötig [MOV96].

Ein zweites Problem schränkt den Einsatz asymmetrischer Kryptographie stärker ein. Sie erfordert weit aufwendigere Rechenoperationen als symmetrische Verfahren. Asymmetrische Verfahren sind deshalb schwieriger in Hardware zu implementieren und arbeiten langsamer. Die Langsamkeit spielt eher beim Einsatz zur Verschlüsselung eine Rolle; hier nutzt man das asymmetrische Verfahren in der Regel nur, um einen Sitzungsschlüssel für einen symmetrischen Algorithmus zu vereinbaren. Wegen des Aufwandes für Hardwareimplementierungen sind Chipkarten für asymmetrische Kryptographie teuer. DSA kann jedoch auch auf elliptischen Kurven implementiert werden, was unter anderem Hardware-

Implementierungen vereinfacht [JuMe97].

Sollen nur sehr kleine Nachrichten authentifiziert werden, macht sich ein weiterer Nachteil bemerkbar. Signaturen sind, insbesondere wenn der RSA-Algorithmus verwendet wird, wesentlich länger als MACs [Schn96].

3.1.5 Schlüsselverwaltung

> »Seattle (dpa) - Die 520 000 Bürger von Seattle im US- Bundesstaat Washington sollten tunlichst nicht mehr die öffentlichen Briefkästen benutzen. Das rät die Polizei, nachdem Diebe offenbar den Zentralschlüssel für alle 1 600 blauen Postbehälter gestohlen und kopiert haben. Ein Postsprecher gab bekannt, daß Briefe bis weit ins nächste Jahr hinein möglichst in den Postämtern aufgegeben werden sollen. Das Ersetzen der derzeitigen Schlösser an allen Briefkästen in Seattle könnte bis Ende nächsten Jahres dauern.« [DPA98a]

Kryptographie dient dazu, komplexe Probleme auf den richtigen Umgang mit einer kleinen Anzahl von Schlüsseln zu reduzieren. [MOV96] Das bedeutet insbesondere sichere Geheimhaltung und Schutz vor mißbräuchlicher Verwendung geheimer Schlüssel sowie die Authentizitätssicherung öffentlicher Schlüssel. Die Benutzer müssen dann nicht mehr dem Gesamtsystem vertrauen, sondern nur noch den wenigen, leicht zu überwachenden Komponenten, die den Umgang mit Schlüsseln betreffen. Ein ansonsten sicheres Design vorausgesetzt, steht und fällt die Sicherheit mit dem Schlüsselmanagement.

Die Schlüsselverwaltung erfolgt auf der Grundlage eines Regelwerks,[14] das die Bedrohungen beschreibt, vor denen das System schützen oder geschützt werden soll. Darüber hinaus legt es Praktiken und Verfahren für das Schlüsselmanagement, die Verantwortungsbereiche der Beteiligten sowie Art und Umfang zu führender Aufzeichnungen fest.

Das Schlüsselmanagement umfaßt nach [MOV96] Techniken und Verfahren zur Initialisierung des Systems, zur Erzeugung und Verbreitung, zur Kontrolle der Benutzung, zum Aktualisieren, und Vernichten sowie zum Speichern und Archivieren von Schlüsseln. Dabei dürfen sie insbesondere nicht verlorengehen und keinem Unbefugten zur Kenntnis gelangen.[15] Die Lebensdauer der Schlüssel muß festgelegt werden und Prozeduren für die Ersetzung einmal im Normalfall und zum anderen für den Fall, daß ein Schlüssel kompromittiert wird.

Für den Ticketverkauf via Internet genügen einfache Verfahren, denn die Situation ist überschaubar. Es gibt nur zwei Parteien, den Veranstalter und die Kunden. Die gesamte Kryptographie findet auf Veranstalterseite statt. Der Kunde ist nichts weiter als ein unsicherer Nachrichtenkanal.[16] Es gibt also insbesondere keine verschiedenen Parteien, die trotz gegenseitigen Mißtrauens Schlüssel austauschen müßten, und ein sicherer Kanal zur Schlüsselübertragung ist leicht herzustellen.

[14] Im Englischen hat sich die Bezeichnung »security policy« eingebürgert. Die wörtliche Übersetzung Sicherheitspolitik wirkt ein wenig holperig.

[15] Wegen der großen Gefahr eines Angriffs von innen ist eigentlich jeder unbefugt.

[16] So der Kunde das als diskriminierend empfindet, lese man ihm laut vor: »There is a Bank B, a collection of vendors $\{V_i\}$, and a collection of customers $\{C_i\}$, all of whom are assumed to be communicating Turing Machines.« [FrYu93]

Das größtmögliche Unglück ist die Kompromittierung eines geheimen Schlüssels. Wer einen solchen ausspäht, kann damit nicht nur selbst Tickets herstellen, sondern stellt den Veranstalter noch vor ein zweites Problem. Wurden bereits Tickets verkauft, die mit diesem Schlüssel gesichert sind, müssen sie benutzbar bleiben. Verlieren ehrliche Kunden ohne eigene Schuld ihr Eintrittsgeld, werden sie zu ehemaligen Kunden.

Der Veranstalter betreibt ein Verkaufssystem und Kontrollstationen. An drei Punkten sind Angriffe auf Schlüssel möglich.

- *Angriffe auf das Verkaufssystem* kompromittieren alle geheimen Schlüssel. Selbst wenn man den Angreifer daran hindern kann, Schlüssel zu kopieren, ist er doch in der Lage, sie bis zur Entdeckung zu benutzen, um scheinbar authentische Nachrichten auf Vorrat zu erzeugen. Da er außerdem die Software manipulieren kann, ist nach einem solchen Angriff praktisch eine Neuinstallation notwendig. Das Verkaufssystem ist möglicherweise bei einem Internet-Provider untergebracht. Angriffe sind ohne physischen Zugang möglich.

- Ein *Angriff auf eine Kontrollstation* kompromittiert bei Einsatz symmetrischer Verfahren die dort benutzten Schlüssel. Der Angreifer kann außerdem die Software manipulieren und eigene Schlüssel installieren. Für einen Angriff auf eine Kontrollstation ist physischer Zugang notwendig.

- Der *Schlüsseltransport* ist nur dann problematisch, wenn Schlüssel regelmäßig gewechselt werden. Schlüssel für symmetrische Verfahren können hierbei ausgespäht werden. Auch in diesem Fall hat der Angreifer die Möglichkeit, eigene Schlüssel in einer Kontrollstation zu installieren. Manipulationen an der Software sind dagegen ausgeschlossen.

Wenigstens für das Verkaufssystem muß vorausgesetzt werden, daß geheime Schlüssel sicher vor unbefugtem Zugriff gespeichert sind. Wer an dieser Stelle erfolgreich einen Angriff vorträgt, hat gewonnen und wird mit einer kostenlosen Dauerkarte belohnt.[17] Jede andere Annahme macht das Schlüsselmanagement lediglich komplizierter, aber nicht sicherer. Vertrauenswürdige Komponenten sind unverzichtbar, und sei es nur zur Speicherung von Hauptschlüsseln, mit denen die übrige Schlüsselverwaltung gesichert wird. Das Verkaufssystem ist in erster Linie durch seine Netzanbindung bedroht.

Die sicherheitsrelevanten Komponenten der Kontrollstation sind vor allem vor physischem Zugriff durch Unbefugte zu schützen. Schlüssel für symmetrische Verfahren können darüber hinaus in besonderen Sicherheitsgeräten (das könnten etwa Chipkarten sein) gespeichert werden. Sie müßten einen Schlüssel speichern und damit MACs prüfen oder Nachrichten entschlüsseln können, dürften den Schlüssel selbst aber nicht herausgeben. Sogar nach einem Diebstahl der gesamten Station könnten die alten Schlüssel weiter verwendet werden, um die bereits verkauften Tickets zu prüfen – wenn nicht gerade das Gerät gestohlen wird, das den Schlüssel enthält.

[17] Das ist kein Scherz. Wer ein ernstes Sicherheitsproblem findet, sollte dafür belohnt werden, indem man ihn die Frchte seiner Entdeckung legal nutzen läßt. Dann hat er nämlich einen guten Grund, seine Entdeckung mitzuteilen.

Daß bei symmetrischen Verfahren jede Kontrollstation einen eigenen Schlüssel bekommt, versteht sich von selbst.

Der letzte Angriffspunkt ist der Schlüsseltransport. Er läßt sich entschärfen, indem man ihn vermeidet. Es gibt bei der hier diskutierten Anwendung keinen Anlaß, regelmäßig die Schlüssel zu wechseln. Die Sicherheit des Kryptosystems leidet nicht wirklich unter den tausend Eintrittskarten, die ein fleißiger Betrüger vielleicht sammeln kann. Einzig bei der Einrichtung oder Erweiterung des Systems sowie nach Kompromittierung sind neue Schlüssel zu installieren.

Die Schlüsselverwaltung ist deshalb einfach: Man kann sie von Hand erledigen. Das Verkaufssystem erzeugt einen Schlüssel, den ein Mensch auf einer Diskette oder einem anderen Datenträger zur Kontrollstation trägt und ihn dort installiert. Seine Lebensdauer endet in der Regel mit der des Systems oder im Falle der Kompromittierung. Fast alle Fragen, um die es beim Schlüsselmanagement sonst noch geht, erledigen sich damit von selbst. Alternativ könnte der Schlüssel, wenn symmetrische Verfahren benutzt werden, gleich bei der Erzeugung auf einer Chipkarte gespeichert werden, die ihn nicht mehr herausgibt, sondern damit selbsttätig MACs prüft oder Nachrichten entschlüsselt.

Ist das nicht zu einfach? Nein, denn jedes Schlüsselmanagement erfordert als ersten Schritt Handarbeit; wenigstens (und im Idealfall: höchstens) einmal muß ein sicherer Kanal ohne kryptographische Hilfe aufgebaut werden [MOV96]. Wo geheime Schlüssel gespeichert werden, sind in jedem Fall Sicherheitsmaßnahmen nötig, um sie vor unbefugtem Zugriff zu schützen. Es brächte also keinerlei Erleichterung, tauschte man in komplizierten Prozeduren regelmäßig Schlüssel aus. Der anfängliche sichere Kanal genügt, um die Arbeitsschlüssel zu übermitteln, und der ohnehin notwendige Zugriffsschutz macht Kompromittierungen unwahrscheinlich, sonst wäre er kaputt.

Ein weiteres Argument spricht für die simpelste Lösung. Die Benutzer werden aller Wahrscheinlichkeit nach nichts von Kryptologie verstehen. Sie besitzen nicht das Wissen, Zertifikate zu prüfen, und auch kein Verständnis für Sicherheit. Sie werden insbesondere nicht begreifen, daß und warum sie Schlüssel überhaupt wechseln sollten. Nicht weil sie dumm wären, sondern weil Kryptologie in ihrem Weltbild keine allzu große Rolle spielt.

Das Schlüsselmanagement muß deshalb entweder automatisch erfolgen, so daß die Benutzer keine schwerwiegenden Fehler machen können, oder sich an dem orientieren, was die Benutzer bereits kennen. Hardware-Schlösser und -schlüssel kennen sie und vermutlich werden sie kryptographische Schlüssel damit assoziieren und sie genauso benutzen [Nor90]. Man kann den Benutzern leicht vermitteln, daß sie den Rechner ihrer Kontrollstation wegschließen oder auf ihre Chipkarten gut aufpassen müssen, denn das ist ein Konzept, das sie längst kennen. Ihre Kartenrollen schließen sie ja jetzt auch gut weg und daß man den Zündschlüssel abzieht, wenn man sein Auto verläßt, ist für die meisten ebenfalls selbstverständlich. Das KISS-Prinzip[18] legt unter diesen Umständen die einfachere Lösung nahe, weshalb auf den automatisierten Wechsel von Schlüsseln verzichtet werden soll.

Was tun, wenn *es* doch passiert, wenn beispielsweise ein Cracker[19] in das Verkaufssy-

[18] »Keep It Simple, Stupid« [Ray96, Ray98]

[19] Der häufig verwendete Begriff *Hacker* bezeichnet eine andere, harmlose Spezies [Ray96, Ray98].

stem eindringt? Dann sollte es einen Weg geben, bereits verkaufte Tickets mit den alten Schlüsseln zu kontrollieren, ohne gleich jedem die Tür aufzuhalten. Dazu müsste man tatsächlich verkaufte Tickets erkennen können, obwohl die kryptographische Sicherung dabei nicht mehr hilft.

Eine allgemeine Lösung für dieses Problem gibt es nicht. Man kann geheime Schlüssel auch auf dem Verkaufssystem so speichern, daß sie nicht kopierbar sind, sondern lediglich benutzt werden können. Der Angreifer kann dann nicht einfach mit ihnen verschwinden, sondern muß immer wieder zum Tatort zurückkehren, wenn er ein neues Billett erzeugen möchte, was sein Entdeckungsrisiko erhöht. Jede Benutzung eines geheimen Schlüssels sollte in einer Protokolldatei vermerkt werden. Die allerdings muß wiederum vor Manipulation geschützt werden und das geht nur für Einträge, die vor einem Einbruch erzeugt wurden [ScKe98].

3.2 Reality Check II

Nach dem kleinen Ausflug in die Kryptologie zurück ins richtige Leben. Unser böser Betrüger hat jetzt ein echtes Problem: Seine Fälschungen werden sicher erkannt. Kopieren fällt damit auch flach. Er könnte sich hinsetzen und so lange über Kryptologie meditieren, bis er einen brauchbaren Angriff auf das benutzte Verfahren gefunden hat. Das möchte er aber gar nicht, denn er ist kein Kryptoanalytiker, sondern ein Betrüger, der für umsonst ins Kino will. Um ihn von Fälschungsversuchen abzuhalten genügt es schon, wenn er nur *glaubt*, das sei kompliziert. (Was uns jedoch nicht zur Schlamperei verführen sollte!)

Der Einsatz kryptographischer Verfahren allein ist daher kein Garant für Sicherheit. Im Gegenteil, er kann – »Mein Schlüssel ist länger als deiner!« – Sicherheit vorgaukeln, wo in Wirklichkeit Scheunentore offenstehen. In einer Fußnote fanden bereits die EC-Karten Erwähnung. Sie sind ein Lehrstück dafür, wie man an sich recht brauchbare Kryptosysteme einsetzen muß, wenn das Gesamtsystem trotzdem unsicher werden soll. Das allein wäre vielleicht noch erträglich, doch wenn auf diese Weise Schäden entstehen, behaupten die Verantwortlichen gern, es sei alles sicher, denn man habe ja Kryptographie.

3.2.1 Unsichere EC-Karten

Am 1. September 1998 verkündete das Amtsgericht Frankfurt am Main ein Urteil [AGF98], das einiges Aufsehen erregte. Es verurteilte eine Bank, der Klägerin den Schaden von mehr als 4.500,- DM zu ersetzen, der bei Abhebungen mit ihrer gestohlenen EC-Karte entstanden war. In der Urteilsbegründung führt das Gericht unter anderem an, daß der Dieb die Geheimnummer (PIN) mit einer Wahrscheinlichkeit von 1:150 habe ermitteln können.

Wie das möglich ist, erläutert [Kuhn97]. Das Problem liegt nicht im DES-Algorithmus, der bei Erzeugung und Überprüfung der PIN eingesetzt wird. Er hat unbestreitbar Schwächen, deren größte die inzwischen zu kleine Schlüssellänge [EFF98a, EFF98b] ist, gilt aber immer noch als gutes und sicheres Design. Der Fehler, der aus 1:3000 theoretischer Ratewahrscheinlichkeit 1:150 in der Praxis macht, wäre mit jedem anderen Verschlüsselungsalgorithmus genauso aufgetreten.

Bei der Erzeugung und Prüfung der Geheimnummer muß unter anderem eine vierstellige Hexadezimalzahl auf eine vierstellige Dezimalzahl abgebildet werden. Nichts leichter als das, dachten die Erfinder des Verfahrens – und entschieden sich für die Division jeder einzelnen Stelle modulo 10. Die Ziffern 0 bis 5 kommen deshalb häufiger in Geheimnummern vor als die übrigen. Darüber hinaus sollte die erste Stelle der PIN niemals den Wert null haben. Tritt die Null dort im Ergebnis auf, wird sie deshalb durch eine Eins ersetzt. Das ändert noch einmal die Wahrscheinlichkeitsverteilung für die Ziffern der ersten Stelle.

In seiner wahrscheinlichkeitstheoretischen Betrachtung weist Kuhn nach, daß die leichtfertig hingenommene Ungleichverteilung der Ziffern die Chance, in drei Versuchen die PIN einer zufällig gewählten EC-Karte zu raten, damit auf 1:150 steigt.

Das EC-Beispiel ist in zweierlei Hinsicht lehrreich. Zum einen zeigt es, daß Fehler im Design und der Implementierung von Sicherheitssystemen auch dann schwere Mängel stecken können, wenn die kryptographischen Verfahren selbst sicher und sauber implementiert sind.

Zum anderen verdeutlicht es die nichttechnischen Wirkungen, die der Einsatz von Kryptosystemen haben kann. Jahrelang haben die Banken die gesamte Verantwortung für Schäden durch Kartenmißbrauch den Kunden zugeschoben. Mit dem Hinweis auf DES und darauf, daß es »auch heute noch technisch ausgeschlossen« [AGF98] sei, die PIN aus der Karte zu errechnen, konnten die Banken bis vor kurzem jede Schadensersatzforderung abwehren.

3.2.2 Knapp daneben ist auch vorbei

Schneier [Schn97, Schn98], Anderson [And93] und Bezuidenhoudt [AnBe96] haben untersucht, wo und wie Sicherheitsprobleme beim Einsatz von Kryptographie tatsächlich auftreten.

Schneier zählt in [Schn98] auf, an welchen Stellen Angriffe auf ein System ansetzen können:

- Angriffe auf das *kryptographische Design* richten sich gegen die Art, wie kryptographische Algorithmen eingesetzt werden. Als häufig angreifbare Komponente nennt Schneier die Erzeugung von Zufallszahlen; werden aus schlechten Zufallszahlen Schlüssel erzeugt, bleibt das System auch mit den besten Verschlüsselungsverfahren unsicher. Anderson erwähnt Sicherheitsmodule im Bankbereich, die lediglich die Uhrzeit zur Schlüsselerzeugung verwenden [And93]. Eine Problembeschreibung und Empfehlungen zur Erzeugung von Zufallswerten sind in RFC 1750 [ECS94] zu finden.

- Angriffe auf die *Implementierung* werden durch Fehler im Programm oder im Programmdesign ermöglicht. Anderson liefert mehrere Beispiele dafür, unter anderem das eines Geldautomaten, der eine eingeführte Telefonkarte für die vorher benutzte Bankkarte hielt. Ein Ganove mußte sich nur mit einer Telefonkarte in die Schlange stellen und die Geheimnummer des Kunden vor ihm ausspähen. [And93]

- Angriffe auf *Passworte* nutzen die Tatsache aus, daß viele Benutzer lausige Passworte wählen. Das wird zum Beispiel von Programmen wie *Crack* ausgenutzt, um

mit einem einfachen Wörterbuchangriff Unix-Passworte herauszufinden, obgleich die Verschlüsselung der Passworte nachweislich sicher ist [Neu95]. Zwingt man die Nutzer, gute Passworte zu verwenden, schreiben sie sie auf.

- Angriffe auf die *Hardware* können auftreten, wenn eine Anwendung manipulationssichere Geräte verlangt. In den letzten Jahren wurden mehrere Methoden entdeckt, ohne Eingriff in das Gerät Informationen über die Vorgänge im Inneren zu erlangen.

- Angriffe auf das *Vertrauensmodell* sind möglich, wenn ein ein System von falschen Annahmen über Beteiligte ausgeht.

- Seine *Benutzer* können eine System versehentlich unsicher machen, indem sie es falsch nutzen. Anderson bringt auch dafür Beispiele, etwa die ungeschützte Aufbewahrung von Schlüsseln [And93].

- Bei der *Fehlerbehandlung* können unsichere Zustände eintreten. Schlimmer noch ist der Fall, daß es keine Möglichkeit gibt, Fehler zu beheben. Das EC-Kartensystem besitzt aus diesem Grund eine Möglichkeit, kompromittierte Schlüssel zu ersetzen, ohne daß neue Karten ausgegeben werden müssen. Leider trägt das Verfahren zur leichten Erratbarkeit der PINs bei [Kuhn97].

- Angriffe auf die *kryptographischen Algorithmen* selbst lassen sich leicht vermeiden, indem man anerkannt starke Verfahren einsetzt, ohne vermeintliche Verbesserungen einzubringen. Anderson schreibt über eine Bank, die jahrelang vor den Augen von Prüfern und Beratern ein offensichtlich untaugliches »Verschlüsselungs«verfahren einsetzte, ohne daß jemand dagegen protestiert hätte [And93].

Schneier zieht daraus zwei Schlüsse. Zum einen sollte sich ein System nicht allein darauf gründen, daß die Kryptographie alle Angriffe verhindert. Für den Fall des Versagens sollte es Methoden zur Betrugserkennung und zur Begrenzung des möglichen Schadens bereithalten. Zum anderen dürfen die Angreifer nicht unterschätzt werden. Sie wählen Wege, an die beim Entwurf niemand gedacht hat und ihnen genügt eine einzige Lücke zum Erfolg. Das System dagegen muß jedem möglichen Angriff standhalten.

3.2.3 Gelegenheit macht Diebe

Abschnitt 3.1.1 beschäftigt sich mit den Fähigkeiten, die ein Betrüger besitzen könnte, und alle Betrachtungen drehten sich bis jetzt um *mögliche* Angriffe. Interessanter ist die Frage, was er tatsächlich versuchen wird, welche Angriffe *wahrscheinlich* sind [And93], denn wir müssen uns vor echten Ganoven schützen und nicht vor theoretischen. Der Gegner ist nicht die CIA, sondern jemand, der sich kostenlos ins Kino, Theater oder Museum schleichen möchte. Es schadet nicht, die Fähigkeiten des Gegners vorsichtshalber zu überschätzen, doch das ist nur die halbe Wahrheit.

Betrüger sind Opportunisten [And93, And94, Schn97]. Sie möchten sich mit möglichst wenig Aufwand einen möglichst großen Vorteil verschaffen. Sorgfältig geplante High-Tech-Angriffe auf ein System kommen deshalb selten vor.

Statt dessen werden Gelegenheiten genutzt, die sich anbieten. Wie die Handtasche auf dem Beifahrersitz, die den Einbruch ins abgestellte Auto provoziert, dienen kleinste Nachlässigkeiten als Ansatzpunkt. Als Schutz davor genügt es oft schon, daß ein System sicherer ist oder auch nur scheint als das nebenan [Schn97]. Ein Fahrrad schützt man am besten vor Diebstahl, indem man es neben ein schöneres stellt, das schlechter gesichert ist.

Selbst gut versteckte Fehler im Entwurf oder der Implementierung werden früher oder später zufällig entdeckt und es wird immer jemanden geben der sie gnadenlos ausnutzt [And93, AnBe96]. Ein gutes Beispiel dafür ist das sogenannte Phone Phreaking [Neu95]. Vor nicht allzu langer Zeit boten Telefonnetze weltweit die Möglichkeit, kostenlos zu telefonieren. Man mußte dazu lediglich einige Töne in den richtigen Frequenzen durch die Leitung schicken. Da die Signalisierung, also die interne Kommunikation etwa zwischen Telefonzelle und Vermittlung oder zwischen Vermittlungsstellen, auf dem gleichen Kanal erfolgte wie die Übertragung der Nutzinformation, ließen sich so interne Ablüfe manipulieren.

Betrüger halten sich nicht an Regeln [Schn97]. Täten sie es, könnte man alle Verbrechen einfach verbieten. Sie werden das System auf Arten angreifen, an die beim Entwurf niemand gedacht hat. Noch auf eine andere Weise halten sie sich nicht an Regeln. Sicherheitssysteme werden meist zum Schutz vor Außenstehenden entworfen, aber viele Angriffe kommen von innen [Neu95]. Anderson gibt an, daß in den englischsprachigen Ländern Jahr für Jahr etwa einem Prozent des Bankpersonals aus disziplinarischen Gründen gekündigt wird [And93].

Angreifer können unterschiedliche Motive haben. Einige sind gewöhnliche Betrüger, vielleicht auch nur solche, die angesichts einer verlockenden Gelegenheit schwach geworden sind; andere suchen Aufmerksamkeit und wollen am liebsten ins Fernsehen. Wieder andere sind Vandalen, die nur Schaden anrichten, ohne selbst einen Nutzen zu haben [Schn97]. Das führt zu Denial-of-Service-Angriffen, wie sie zum Beispiel [Neu95] schildert.

Und schließlich ist es einem Betrüger egal, wer durch seine Tat welchen Schaden erleidet. Er sieht nur seinen Vorteil. Das führt zur nächsten Frage.

3.2.4 Wer hat den Schaden?

Bis hierher wurde stillschweigend unterstellt, daß alle betrachtenswerten Schäden durch Betrug des Kunden entstehen und der Anbieter der Geschädigte ist. Das ist sinnvoll, denn davor soll das zu entwickelnde System schützen. Aber auch das ist nur die halbe Wahrheit. Zudem bietet, wie die vorigen Abschnitte zeigen, auch das beste System (und unseres wird sich später als übel zusammengehackter Prototyp erweisen) keine hundertprozentige Sicherheit, sondern kann allenfalls dafür sorgen, daß die verbleibenden Risiken akzeptabel und im Vergleich zum Nutzen klein sind [Schn97].

Welche Verlustrisiken entstehen beim Einsatz und wie sind sie zu bewerten? In [Luk97] werden vier Kriterien genannt. Wir wenden sie sogleich an:

1. *Warum entstehen Schäden?* Sie können durch eigenes Fehlverhalten oder solches

der jeweils anderen Partei entstehen, außerdem durch Handlungen Dritter sowie durch Versagen des Systems.

2. *Wer hat den Schaden?* Schäden durch eigenes Fehlverhalten sollte jeder selbst tragen. Unser System sollte eigenes Fehlverhalten schwer machen und vor allem nicht provozieren. Dem Veranstalter ersparen wir deshalb die Schlüsselverwaltung und der Kunde wird im nächsten Kapitel einen einfach zu benutzenden Datenträger bekommen. Durch Fehlverhalten des jeweils anderen sollte im Idealfall niemand einen Schaden erleiden. Unterstellt man beliebig hohe kriminelle Energie, läßt sich das nicht umsetzen. Deshalb muß die etwas schwächere Forderung genügen, daß jeder sein Risiko einschätzen können soll. Dasselbe gilt für Handlungen Dritter. Schäden aufgrund von Systemversagen sollte der Veranstalter tragen. Es ist sein System. Der Veranstalter wird das nicht wollen. Er soll trotzdem.

3. *Wann entsteht der Schaden?* Diese Frage scheint hier nicht relevant.[20]

4. *Wie hoch ist der Schaden?* Der Kunde soll auch im schlimmsten Fall nicht mehr verlieren können als den Wert seines Tickets – der durchaus hoch sein kann – und überhaupt nichts, solange er ein wenig gesunden Menschenverstand einsetzt. Der einzige kritische Vorgang ist in dieser Hinsicht ist der Verkauf; die Existenz geeigneter Verfahren wird hier einfach vorausgesetzt. Der Veranstalter trägt als Unternehmer ohnehin ein hohes Risiko. Der Ticketverkauf im Internet sollte ihn aber wenigstens nicht in den Bankrott führen können. Dazu muß er zum einen hinreichend gut vor Betrug geschützt sein, zum anderen darf ein Ausfall des Systems sein Geschäft allenfalls kurzzeitig stören.

Die meisten Fragen sind schon geklärt. Der Veranstalter bekommt Kryptographie und hoffentlich ein sicheres Drumherum, und aus heutigem Blickwinkel ist zu vermuten, daß der Ticketverkauf im Internet für die nächsten Jahre Nebensache bleiben wird. Der Kunde braucht im wesentlichen noch einen brauchbaren Datenträger und Sicherheit beim Verkaufsvorgang.

Zu klären bleibt, wer das Risiko bei Systemversagen trägt. Als Versagen werden auch Entwurfs- und Spezifikationsfehler eingestuft. Perfekte Technik gibt es nicht, die Beispiele dafür aus der Computerwelt füllen ein ganzes Buch [Neu95].

Entsteht durch technische Fehler nur dem Veranstalter ein Schaden, ist das sein Problem. Was aber, wenn der Kunde dadurch geschädigt wird, obwohl er keinen Fehler gemacht hat? Dann wird der Kunde schimpfen und der Veranstalter jede Verantwortung von sich weisen. Schlimmstenfalls landet der Streit vor Gericht – manche Eintrittskarten kosten richtig Geld – und das muß entscheiden.

Erhält der Kunde erst gar kein Ticket, obwohl er bezahlt hat, kann ihn höchstens das Bezahlprotokoll noch retten, sofern es die Zahlung beweisbar macht. Hat er seine Eintrittskarte bekommen, kann sie aber nicht nutzen, so möchte er den Eintrittspreis erstattet haben. Doch da könnte ja jeder kommen, einen Fehler der Technik behaupten und Geld verlangen.

[20] Klartext: Keine Ahnung, sorry.

Die Lösung ist einfach, nichttechnisch und aus Amerika [And94]. Dem Veranstalter wird die Beweislast auferlegt. Der Kunde muß die Erstattung schriftlich fordern und möglicherweise klagen; das ist eine wirksame Hemmschwelle gegen Betrug an dieser Stelle. Kommt der Fall aber vor Gericht, muß der Veranstalter nachweisen, daß seine Systeme korrekt arbeiten und das Ticket des Kunden zu Recht als falsch erkannt haben. Daß der Veranstalter starke Kryptoalgorithmen einsetzt, soll angesichts der unzähligen Fehlermöglichkeiten nicht genügen, die Last des Beweises auf den Kunden abzuwälzen.

Das schreibt sich einfach und ist vermutlich kompliziert, doch über die Details mögen sich die Juristen Gedanken machen. Das Signaturgesetz wird ihnen dabei allerdings nicht helfen, denn es regelt lediglich, wie man zu einer Signatur im Sinne des Signaturgesetzes kommt, sonst nichts [Schu98]. Auch kann nur die Herkunft einer Bitfolge durch eine Signatur beweisbar gemacht werden, nicht aber ihre Semantik [Fox98].

3.3 Schlußfolgerungen

3.3.1 Kryptographie

Die Kryptographie bietet die beschriebenen drei Möglichkeiten, Integrität und Authentizität der Tickets zu sichern. Bleibt die Frage, welche davon eingesetzt und welche Kryptosysteme dabei benutzt werden sollen. Der Autor kann sich nicht entscheiden und implementiert deshalb alle drei Varianten. Mögen andere herausfinden, welche am besten funktioniert. Auf umfangreiche Praxistests kann ohnehin nicht verzichtet werden.

Die Auswahl geeigneter kryptographischer Algorithmen ist einfach. Man verhalte sich konservativ und wähle gut untersuchte Verfahren, sofern sie die Untersuchung überstanden haben, ohne grössere Schwächen zu zeigen. Zu solchen Verfahren sind auch am einfachsten brauchbare und ausgiebeig getestete Implementierungen zu bekommen. Die sollte man tunlichst benutzen, denn der beste Algorithmus nützt nichts, wenn er falsch umgesetzt wird.

An sich genügt symmetrische Verschlüsselung, jedenfalls im unteren Preisbereich. Allerdings bleibt dabei ein ungutes Gefühl, denn der Ansatz ist wackelig wie ein Kartenhaus.

Wie in 3.1.2 dargelegt, hängt die Sicherheit bei diesem einfachen Verfahren wesentlich vom Nachrichteninhalt ab. Das scheint unproblematisch, denn der ist bekannt. Doch bei modularer Programmierung werden Inhalt und Authentifizierung von weitgehend unabhängigen Teilen der Software bestimmt. Das kann leicht dazu führen, daß der Zusammenhang zwischen beiden Teilen bei späteren Erweiterungen und Modifikationen übersehen wird [Neu95].

Auf der anderen Seite steht der Vorteil, daß kurze Nachrichten kurz bleiben. Sie müssen allenfalls auf die Blocklänge aufgefüllt werden. Außerdem ist die Lösung unschlagbar einfach.

Für die Implementierung empfiehlt sich der Algorithmus IDEA. Er arbeitet mit 128-Bit-Schlüsseln und 64 Bit Blocklänge und gilt als sehr sicher [MOV96]. Implementierungen sind frei verfügbar; die Schweizer Firma Ascom[21] besitzt allerdings in einigen Ländern, darunter der Bundesrepublik Deutschland, ein Patent auf den Algorithmus

[21] http://www.ascom.ch/

und verlangt bei kommerzieller Nutzung Lizenzgebühren. Ob das juristisch wasserdicht ist – reine Algorithmen sind hierzulande nicht patentierbar – ist unklar. Ein Problem ist es allemal.

Gut untersuchte Algorithmen mit größerer Blocklänge und verfügbarer Implementierung gibt es kaum. RC5 arbeitet mit Blocklängen bis zu 128 Bit [RLab98] und ist vermutlich ausreichend sicher [BaRi96]. Einige Verfahren mit noch längeren Blöcken stellt Ritter [Rit98] vor; sie sind aber auch noch weniger analysiert als zum Beispiel RC5.

Sobald die Ticketnachrichten länger als 128 Bit sind, kommt man kaum umhin, MACs oder Signaturen zu benutzen. Sie sind zudem unabhängig vom Nachrichteninhalt. Auch hier bekommt die einfachere Lösung den Vorzug. Asymmetrische Signaturverfahren scheinen zwar das Nonplusultra zu sein, doch wenn man sich vor Augen führt, daß das Verkaufssystem mit den geheimen Schlüsseln eventuell bei einem Internet-Provider untergebracht ist und die Schlüsselverwaltung ohnehin wegdiskutiert wurde, schmilzt der Vorsprung auf ein bisschen Buzzword-Compliance [Schn98] für die Marketingabteilung zusammen.

Zur Erzeugung von MACs kann das bereits beschriebene HMAC-Verfahren [KBC97] mit verschiedenen Hash-Funktionen benutzt werden. Verbreitet und verfügbar sind SHA-1 nach dem amerikanischen Secure Hash Standard, RIPEMD-160 [BDP97] und MD5 [Riv92]. Die Nachrichten werden damit um 20 bzw. 16 Bytes länger.

Bei den Signaturen heißen die Kontrahenten RSA und DSA. RSA ist älter und besser untersucht; DSA liefert kürzere Signaturen (und scheint recht solide konstruiert; die zugrundeliegenden Verfahren sind nicht neu). Die Erzeugung von Signaturen geht bei DSA schneller als die Prüfung. Das könnte sich als problematisch erweisen, denn bei der Ticketkontrolle müssen viele Signaturen in kurzer Zeit geprüft werden und die Kontrollstationen sollten mit preiswerter Hardware auskommen. Also RSA. Eine knappe Zusammenfassung der Diskussion RSA vs. DSA mit einigen Literaturverweisen ist in [RLab98] zu finden.

3.3.2 Umgebung

So sehr sich der Entwickler beim Entwurf und bei der Implementierung auch anstrengt, er wird Fehler machen. Andere werden diese Fehler finden – und sie entweder mitteilen oder heimlich ausnutzen. Der Weg zu einem sicheren System ist lang und steinig. Er erfordert die Mitarbeit vieler Fachleute, denn $2n$ Augen sehen mehr als 2 für jedes $n \in \mathbb{N}, n > 1$ [Schn97, And93, AnBe96]. Von unschätzbarem Wert ist dabei die Offenlegung der Quelltexte unter einer geeigneten Lizenz [OS98]. Auf diese Weise wird ein offener Arbeitsstil geradezu erzwungen, der die Softwarequalität spürbar verbessert [Gra90]. Ein Vergleichstest von Miller et al. [Mil98] weist nach, daß frei verfügbare GNU/Linux-Komponenten von deutlich höherer Qualität sind als ihre kommerziellen, unter kooperationsfeindlichen Belohnungsmodellen entwickelten Gegenstücke.

Welche Angriffsmöglichkeiten ein System läßt, kann nur der Praxistest zeigen [And93, AnBe96]. Woher sonst sollten die Entwickler auch erfahren, an was sie alles *nicht* gedacht haben?

Ebenfalls von großem Wert sind Vielfalt und Konkurrenz [AnBe96]. Je mehr verschiedene, unabhängig voneinander entwickelte Systeme im Einsatz sind, desto geringer sind

die Folgen eines Fehlers in einem dieser Systeme. Ein Sicherheitsproblem in Windows macht mit einem Schlag die ganze Welt verwundbar, sobald es entdeckt wird ...

Ein sicheres System muß genau das tun, was man von ihm erwartet. Nicht mehr und nichts anderes. Anderson [And93] schildert den Fall eines Geldautomaten, der – im Handbuch dokumentiert – Testtransaktionen erlaubte, bei denen jeweils zehn Geldscheine ausgegeben wurden. Das ist strafbare Dummheit und sie wurde bestraft. Wer könnte solch einer verlockenden Gelegenheit auch dauerhaft widerstehen? Ein anderer, ebenfalls für die Sicherheit bedeutsamer Aspekt sind die Erwartungen der Benutzer. Sie haben aufgrund dessen, was sie vom System wahrnehmen, eine Vorstellung über seine Arbeitsweise. Diskrepanzen zwischen diesen Vorstellungen und der tatsächlichen Funktion führen unweigerlich zu Fehlbedienungen [Nor90], die weitreichende Folgen haben können.

Hundertprozentige Sicherheit gibt es nicht, deshalb ist Redundanz nötig. Mißbrauch und Betrug sollen verhindert werden, aber falls das aus irgendwelchen Gründen nicht wie beabsichtigt funktioniert, muß wenigstens erkennbar sein, daß es ein Problem gibt. Beim vorausbezahlten Verkauf von Elektrizität in Südafrika [AnBe96] werden dazu Verbrauch und Verkauf verglichen, für den elektronischen Tickethandel bietet sich ein ähnliches Verfahren an. Werden entsprechende Aufzeichnungen geführt, läßt sich ohne weiteres prüfen, ob alle benutzten Tickets auch tatsächlich verkauft wurden. Im Betrugsfall findet man damit zwar nicht den Täter, aber man merkt zumindest schnell, daß man betrogen wird.

Angriffe von innen dürfen nicht vernachlässigt werden. »Der Veranstalter«, von dem hier immer die Rede ist, ist keine einzelne Person, sondern eine Firma mit Mitarbeitern – auch unzufriedenen und ehemaligen, die ihre Kenntnis der Interna mißbrauchen können und werden. Die Lösung des speziellen Problems muß deshalb in ein allgemeines Sicherheitskonzept integriert werden, das insbesondere Zugriffsschutz und Protokollierung umfaßt.

3.4 Internet-Sicherheit

Das Verhältnis zwischen Veranstalter und Kunden ist soweit geklärt; betrügerische Kunden sind für den Veranstalter dasselbe. Doch es könnte auch jemand versuchen, den Kunden zu schädigen. Gegen herkömmliche Methoden wie einfachen Diebstahl kann und soll das Verfahren nicht schützen, aber es soll auch keine neuen Wege eröffnen, oder jedenfalls keine einfachen.

Hier ist vor allem der Weg vom Verkaufssystem zum Kunden bedeutsam. Der Verkaufsvorgang erfolgt über ein unsicheres Netz. Dennoch möchte der Kunde sicher sein, daß er sein Geld dem Veranstalter anvertraut und keinem Betrüger. Das ist schon bei herkömmlichen Eintrittskarten ein Problem [DPA98b] und wird im Netz noch gravierender. Hübsch anzusehende Web-Sites kann jeder gestalten. Ob die Bitfolge, die er gekauft hat, ein gültiges Ticket ist, kann der Kunde aber erst bei der Kontrolle sicher feststellen. Er muß sich deshalb der Identität des digitalen Verkäufers sicher sein können.

Die Netzverbindung zwischen dem Rechner des Kunden und dem Verkaufssystem ist abhörbar. Wer hier lauscht, kann Tickets stehlen, indem er sie kopiert. Der Kunde bemerkt auch das erst bei der Kontrolle, wenn er abgewiesen wird, weil der Datendieb

schneller war. Kann ein Ganove beim Abhören feststellen, *wer* eine bestimmte Eintrittskarte gekauft hat, so wird er vielleicht statt ins Theater lieber zur Wohnung des Käufers gehen, um sie während dessen Abwesenheit in aller Ruhe nach Wertvollem zu durchsuchen.

Beide Probleme löst das Secure-Sockets-Layer-Protokoll (SSL), das von der Firma Netscape entwickelt und öffentlich dokumentiert wurde [Net98]. Es erweitert die Transportschicht zwischen Anwendungs- und Internet-Protokoll um Dienste zur

- Teilnehmerauthentisierung
- Nachrichtenauthentifikation sowie
- Verschlüsselung.

Die gängigen Web-Browser unterstützen SSL, geeignete Serversoftware steht ebenfalls zur Verfügung. Erwähnenswert ist die freie SSL-Implementierung SSLeay [HuYo98], die auch eine universell einsetzbare Kryptographiebibliothek enthält. Eine Arbeitsgruppe der IETF arbeitet an der Standardisierung eines SSL-Nachfolgers unter dem Namen TLS (Transport Layer Security).

Zur Nutzung von SSL genügt auf der Kundenseite der Besitz eines Browsers mit SSL-Unterstützung. Auf der Serverseite ist neben der korrekten Installation geeigneter Software ein Schlüselzertifikat einer – möglichst bekannten – Zertifizierungsstelle erforderlich. Das ist langweilig und wird daher nicht vertieft.

Für das Verkaufssystem ist der Netzanschluß die wesentliche Schnittstellle zur Außenwelt. Angriffe auf dieses System werden in erster Linie aus dem Netz kommen. Nach Möglichkeit sollte deshalb der Teil des Systems, der kryptographische Operationen ausführt, auf einem anderen Rechner laufen als der Webserver, der die Kunden unmittelbar bedient. Zusammen mit einem Firewall zwischen beiden Computern, der alles bis auf die beim Verkauf notwendige Kommunikation blockiert, erreicht man damit sehr hohe Sicherheit. Wer in den Rechner eindringt, auf dem der Webserver läuft, kann von dort aus zwar selbst Tickets erzeugen, aber dazu muß er immer wieder an den Tatort zurückkehren. Um die kryptographischen Schlüssel nicht nur benutzen, sondern auch kopieren zu können, muß er erst noch den Firewall überlisten *und* dann den eigentlichen Verkaufsrechner.

3.5 Plattform

Ein Sicherheitsfaktor ist das Betriebssystem. Hier ist es vor allem für den Zugriffsschutz und die Protokollierung von Ereignissen bedeutsam. Für die Entwicklung des Prototypen standen nur zwei Systeme zur Wahl, so daß die Frage lautet: Unix oder NT?[22]

Wie zufällig gemachte Erfahrungen zeigen, ist das Betriebssystem Microsoft Windows NT 4.0 nicht einmal in der Lage, Benutzerkonten sauber zu trennen. Wenn Bedienhandlungen eines Nutzers dazu führen können, daß sich das Aussehen der grafischen Oberfläche für andere Nutzer verändert, braucht man über den Einsatz des Systems für andere Zwecke als die Büroarbeit einer einzelnen Person gar nicht weiter nachzudenken. Es ist

[22] Diese Diskussion kann in der Newsgroup de.comp.advocacy beliebig vertieft werden.

weder für den Mehrbenutzerbetrieb noch gar für Anwendung mit Sicherheitsanforderungen zu gebrauchen.

Als Alternative bleibt nur Unix. Es ist in vielerlei Geschmacksrichtungen von kommerziellen Anbietern wie auch als freie Software erhältlich. Im Bereich der Internetserver ist es nicht ohne Grund weit verbreitet. Es funktioniert einfach. Die freien Implementierungen (Linux, verschiedene BSD-Varianten) erlauben darüber hinaus Einblick in jede einzelne Zeile ihrer Quelltexte, und erfüllen damit eine Forderung, die an alle sicherheitsrelevanten Komponenten eines Systems zu stellen ist.

4 Träger

Kommunikationsskulptur Heißt es ab sofort, wenn Musiker am Telephon sitzen und ihren Kollegen über Draht eines vordudeln. Außerdem ist es eine »experimentale Performance« und ein ausgesuchter Schwachsinn, den die Stadt Frankfurt auch noch bezahlt.
(Eckhard Henscheid: Dummdeutsch)

Sehet, das Himmelreich ist nahe! Das Billett ist gekauft, signiert und sicher zum Kunden übertragen. Auch wie es kontrolliert wird, ist im wesentlichen klar. Immer noch ist das Ticket aber nichts als eine Bitfolge, die der Kunde zum Veranstaltungsbesuch mit sich führen muß. Ein geeigneter Datenträger muß her. Er soll in die Jackentasche passen und weder beim Kauf noch bei der Kontrolle besondere Umstände breiten, außerdem haltbar und umempfindlich sein und die Umwelt nicht belasten.

4.1 Die Kandidaten

4.1.1 Disketten & Co.

Nahezu jeder Computer verfügt über ein Diskettenlaufwerk, das 3 1/2"-HD-Disketten mit einer Kapazität von 1,44 MB lesen und schreiben kann. Allerdings sind die Datenstrukturen, die auf eine Diskette geschrieben werden, nicht standardisiert. Es gibt auch keine plattformunabhängige Möglichkeit, Daten, die über einen Web-Browser abgerufen wurden, in einem definierten Format auf einer Diskette abzulegen.

Weit verbreitet ist das FAT-Dateisystem der DOS/Windows-Welt; es kann auch von vielen anderen Systemen gelesen und geschrieben werden. Im Prinzip jedenfalls, aber nicht in jedem Fall einfach mit einem Mausklick.

Diskettenlaufwerke sind langsam und wegen des hohen Mechanikanteils im Dauerbetrieb störanfällig. Beim Diskettenaustausch zwischen verschiedenen Rechnern können Probleme auftreten, das heißt Daten, die von einem Laufwerk auf eine Diskette geschrieben wurden, sind unter Umständen mit anderen Laufwerken nicht lesbar.

Inzwischen gibt es eine Reihe weiterer diskettenartiger Speichermedien, die jeweils eigene Laufwerke benötigen und weniger verbreitet sind. Weiterhin gibt es Rechner wie den iMac, die ohne Laufwerk geliefert werden.

Schon aufgrund der umständlichen Handhabung und der geringen Lesegeschwindigkeit erscheinen Disketten wenig geeignet. Die Kontrolle, zum Beispiel am Eingang zu einem Kino, würde viel zu lange dauern. Hinzu kommt die fehlende Standardisierung. Disketten scheiden deshalb als Medium aus.

4.1.2 Papier

Ebenfalls fast alle Computer sind an einen Drucker angeschlossen. Die Spanne reicht vom betagten Nadel- bis zum modernen Laserderucker. Typenraddrucker und ähnliche Geräte, die ausschließlich Text zu Papier bringen können, sind heute praktisch nicht mehr anzutreffen, ebenso 9-Nadel-Drucker. Unter den übrigen Geräten markieren 24-Nadel-Drucker die untere Grenze der Druckqualität.

Wir können also guten Gewissens davon ausgehen, daß der Kunde Text und Graphik wenigstens im Schwarz/weiss-Druck mit der Qualität eines 24-Nadlers zu Papier bringen kann. Derart gespeicherte Informationen können sowohl von Menschen als auch von einem Computer gelesen werden. Um gedruckte Informationen maschinell lesen und verarbeiten zu können, sind allerdings besondere Vorkehrungen nötig, etwa die Darstellung in geeigneter Schrift oder als Barcode.

Der Drucker als Ausgabe- und damit Papier als Speichermedium ist grundsätzlich geeignet. Bisherige Tickets sind schließlich auch aus Papier. Bedingung für den Einsatz dieses Mediums ist die Möglichkeit, eine genügend große Datenmenge maschinenlesbar auf einem handlichen Stück Papier unterzubringen. Gelingt das, so entstehen auf der Kundenseite kaum Probleme. Der Drucker ist einfach da, Papier auch und aus dem Gerät kommt etwas, was der Käufer schon kennt und was er anfassen und lesen kann.

Zur Datenrepräsentation auf Papier eignen sich zum einen Schriften, die für die optische Erkennung durch Computer optimiert sind. Trotz aller Optimierung bleibt diese Methode fehleranfällig. Außerdem gibt es keine plattformunabhängige Möglichkeit, solche Schriften aus dem Web-Browser heraus zu drucken. Die Alternative kennt jeder aus dem Supermarkt. Strichkodes sind dort seit langem im Einsatz und die Lesetechnik funktioniert augenscheinlich gut. Das Manko der sehr geringen Kapazität – ein EAN-Produktkode enthält gerade 13 Ziffern – ist seit der Entwicklung zweidimensionaler Codes behoben.

4.1.3 Magnetstreifenkarten

Magnetkarten besitzen gegenüber Disketten eine wesentlich geringere Speicherkapazität, können dafür aber schneller gelesen werden. Zugangskontrollsysteme auf Magnetkartenbasis sind verfügbar und funktionieren. Schreibgeräte für diese Karten stehen allerdings kaum einem Computeranwender zur Verfügung. Damit scheidet dieser Träger aus.

4.1.4 Chipkarten

In jüngster Zeit verbreiten sich zunehmend Chipkarten. Sie können grob in Speicher- und Prozessorkarten unterteilt werden [Kab96]. Eine Speicherkarte besteht im wesentlichen aus einem nichtflüchtigen Speicher, dessen Inhalt von außen verändert werden kann, allerdings nur durch solche Operationen, die die Schaltung der Karte erlaubt. So kann etwa der gespeicherte Wert einer Telefonkarte nicht erhöht werden.

Prozessorkarten (Smartcards) enthalten neben diesem Speicher noch einen programmierbaren Microcontroller und damit einen Computer, dem nur noch Stromversorgung

und Benutzerschnittstelle fehlen [FRW98]. Er besitzt ein Betriebssystem, das unter anderem ein hierarchisches Filesystem für Daten und Programme verwaltet. Krypto-Karten enthalten darüber hinaus einen Koprozessor, der Funktionen für die asymmetrische Kryptographie schnell ausführen kann. Auf kontaktlose Chipkarten, wie sie in einigen Städten zur Zahlung im Nahverkehr eingesetzt werden, kann aus bis zu einem Meter Abstand zugegriffen werden [FRW98, Mei97].

Chipkarten besitzen zwei Eigenschaften, die sie für den Einsatz in Zahlungs- und anderen Werttransfersystemen interessant machen. Zum einen können sie Daten einigermaßen manipulationssicher speichern. Man kann die Karte deshalb beruhig einem potentiellen Betrüger anvertrauen, wenn die sensiblen Daten erst auf dem Chip sind (dorthin gelangen sie bei Telefonkarten zum Beispiel während der Herstellung).

Zum anderen sind Smartcards mobile Computer, so daß die Anwendungsmöglichkeiten weit über einfaches Vorzeigen hinausreichen. So können diese Karten etwa PINs oder biometrische Merkmale prüfen und auf diese Weise unrechtmäßige Benutzer erkennen.

Schnittstellen zur Anbindung von Smartcards an den PC sind verfügbar, aber noch nicht sehr verbreitet. Die Standardisierung ist vor allem auf der Softwareseite noch nicht abgeschlossen [FRW98]. Die Benutzung von Chipkarten am PC erfordert besondere Programme; ein gewöhnlicher Web-Browser genügt nicht.

Der Besitzer kann einer Chipkarte nicht ansehen, was auf ihr gespeichert ist. Das weckt zum einen allgemeines Unbehagen, zum anderen macht es die Benutzung umständlich, wenn sich die Informationen auf der Karte häufig ändern. Man muß den Inhalt zusätzlich auf Papier drucken oder jedesmal einen Computer bemühen, um ihn zu lesen. So ließe sich beispielsweise die Sitzplatznummer nicht einfach von der Karte ablesen, wenn das Billett nur auf dem Chip gespeichert ist.

Vor der ersten Benutzung muß sich der Kunde eine Chipkarte besorgen, und zwar für jeden Veranstalter eine eigene. Blankokarten, die man ähnlich wie Disketten auf Vorrat kaufen und dann für verschiedene Zwecke nutzen könnte, gibt es nicht. (Solche Karten wären nicht manipulationssicher, könnten aber immer noch als mobiler Kleincomputer dienen.) Eine Universalkarte wird es vermutlich auch in ferner Zukunft nicht geben [Riv97].

Die Chipkarte scheint als Medium zunächst verlockend, zumal wir damit auch Fahrkarten für die Bahn im Internet verkaufen könnten. Bei näherem Hinsehen überwiegen jedoch die Nachteile. Spontankäufe sind nicht möglich; der Kunde braucht besondere Hard- und Software, die noch nicht weit verbreitet ist; die Kartenflut wird Müllberge nicht weg- sondern hinspülen [Kab96] und der Kunde braucht einen Computer, sobald er sich nicht mehr sicher ist, was er gekauft hat. Bei der Ticketkontrolle bereiten Chipkarten zudem Handhabungsprobleme, wenn sie umständlich in den Schlitz eines Lesegeräts eingeführt werden müssen. Kontaktlose Karte können dieses Problem lösen, aber mit ihnen kann der PC des Kunden erst recht nichts anfangen.

4.2 Strichkode

Strichkodes (Barcodes) werden vor allem dort eingesetzt, wo Objekte automatisch oder halbautomatisch identifiziert werden müssen. Erste Anwendungen in der Industrie gab

es bereits in den sechziger Jahren, und Mitte der 70er wurden UPC (Universal Product Code) und EAN (European Article Numbering) eingeführt [Pal95], die man heute auf fast jedem Produkt findet. Typische Einsatzgebiete sind neben dem Handel zum Beispiel Bibliotheken und die Verwaltung von Lagerbeständen in der Industrie.

Klassische, eindimensionale Barcodes kodieren einige Zeichen, meist weniger als 20 aus einem kleinen Zeichenvorrat, als Folge von Strichen und Leerräumen. Grundelement ist ein Modul, das ist ein Strich einer gewissen Breite, der entweder schwarz oder weiß sein kann. Der Kode legt fest, wie die Zeichen eines Zeichenvorrates, etwa Ziffern oder alphanumerische Zeichen, auf Modulfolgen fester Länge abgebildet werden. Mehrere aufeinanderfolgende Module gleicher Farbe ergeben breitere Striche oder Lerräume.

Um die so abgelegten Daten zu lesen, werden sie linear abgetastet, im einfachsten Fall mit einem stiftartigen Gerät, das der Benutzer über den Strichkode bewegt. Dabei wird das Symbol beleuchtet und von einer Photozelle die Intensität des reflektierten Lichts gemessen. So entsteht ein analoger Spannungsverlauf, aus dem durch Quantisierung die gelesenen Daten gewonnen werden [Pal95]. Statt eines solchen Stiftes werden heute meist Geräte eingesetzt, die einen Laserstrahl über das Symbol führen und das reflektierte Licht auffangen.

Der Lesevorgang ist fehlerträchtig. Vor allem unterschiedliche Strichbreiten sind schwer zu unterscheiden. Deshalb werden zum einen Prüfziffern zu den Daten hinzugefügt, zum anderen ist die Kodierung stark redundant. Ein EAN-Symbol zum Beispiel stellt jede der Ziffern 0 bis 9 durch einen 7-Bit-Kode dar. Das ermöglicht die Erkennung von Lesefehlern mit ausreichender Sicherheit. Toleranz gegenüber Beschädigungen erreicht man durch die Höhe des Symbols, also die Länge der Striche.

Da sich die Kapazität klassischen Barcodes nicht beliebig steigern läßt, werden seit 1987 auch zweidimensionale Kodes eingesetzt. Stapelkodes wenden im wesentlichen das herkömmliche Verfahren an, verteilen den Strichkode aber auf mehrere Zeilen. Sie können deshalb auch mit herkömmlichen Methoden gelesen werden. Matrixkodes legen die Informationen hingegen als zweidimensionales Muster von Punkten, Quadraten oder anderen Polygonen ab. Um sie zu lesen, braucht man CCD-Kameras und Bildverarbeitung.

Dabei sind mehrere Probleme zu lösen. Das Symbol muß zunächst überhaupt im Bild gefunden werden. Danach ist seine Lage zu bestimmen. Die Grundelemente, meist Quadrate, manchmal auch Kreise oder Rechtecke, liefern, anders als die Striche von eindimensionalen und Stapelkodes, von sich aus keine Richtungsinformation. Danach erst können die gespeicherten Daten aus der Anordnung heller und dunkler Elemente gelesen werden.

Zweidimensionale Symbole sind anfälliger für Beschädigungen. Die Grundelemente können nicht mehr einfach in die Länge gezogen werden, um Fehlertoleranz zu erreichen. Statt dessen kodiert man die Daten in einem fehlerkorrigierenden Kode, bevor man daraus ein Symbol erzeugt.

Die meisten zweidimensionalen Kodes können wenigstens die 128 ASCII-Zeichen darstellen. Viele gestatten auch die Speicherung beliebiger Oktette. Um die begrenzte Kapazität besser auszunutzen, umfassen diese Kodes simple Kompressionsverfahren, mit denen ASCII-Text platzsparend abgelegt werden kann.

Tabelle 4.1 listet die wichtigsten zweidimensionalen Kodes auf und ihre maximalen

Kode	Kapazität			Typ	Bemerkungen
	ASCII	Ziffern	Oktette		
Aztec	3067	3832	1914	Matrix	
Code 1	2218	3550	-	Matrix	
Code 16K	77	154	-	Stapel	
Code 49	49	81	-	Stapel	erster 2D-Kode (1987)
Data Matrix	2000	-	-	Matrix	
MaxiCode	-	-	93	Matrix	sechseckige Elemente
PDF 417	1850	2710	1108	Stapel	am weitesten verbreitet
QR Code	4464	7366	-	Matrix	
SuperCode	4083	5102	2546	Stapel	

Tab. 4.1: *Zweidimensionale Kodes*

Speicherkapazitäten auf. Die Daten sind [Ada98, Pal95, AIM94] entnommen. Für einen davon müssen wir uns entscheiden.

Wir erinnern uns: In Kapitel 3 wurden drei Authentifikationsverfahren diskutiert. Die meisten Daten fallen bei einer RSA-Signatur an, zum Beispiel 128 Bytes bei einer Schlüsellänge von 1024 Bit. Die Ticketdaten aus Kapitel 2 nehmen etwa noch einmal so viel in Anspruch, wenn wir alle denkbaren Parameter aufnehmen. Code 16K, Code49 und MaxiCode sind also wenig geeignet.

Wir müssen Binärdaten transportieren. Der Kode sollte deshalb beliebige Oktettfolgen darstellen können und nicht nur ASCII-Zeichen. Das reduziert die Auswahl auf SuperCode, PDF 417 und Aztec. SuperCode scheint nicht sehr verbreitet zu sein. Einzig in [Ada98] findet er Erwähnung. Lesegeräte und Software zur Erzeugung sind kaum zu bekommen.

Für PDF 417 spricht, daß dieser Kode weit verbreitet ist und als Stapelkode mit herkömmlicher[23] Lasertechnik gelesen werden kann. Im Gegensatz dazu benötigt der Matrixkode Aztec einen CCD-Scanner. Er ist eine recht junge Entwicklung der Firma Welch Allyn, aus der erst im Oktober 1997 eine offizielle AIM-Spezifikation[24] wurde. Andererseits ist die Aztec-Spezifikation [WAl97] als einzige frei erhältlich. Zwar können auch andere Kodes frei benutzt werden, aber dazu muß man die Spezifikationen bei AIM kaufen oder fertige Software einsetzen. Das erwies sich als Problem, denn die Auswahl an Software ist beschränkt. Die wenigen Bibliotheken, mit denen sich die Erzeugung von 2D-Kodes sauber in Anwendungen integrieren läßt, gibt es nur für Windows NT, ein System, das – vgl. 3.5 – für unsere Anwendung unbrauchbar ist. Davon abgesehen wäre schon allein die Festlegung auf eine einzelne Plattform ein Fehler, den man später bereuen wird.

Daher war es unvermeidlich, selbst einen Symbolgenerator zu implementieren, ob-

[23] Herkömmlich jedenfalls was die Funktionsweise betrifft. Einige Modifikationen sind jedoch unumgänglich.

[24] AIM steht für Automatic Identification Manufacturers. URL: `http://www.aim-europe.org/`.

gleich der Autor das drohende NIH-Syndrom[25] erkennt. Ursprünglich war die Verfügbarkeit der Spezifikation der einzige Grund, sich für Aztec zu entscheiden, doch die Wahl eines Matrix-Kodes erscheint auch aus einem anderen Grund als sinnvoll.

Der Käufer soll seine Eintrittskarte nach dem Kauf selbst ausdrucken. Damit das Papier nicht zum Datengrab wird, muß es später bei der Kontrolle sicher lesbar sein und die Druckqualität ist ein Faktor, der die Lesbarkeit beeinflußt. Wir müssen eine ausreichende Qualität sicherstellen, ohne den Druckvorgang selbst beeinflussen zu können.

Die Elemente sollten in jeder Richtung mehrere (Drucker-)Pixel umfassen, auch wenn sie auf einem alten Nadeldrucker zu Papier gebracht werden. Für einen Stapelkode wie PDF 417 bedutet das, daß ein Modul einige Pixel breit sein muß. Die Zeilenhöhe beträgt mindestens das Dreifache der Modulbreite [AIM94]. Daraus ergibt sich eine Untergrenze der Datendichte – ein Element kann höchstens ein Bit repräsentieren. Beim Matrixkode tut es das auch, während PDF 417 immerhin 85 Module braucht, um 48 Bits darzustellen.

[Pal95] gibt an, daß mit einem Nadeldrucker Modulbreiten von 0,4 bis 0,5 Millimetern erreichbar sind. Bei einer Modulbreite von $0,5$mm benötigen 48 Bits in einem PDF-417-Symbol ungefähr $64mm^2$.

Ein Matrixkode stellt jedes Bit durch ein quadratisches Element dar. Auch das soll wenigstens einige Pixel breit sein, ist dann aber auch nur genauso hoch. Auf die Fläche, die beim Stapelkode der schmalste Strich einnimmt, passen bei einem Matrixkode mehrere Bits, wenn man gleiche Elementgröße verwendet. Man kann sie sogar sicherheitshalber verdoppeln und bringt immer noch mehr Daten auf der gleichen Fläche unter!

Selbst mit Elementen von 1mm Kantenlänge – jeder Nadeldrucker kann Quadrate dieser Größe in ausreichender Qualität erzeugen – nehmen 48 Bits in einem Aztec-Symbol eine kleinere Fläche ein als bei PDF 417, nämlich nur $48mm^2$.

PDF 417 benötigt außerdem Ruhezonen, das heißt unbedruckte Bereiche um das Symbol. Für Aztec-Symbole sind keine erforderlich. Die PDF-417-Spezifikation ist in einigen Punkten exakter als die des Aztec-Kodes, dadurch aber fast unlesbar.

4.3 Der Aztec-Kode

4.3.1 Überblick

Ähnlich wie ein Rechnernetz, in dem mehrere unabhängige Protokollschichten aufeinander aufbauen, enthält der Aztec-Kode drei Ebenen. Die Nachrichtenschicht erzeugt aus der Eingabe, die eine beliebige Oktettfolge sein kann, in eine Folge von 6, 8, 10 oder 12 Bits langen Zeichen. Die zweite Ebene erhält diese Zeichenfolge als Eingabe und fügt weitere Zeichen zur Fehlerkorrektur hinzu. Außerdem entsteht hier die Mode Message, die dem Header eines Datenpaketes im Netz vergleichbar ist. Im dritten Schritt schließlich wird aus der Mode Message, den Nutzdaten und den Korrekturinformationen die graphische Darstellung des Symbols erzeugt.

Der Empfänger dekodiert in umgekehrter Reihenfolge. Zunächst gewinnt er aus einem Bild eine Zeichenfolge. Anhand der Korrekturinformationen findet und beseitigt er Fehler,

[25] NIH ist die Abkürzung für »Not Invented Here«.

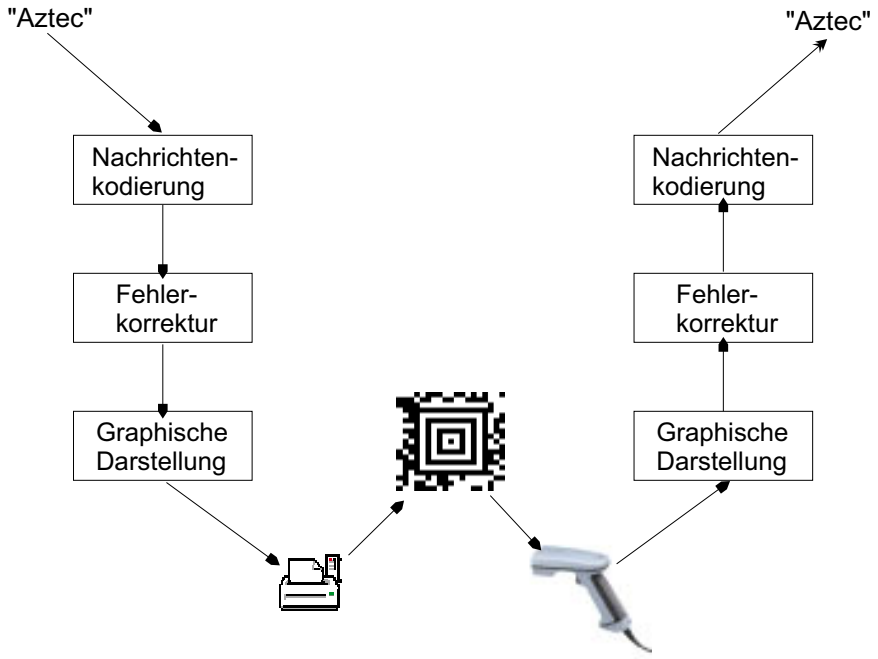

Abb. 4.1: *Aztec scheibchenweise*

die durch Beschädigung des Symbols oder bei der Bildverarbeitung entstanden sind. Die rekonstruierte Zeichenfolge wird schließlich zur Nachricht dekodiert.

Aztec-Symbole sind quadratisch und können 32 Größen annehmen. Die Zeichenlänge in Bits ist von der Symbolgröße abhängig. Die Zahl der Korrekturzeichen wird in der zweiten Schicht immer so gewählt, daß Daten- und Korrekturzeichen zusammen genau eine der 32 Symbolgrößen ausfüllen.

Das größte Aztec-Symbol nimmt bis zu 1914 Bytes, 3067 Buchstaben oder 3832 Ziffern auf [AIM97]. Es kann unabhängig von seiner Orientierung gelesen werden. Für kurze Nachrichten (bis 64 Bytes) steht die Variante *Small Aztec* zur Verfügung, die das Symbol etwas kompakter macht. Lange Nachrichten lassen sich mit dem *Ordered Append Protocol* auf bis zu 26 Symbole verteilen.

4.3.2 Nachrichtenkodierung

Der erste Schritt bei der Erzeugung eines Aztec-Symbols ist die Kodierung der Eingabe in eine Zeichenfolge. Sie geschieht in zwei Teilschritten. Zunächst wird aus der Eingabe eine Bitfolge erzeugt.

Zur Kodierung von Oktetten, die ASCII-Zeichen repräsentieren,[26] dienen fünf Tabellen mit den Namen *Upper* für Großbuchstaben, *Lower* für Kleinbuchstaben, *Mixed* für Steuer- und einige Sonderzeichen, *Punct* für alle anderen Steuerzeichen und *Digit* für

[26] Wer sich über die sperrige Ausdrucksweise wundert, sei noch einmal auf [Kor98] hingewiesen.

Ziffern. Jede umfaßt 32 Einträge mit Ausnahme der Tabelle *Digit*, die nur halb so lang ist.

Upper				Digit	
Kode	Zeichen	Kode	Zeichen	Kode	Zeichen
00000	*PS*	10000	O	0000	*PS*
00001	space	10001	P	0001	space
00010	A	10010	Q	0010	0
00011	B	10011	R	0011	1
00100	C	10100	S	0100	2
00101	D	10101	T	0101	3
00110	E	10110	U	0110	4
00111	F	10111	V	0111	5
01000	G	11000	W	1000	6
01001	H	11001	X	1001	7
01010	I	11010	Y	1010	8
01011	J	11011	Z	1011	9
01100	K	11100	*LL*	1100	,
01101	L	11101	*ML*	1101	.
01110	M	11110	*DL*	1110	*UL*
01111	N	11111	*BS*	1111	*US*

Bei den *Umschaltkodes* bezeichnet der erste Buchstabe die Zieltabelle (B = binary) und der zweite die Art der Umschaltung (S = shift, L = latch).

Tab. 4.2: *Die Tabellen Upper und Digit*

Die meisten Einträge ordnen einem ASCII-Zeichen einen 5- oder 4-Bit-Kode zu; die Namen deuten an, welche Zeichen in welcher Tabelle zu finden sind. Einige Zeichen sind in mehreren Tabellen enthalten, um Umschaltungen zu vermeiden. Die Tabelle *Punct* enthält vier Kodes für Doppelzeichen wie `CR LF`.[27] Die übrigen 5- oder 4-Bit-Kodes dienen zum Umschalten zwischen den Tabellen. Die Umschaltung kann nur für das nächste Zeichen (»shift«) oder dauerhaft (»latch«) wirksam sein. Jede Tabelle enthält jedoch nur einen Teil der möglichen Umschaltkodes, so daß der Wechsel zwischen zwei Tabellen zum Teil eine oder manchmal sogar zwei Zwischenstationen braucht.

Für natürlichsprachigen Text oder Daten, die nur Zeichen aus einer einzigen Tabelle enthalten, liefert das Verfahren mit einfachen Mitteln eine gute Kompression. Jede Ziffer wird mit vier Bits dargestellt, fast[28] jedes andere ASCII-Zeichen mit fünf Bits und gelegentlich ist eine Umschaltung nötig, die vier oder fünf Bits Overhead mit sich bringt. Sind dagegen viele Tabellenwechsel nötig oder solche, die nur über Umwege möglich sind, ist diese Kodierung unbrauchbar und verlängert die Nachricht unnötig.

Für solche Daten und für Oktette, die in keiner der Tabellen enthalten sind, gibt es den Binärmodus (»Binary Shift«). In ihm werden Oktette aus der Eingabe unverändert

[27] Carriage Return und Line Feed – Wagenrücklauf und Zeilenvorschub.
[28] Die Tabelle *Mixed* enthält nur einen Teil der ASCII-Steuerzeichen.

übernommen. Dem Oktettstrom wird seine Länge in fünf oder elf Bits kodiert vorangestellt. Der Binärmodus wird über einen Umschaltkode aktiviert und kann so für Teile einer Nachricht verwendet werden.

Im zweiten Schritt wird die Bitfolge, die auf diese Weise entstanden ist, in eine Folge von Zeichen[29] überführt. Die Bitlänge B der Zeichen hängt von der Symbolgröße ab. Die Umwandlung ist simpel: Beginnend beim höchstwertigen werden jeweils B Bits der Bitfolge in ein B-Bit-Zeichen überführt. Keines der Zeichen soll aber ausschließlich Nullen oder Einsen enthalten (so sind später Auslöschungen erkennbar). Sind deshalb die ersten $B-1$ Bits des Zeichens alle 0, erhält das letzte Bit des Zeichens deshalb den Wert 1 und umgekehrt. Mit dem nächsten Bit der Bitfolge und dem nächsten Zeichen geht es dann wie gewohnt weiter. Das letzte Zeichen wird mit Einsen aufgefüllt und gegebenenfalls sein niederwertigstes Bit wieder auf 0 gesetzt.

Die so gebildete Folge $d_1 d_2 \ldots d_D$ von D Zeichen wird an die Fehlerkorrekturschicht weitergegeben.

4.3.3 Fehlerkorrektur

Das Arbeitsergebnis der ersten Schicht wird um Korrekturinformationen ergänzt. Die Zahl der Korrekturzeichen ergibt sich aus der gewählten Symbolgröße. Ein Symbol der Größe $L = 1 \ldots 32$ kann insgesamt C_L Zeichen der Bitlänge B aufnehmen. Diese Werte legt die Spezifikation fest. Mit den Korrekturzeichen wird die Nachricht so aufgefüllt, daß sie genau in ein Symbol – das muß nicht das nächstgrößere sein – paßt. Also werden $K = C_L - D$ Korrekturzeichen hinzugefügt.

Die Korrekturzeichen werden durch systematische Reed-Solomon-Kodierung über den endlichen Körper $GF(2^B)$ (B ist die Zeichenlänge in Bits) gewonnen. Den Zusammenhang zwischen der Symbolgröße, der Zeichenlänge und dem benutzten Körper zeigt Tabelle 4.3. Die angegebenen Minimalpolynome definieren die Multiplikation in $GF(2^B)$.

Symbol-größe	Zeichen-länge B	Körper	Minimalpolynom
Mode Message	4 Bit	$GF(16)$	$x^4 + x + 1$
1-2	6 Bit	$GF(64)$	$x^6 + x + 1$
3-8	8 Bit	$GF(256)$	$x^8 + x^5 + x^3 + x^2 + 1$
9-22	10 Bit	$GF(1024)$	$x^{10} + x^3 + 1$
23-32	12 Bit	$GF(4096)$	$x^{12} + x^6 + x^5 + x^3 + 1$

Tab. 4.3: *Endliche Körper für die Fehlerkorrektur [WAl97]*

Die Datenzeichen $d_1 d_2 \ldots d_D$ faßt man als Koeffizienten eines Polynoms $d(x) = d_1 x^{D-1} + d_2 x^{D-2} + \ldots + d_D$ über $GF(2^B)$ auf. Es ist vom Grad D, denn die Nachrichtenschicht garantiert, daß d_1 von null verschieden ist. Durch Multiplikation mit x^K

[29] Die Bezeichnung »Symbol« findet der Autor schöner, aber sie wäre hier mißverständlich. Die Spezifikation spricht von *code words*; der Autor versteht unter einem Kodewort aber eine Folge solcher Zeichen.

erhält man daraus Polynom $x^K d(x)$ vom Grad $K + D - 1 = C_L - 1$. Es hat C_L Koeffizienten; das ist gerade die Kapazität des Symbols.

Die RS-Kodierung erfolgt mittels eines Generatorpolynoms[30] $g(x) = (x - 2^1)(x - 2^2)\ldots(x - 2^K)$. Sein Grad K ist die Zahl der Korrekturzeichen, die hinzugefügt werden sollen. Dividiert man $x^K d(x)$ durch den Generator $g(x)$, so erhält man ein Restpolynom $r(x)$, das höchstens vom Grad $K - 1$ ist. Das wird von $x^K d(x)$ subtrahiert und man erhält ein Polynom $x^K d(x) - r(x)$, das ohne Rest durch das Generatorpolynom teilbar ist.

Die ersten D Koeffizienten von $x^K d(x) - r(x)$ sind die unveränderten Datenzeichen, die übrigen die Korrekturzeichen. Alle zusammen füllen genau ein Symbol und werden der nächsten Schicht zur graphischen Darstellung übergeben.

Mit K Korrekturzeichen können $\frac{K}{2}$ Zeichenfehler korrigiert werden. Die Spezifikation fordert wenigstens fünf Korrekturzeichen; empfohlen werden etwa 25% der Symbolkapazität [AIM97, WAl97]. Mit der Wahl verschiedener Zeichenlängen wird die Leistung der Fehlerkorrektur an die Symbolgröße angepaßt.

Auf eine ausführliche Darstellung wird hier verzichtet. Literatur zum Thema gibt es genug [MWSl77, RaFu89, Schu91, Lyp97] und die funktionierende Implementierung muß als Verständnisnachweis genügen.

Neben den Korrekturzeichen wird eine 16 Bit lange Mode Message erzeugt. Sie enthält die Symbolgröße $L = 1\ldots32$ und in den übrigen 11 Bits die Zahl D der Zeichen, die Nutzdaten enthalten, beide um eins verringert. Die Mode Message wird als Folge von vier 4-Bit-Zeichen aufgefaßt und um sechs Zeichen für die Fehlerkorrektur ergänzt. Das Verfahren ist dasselbe wie für die Nutzdaten.

4.3.4 Graphik

Zur graphischen Darstellung wird die Zeichenfolge aus dem zweiten Schritt wieder zerlegt, diesmal in »Dominosteinchen« von je zwei Bit. Dabei wird sie gleichzeitig umgekehrt. Die niederwertigen Bits des letzten Korrekturzeichens werden zum ersten Dominostein der neuen Folge.

Neben Mode Message, Nutzdaten und Korrekturzeichen enthält das Symbol einige feste Elemente. In der Mitte befindet sich der Finder ein Muster aus sieben ineinander geschachtelten Quadraten, die abwechselnd schwarz und weiß sind. Beim Lesen eines Aztec-Symbols wird zuerst dieses Muster im aufgenommenen Bild gesucht. An jeder Ecke des Finder-Musters sind drei Orien-tat-ion-Bits untergebracht, an denen der Leser die Lage des Symbols erkennen kann. Den dritten festen Bestandteil eines jeden Symbols bildet ein Referenzgitter aus abwechselnd schwarzen und weißen Elementen. Damit können beim Lesen Verzerrungen erkannt werden.

Rund um das Finder-Muster ist im Uhrzeigersinn die Mode Message abgelegt. Die Domino-Folge aus Nutzdaten und Korrekturzeichen liegt, an den oberen linken Orientation-Bits beginnend, ebenfalls im Uhrzeigersinn, aber sprialförmig um den Finder. Schwarze Elemente repräsentieren die binäre 1, weiße die 0. Das höherwertige Bit der Dominosteine

[30] Die Zahl 2 in $(x-2^n)$ bezeichnet das Element von $GF(2^B)$, das durch die Bitfolge $0\ldots010$ repräsentiert wird.

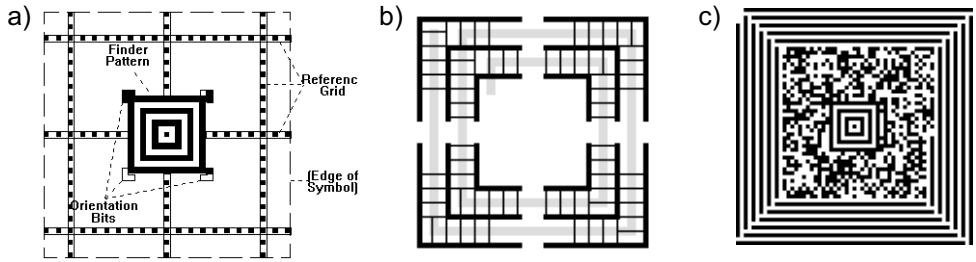

Abb. 4.2: *Aztec-Symbol. a) Feste Elemente; b) Anordnung der Dominosteine; c) Nutzdaten und Korrekturzeichen. Quelle (a+b): [WAl97]*

liegt immer auf der dem Finder abgewandten Seite. Elementpositionen, die bereits vom Referenzgitter belegt sind, werden übersprungen. Die Dominosteine können dabei sogar zerteilt werden.

Wegen der Umkehrung der Reihenfolge nehmen die zusätzlichen Korrekturzeichen die Fläche unmittelbar um den Finder ein und die Nutzdaten den äußeren Teil des Symbols. Das erste Datenbit findet sich schließlich in der linken oberen Ecke des Symbols wieder. Dort können einige Dominosteine frei bleiben, wenn sie nicht mehr nausreichen, ein weiteres Zeichen abzulegen.

Abbildung 4.2c zeigt ein Symbol, in dem der Text TITITITITITITI... kodiert ist. Die Nachrichtenschicht macht daraus die Bitfolge 101010101010101010... (vgl. Tabelle 4.2), die die regelmäßigen Strukturen im äußeren Teil ergibt. Die Korrekturinformationen heben sich davon deutlich ab.

4.4 Implementierung

Die Suche nach geeigneten 2D-Symbolgeneratoren brachte nur ein spärliches Ergebnis. Die gewünschte plattformunabhängige Bibliothek war nicht aufzutreiben. Aus diesem Grund entstand eine eigene Implementierung. Sie ist in einigermaßen portablem ANSI-C geschrieben und läuft ohne Anpassungen auf verschiedenen Unix-Systemen (Solaris, AIX, Linux).

Hauptbestandteil des entwickelten Symbolgenerators ist die Bibliothek *libaztec*. Sie enthält alle Funktionen, die zur Symbolerzeugung notwendig sind, und liefert das Ergebnis ihrer Arbeit in einer Datenstruktur, die in beliebige Ausgabeformate, wie etwa Postscript und GIF, überführt werden kann.

Der Aufbau der Bibliothek orientiert sich an den drei Schichten der Spezifikation [WAl97]. Jeder Schicht ist ein eigenes Programmodul gewidmet.

Die meisten Funktionen der Bibliothek geben einen Zeiger zurück. Im Falle eines Fehlers ist das ein Nullpointer und die globale Variable `az_errno` enthält die Fehlernummer. Nimmt eine Funktion Flags als Parameter entgegen und gibt es deren mehrere an dieser Stelle gültige, so können sie durch das bitweise Oder verknüpft werden.

Name	Aufgabe
`az_bitstream`	kodiert die Eingabenachricht in eine Bitfolge
`az_mksymbol`	RS-Kodierung, Festlegung der Symbolgröße, Mode Message
`az_drawsym`	graphische Darstellung als Bitmap zur Weiterverarbeitung
`az_bmalloc`	Speicher für Bitmap reservieren
`az_bmfree`	Bitmap-Speicher freigeben
`az_getpixraw`	Bitmap-Pixel lesen
`az_perror`	Ausgabe von Fehlermeldungen
`az_setelem`	Bitmap-Manipulation

Tab. 4.4: *Überblick: API der Bibliothek libaztec*

4.4.1 Nachrichtenkodierung

Das Modul `msglevel` stellt die Funktion `az_bitstream()` zur Verfügung. Sie übernimmt im wesentlichen die Kodierung der ersten Schicht, liefert jedoch nur die Bitfolge zurück, die dabei als Zwischenprodukt entsteht. Die Abbildung dieser Bitfolge in eine Zeichenfolge erfolgt zusammen mit der Fehlerkorrektur. Das ist sinnvoll, da die Zeichenlänge von der Symbolgröße abhängt, welche ihrerseits an die Datenmenge und die gewünschte Zahl von Korrekturzeichen angepaßt werden muß.

Die Funktion `az_bitstream()` ist wie folgt definiert:

```
az_bitstr* az_bitstream( az_buffer buffer, unsigned int flags );
```

Die Struktur `buffer` enthält einen Zeiger auf die zu kodierende Nachricht sowie Längenangaben für Nachricht und Speicherbereich. Der zweite Parameter nimmt Flags auf, die den Kodiervorgang beeinflussen. Das Flag `AZ_FLAG_ALLBIN` erzwingt die Kodierung der gesamten Nachricht im Binary-Shift-Modus. Ohne dieses Flag werden nur die Zeichen im Binärmodus kodiert, bei denen das notwendig ist. Mit dem Flag `AZ_FLAG_TRUNC` wird das Kürzen zu langer Nachrichten erlaubt. Ist es nicht angegeben, führen zu lange Nachrichten zu einem Fehler. Zurückgegeben wird ein Zeiger auf eine Struktur, die eine Bitfolge nebst Längenangabe enthält.

Schwierigkeiten bereiteten die Ausnahmen in den Kodiertabellen und die Tatsache, daß nicht beliebig zwischen den Tabellen umgeschaltet werden kann.

Die Kodierung beginnt nach Spezifikation [WAl97] immer im *Upper*-Modus. Die Eingabe wird Oktett für Oktett bearbeitet und für jedes die interne Funktion `transascii()` aufgerufen. Sie erhält neben dem aktuellen Zeichen dessen Nachfolger sowie die momentan gültige Tabelle als Parameter. Kann das Zeichen mit der aktuellen Tabelle kodiert werden, gibt sie den Zeichenkode zurück und es geht mit dem nächsten Zeichen weiter. Ist jedoch erst eine Umschaltung nötig, liefert sie den Modus, in den geschaltet werden muß, sowie den Umschaltkode. Im neuen Modus geht es dann nicht mit dem nächsten, sondern dem eben schon mal bearbeiteten Zeichen weiter.

4.4.2 Fehlerkorrektur

Die Bitfolgen, die das Modul `msglevel` liefert, werden im Modul `ecclevel` weiterverarbeitet. Das ist ein wenig aufregender als die Nachrichtenkodierung der ersten Schicht, die zu einem Gutteil aus Kopieren und nachschlagen in Tabellen besteht.

Nach außen ist nur die Funktion `az_mksymbol()` sichtbar, die so definiert ist:

```
az_symdata *
az_mksymbol( const az_bitstr *indata,
             unsigned short maxlayers,
             unsigned short ecclevel,
             unsigned flags
);
```

Zu der Bitfolge aus der Nachrichtenschicht (`*indata`) kommen wieder Flags hinzu, außerdem zwei Parameter, die den Umfang der Fehlerkorrektur und die Symbolgröße bestimmen. Die maximale Größe wird in `maxlayers` angegeben. Der Parameter `ecclevel` bestimmt die Mindestzahl von Korrekturzeichen. Er wird als Prozentangabe bezüglich der Nutzdatenmenge interpretiert. Ist die RS-kodierte Nachricht zu lang für die geforderte Symbolgröße, verweigert die Funktion den Dienst und gibt einen Nullpointer zurück.

Die einzige Ausnahme: Ist der Parameter `ecclevel` mit 0 angegeben, werden zunächst 50% versucht, bei Bedarf aber reduziert. Paßt es dann immer noch nicht, gibt es wieder einen Fehler, der durch einen Nullpointer als Rückgabewert angezeigt wird. Das ist ein schneller Hack; besser wäre es, die Entscheidung zwischen Symbolgröße und Fehlerkorrektur mit Flags zu beeinflussen. Um die Programmierschnittstelle konsistent zu halten, wird auch beim Parameter `maxlayers` der Wert 0 akzeptiert; er ist gleichbedeutend mit der Angabe 32 für das größte Symbol, das die Spezifikation erlaubt.

In der aktuellen Version[31] der Bibliothek ist jedoch `AZ_FLAG_MENU` das einzige zugelassene Flag. Es macht aus dem erzeugten Symbol ein sogenanntes *Menu*-Symbol. Das ist ein Symbol, dessen Inhalt nach der Dekodierung vom Lesegrät interpretiert und nicht an den angeschlossenen Rechner weitergegeben wird. Auf diese Weise können Einstellungen des Scanners verändert werden, ohne daß besondere Software oder überhaupt ein Computer notwendig ist.

Zur Reed-Solomon-Kodierung wird in $GF(2^B), B \in \{4, 6, 8, 10, 12\}$ gerechnet. Addition und die Subtraktion sind einfach; beide entsprechen dem bitweisen XOR der Bitfolgen, die die Zeichen repräsentieren. Die Multiplikation ist komplizierter. Die endlichen Körper $GF(2^B)$ besitzen jeweils ein primitives Element α. Dieses ist Erzeugendes der multiplikativen Gruppe von $GF(2^B)$, das heißt jedes von null verschiedene Element von $GF(2^B)$ kann als Potenz von α dargestellt werden und es ist $\alpha^{2^B-1} = 1$. Kennt man diese Darstellung für zwei Elemente $a = \alpha^m$ und $b = \alpha^n$, kann man leicht multiplizieren: $a \cdot b = \alpha^m \cdot \alpha^n = \alpha^{(m+n) \ mod \ (B-1)}$.

Was fehlt ist die Abbildung der – durch Bitfolgen der Länge B repräsentierten – Zeichen auf Exponenten von α und umgekehrt. Sie wird mit Hilfe des Minimalpolynoms

[31] 0.1

von α gewonnen. Wie in Tabelle 4.3 zu sehen ist, hat es für $GF(2^B)$ gerade den Grad B und ist normiert, hat also die Form $x^B + p(x)$. Der grad von $p(x)$ ist kleiner als B.

Das primitive Element α ist Nullstelle seines Minimalpolynoms und aus $\alpha^B + p(\alpha) = 0$ folgt unmittelbar die Beziehung $\alpha^B = p(\alpha)$ (Addition und Subtraktion sind in den betrachteten Körpern identisch).

Setzt[32] man $1 = \alpha^0 = \alpha^{2^B-1} = 00\ldots001$, $\alpha^2 = \alpha \cdot \alpha = 00\ldots010$, $\alpha^3 = \alpha \cdot \alpha^2 = 00\ldots100$ und so weiter bis $\alpha^{B-1} = 10\ldots00$, kann man jedes Element von $GF(2^B)$ als Summe von Potenzen $\alpha^0 \ldots \alpha^{B-1}$ darstellen. Für $\alpha^0 \ldots \alpha^{B-2}$ (und damit für jede Summe dieser Elemente) ist auch die Multiplikation mit α erklärt, denn das zur nächsthöheren nächsthöheren Potenz gehörende Zeichen ist bekannt. Und $\alpha \cdot \alpha^{B-1}$ schließlich ist nichts anderes als $\alpha^B = p(\alpha)$.

Die Zuordnung zwischen Bitfolgen und Exponenten von α erhält man damit durch sukzessive Multiplikation $\alpha^i = \alpha \cdot \alpha^{i-1}$. Wann immer sich dabei auf der Seite der Bitfolgen α^B zeigt, wird es sogleich durch $p(\alpha)$ ersetzt. Auf diese Weise erhält man zwei Tabellen. Die eine, nennen wir sie `alpha`, enthält zu jedem Exponenten $e \in \{0, \ldots, (B-1)\}$ die Darstellung als Bitfolge, die andere, die auf den schönen Namen `poly` hört, umgekehrt zu jeder Bitfolge den entsprechenden Exponenten von α. Der Null ordnen wir den Exponenten $2^B - 1$ zu. Das wird uns keine Schwierigkeiten bereiten.

Wer sich nach dieser knappen Darstellung an »Mathematik zwischen Wahn und Witz« [Dud95] erinnert fühlt, sei auf die Literatur [MWSl77, Swe92, Schu91, RaFu89] verwiesen und mit einigen Zeilen Programmtext getröstet. In *C* aufgeschrieben, ist das nämlich alles viel einfacher.

```
unsigned short alpha[LANG_GENUG];
unsigned short poly[LANG_GENUG];
unsigned short minimalpolynom;
int i;

alpha[0] = 1;
for ( i = 1; i < (1<<B)-1; i++ )  {
    alpha[i] = alpha[i-1] << 1;
    if ( alpha[i] >= (1 << B) )
        alpha[i] ^= minimalpolynom;
    poly[alpha[i]] = i;
}
alpha[(1<<B)-1] = 0;
poly[0] = (1 << B) - 1;
```

Diese Zeilen finden sich so ähnlich – die Varialen sind anders benannt – im Quelltext der Aztec-Bibliothek.

Was tun sie? Eigentlich genau das oben Beschriebene. Die `for`-Schleife durchläuft einmal alle Exponenten von 1 bis $2^B - 1$. Die Multiplikation $\alpha \cdot \alpha^{i-1}$ ist nichts weiter

[32] Man kann das auch mit der Betrachtung von Bitfolgen als Polynome über $GF(2)$ und einer gehörigen Portion Algebra rechtfertigen. Will der Autor aber aus Platz- und Zeitgründen nicht.

als eine Linksverschiebung. Wir hatten die ersten Potenzen von α ja so gewählt, daß wir daraus jede Bitfolge durch Addition erhalten können, und für jeden einzelnen Summanden ist die Multiplikation mit α nur eine Verschiebung um eine Stelle. Mit Ausnahme eines einzigen: Taucht α^B irgendwo auf, soll es durch $p(x)$ ersetzt werden, dessen Grad kleiner ist als B. Die `if`-Verzweigung prüft, ob dieser Zustand beim Schieben eingetreten ist, und leitet gegebenenfalls alles Nötige in die Wege. Das ist nicht viel. Da die Addition und die Subtraktion in unseren endlichen Körpern dem bitweisen exklusiven Oder entsprechen, werden wir in einem einzigen Schritt das α^B los, das sich da eingeschlichen hat, und addieren gleichzeitig $p(\alpha)$. Die Variable `minimalpolynom` enthält die Binärdarstellung des Minimalpolynoms.

Die so entstehenden Tabellen – auch die größte für $GF(2^{12})$ braucht noch nicht allzu viel Speicherplatz – werden bei Bedarf erzeugt und dann zur erneuten Verwendung aufbewahrt. Eigenhändiges Nachschlagen erspart die Funktion `gfmul()`, die anhand der Tabellen multipliziert.

Die eigentliche Reed-Solomon-Kodierung nutzt nicht viel mehr als Schulwissen. Wie bereits erläutert, ist der Kodiervorgang eine Polynomdivision mit Rest. Die kann man nach wohlbekanntem Verfahren auf einem Blatt Papier ausführen. Das tut die Aztec-Bibliothek nicht, aber sie nutzt die gleiche Methode.

Die Polynome werden durch ihre Koeffizienten aus $GF(2^B)$ repräsentiert, die in einem Array abgelegt sind. Der Leitkoeffizient steht dabei an erster Position, trägt im Array also den Index 0. Für unsere Zwecke genügt das, denn die einzige Operation, die hier den Grad eines Polynoms erhöht, ist die Multiplikation $x^K d(x)$. Sie führt lediglich dazu, daß die Folge der Koeffizienten am anderen Ende um K Nullen erweitert wird.

Zur Division mit Rest wird der nächste Term des Quotienten bestimmt – genau genommen nur sein Koeffizient – und das Generatorpolynom mit ihm multipliziert. Das Ergebnis wird von dem Rest subtrahiert, den der vorhergehende Divisionsschritt gelassen hat. Der erste Schritt subtrahiert statt dessen von der Folge der Datenzeichen. Nach D – das war die Zahl der Datenzeichen – Divisionsschritten liegen die Koeffizienten eines Restpolynoms im Puffer, dessen Grad kleiner ist als der des Generators. Das ist auch schon alles.

Die Funktion `az_mksymbol()` liefert das Ergebnis ihrer Arbeit in einer Struktur vom Typ `az_symdata()` zurück. Sie enthält neben der Mode Message und den RS-kodierten Daten auch Informationen über die Symbolgröße, die in der Graphikschicht benötigt werden.

4.4.3 Graphik

Den letzten Schritt der Symbolerzeugung übernimmt das Modul `grphlevel`. Neben der Funktion `az_drawsym()`, die die eigentliche Aufgabe des Moduls erledigt, exportiert es `az_bmalloc()` und `az_bmfree()` zur Speicherverwaltung sowie `az_setelem()` und `az_getpixraw()` für den einfacheren Umgang mit den erzeugten Bitmaps.

Die Funktion

```
az_bitmap *az_drawsym( const az_symdata *sdata )
```

erhält als Parameter einen Zeiger, wie er im vorhergehenden Schritt von `az_mksymbol()` geliefert wird. Bei erfolgreicher Abarbeitung liefert sie ein einfaches Bitmap. Darin entspricht jedem quadratischen Grundelement des Symbols ein Pixel.

Die von `az_drawsym()` gelieferte Bitmap-Struktur kann von Anwendungsprogrammen beliebig weiterverarbeitet werden, etwa zur Ausgabe auf dem Bildschirm, auf dem Drucker oder in eine Datei. Sie wird deshalb genauer betrachtet.

```
typedef struct  {
    unsigned char *buf;     /* Bitmap im Speicher... */
    unsigned buflen;        /* ... und seine Gesamtlaenge */

    unsigned lines;         /* Bitmap-Groesse */
    unsigned cols;
    unsigned colbytes; /* Bytes pro Zeile */

    unsigned c_x; /* Symbolmittelpunkt */
    unsigned c_y;
} az_bitmap;
```

Koordinatenursprung ist die obere linke Ecke. Jedes Element wird durch ein Bit repräsentiert; schwarze durch 1 und weiße durch 0. Die Zeilen des Symbols liegen hintereinander im Speicher `*buf`. Jedes Byte wird dabei beginnend mit dem höchstwertigen Bit gefüllt. Eine neue Zeile beginnt immer an einer Bytegrenze, so daß am Zeilende einige Bits ungenutzt bleiben können. Die Zahl der Bytes pro Zeile gibt `colbytes` an. Das dient in erster Linie der Bequemlichkeit, man könnte den Wert von `colbytes` auch jedesmal ausrechnen, wenn er benötigt wird. Abbildung 4.3 zeigt den Aufbau schematisch. Mit der Funktion `az_getpixraw()` kann die Farbe eines Elements anhand seiner Koordinaten ermittelt werden.

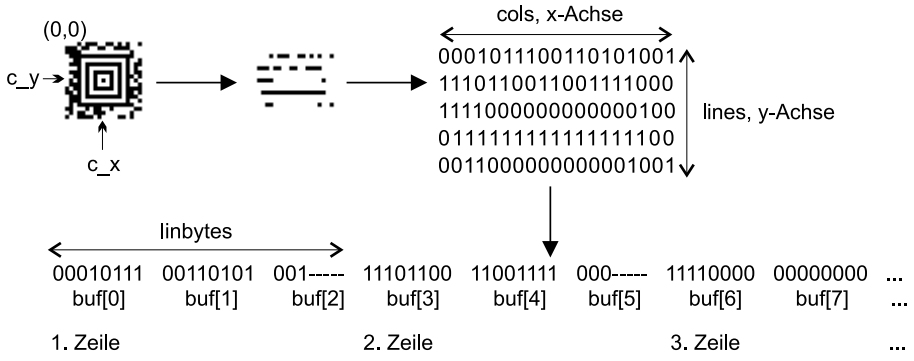

Abb. 4.3: *Bitmap-Struktur*

In `c_x` und `c_y` sind die Koordinaten des einzelnen Elements in der Mitte des Finder-Musters. Auf diese Weise wird ein zweites Koordinatensystem definiert, dessen Ursprung

der Symbolmittelpunkt ist. Dieses Koordinatensystem wird intern beim Zeichnen des Symbols verwendet. In Gestalt der Funktion `az_setelem()` steht es auch für die nachträgliche Bearbeitung außerhalb der Bibliothek zur Verfügung.

Anders als die übrigen Funktionen der `libaztec` gibt `az_drawsym()` keinen Zeiger auf statischen Speicher zurück, sondern reserviert die benötigten Bereiche abhängig von der Symbolgröße. Die Funktion `az_bmalloc()`, die dazu benutzt wird, steht auch Anwendungsprogrammen zur Verfügung. Als Parameter erwartet sie die Zeilen- und Spaltenzahl des zu erzeugenden Bitmaps. Der so reservierte Speicher wird mit `az_bmfree()` freigegeben.

Die Innereien von `az_drawsym()` folgen, wie auch der Rest der Bibliothek, fast sklavisch der Spezifikation. Nachdem Speicher für eine Bitmap-Struktur in der benötigten Größe reserviert wurde, werden zunächst die festen Elemente (Finder, Orientation Bits und Referenzgitter) gezeichnet. Danach kommt die Mode Message dran und zuletzt die Daten nebst Korrekturinformation, die, von hinten nach vorn gelesen, sprialförmig um den Finder gewickelt werden.

Dabei hilft die interne Funktion `ecmpnt()`, die für ein (in Symbolkoordinaten) gegebenes Element bestimmt, ob es bereits zu einem anderen Bestandteil des Symbols (z.B. zum Referenzgitter) gehört. Beginnend beim immer gleichen Startpunkt, $(-7, -8)$ in Symbolkoordinaten, werden die Dominopositionen durchlaufen. Elementpositionen, an denen sich bereits Teile des Referenzgitters befinden, werden dabei übersprungen. Wann die Laufrichtung geändert werden muß, gibt ebenfalls `ecmpnt()` an.

4.4.4 Weiterverarbeitung

Die Bitmaps, die die Aztec-Bibliothek liefert, lassen sich zu beliebigen Ausgabeformaten weiterverarbeiten. Im Modul `output` ist das beispielhaft für die Graphikformate GIF und PNG implementiert; ebenso wären aber auch zum Beispiel Postscript oder PDF möglich. Warum die Wahl von Pixelgraphikformaten sinnvoll ist, erläutert Abschnitt 4.5. Die GIF-Erzeugung erfolgt mit Hilfe der Bibliothek GD [Bou98] und für PNG wird die Referenzimplementierung Libpng [PDG98] benutzt. Aus diesem Grunde ist das Modul `output` nicht in die `libaztec` integriert; sie soll nicht von anderen Bibliotheken abhängig sein.

Die Funktionen `wrtgif()` und `wrtpng()` erwarten jeweils eine zum Schreiben geöffnete Datei, einen Zeiger auf eine `az_bitmap`-Struktur sowie die Elementgröße `x` in Pixeln. Obwohl sich die Formate GIF und PNG sehr ähneln, unterscheiden sich die benutzten Bibliotheken und damit die Vorgehensweisen erheblich.

Die Bibliothek GD erlaubt das Zeichnen von Rechtecken. Das macht die Sache einfach. Spalte für Spalte und Zeile für Zeile durchlaufen und für jedes schwarze Element ein Quadrat der Größe gezeichnet. Die Libpng ist deutlich sperriger zu handhaben. Sie erzeugt zwar klaglos PNG-Dateien, erwartet aber einen Speicherblock als Parameter, der bereits eine Bilddarstellung gemäß PNG-Spezifikation [Bou96b] enthält.

Die Funktion `wrtpng` erzeugt deshalb eine zweite `az_bitmap`-Struktur – das Format ist schon geeignet, nur die Größe stimmt noch nicht. Aus dem Symbolbitmap wird jeweils eine Zeile übertragen und dabei jedes Element ver-`x`-facht. Die so entstandene Zeile wird

in die (x-1) folgenden kopiert. Das ist nicht besonders schön, aber es funktioniert.

4.4.5 CGI-Programm

Als Beispielanwendung entstand das CGI-Programm `azcgi`, welches Aztec-Symbole auf dem Web-Server erzeugt. Für die CGI-Anbindung benutzt es die Bibliothek cgic [Bou96a]. Tabelle 4.5 zeigt die Aufrufparameter, die im URL übergeben werden können. Mit Ausnahme der zu kodierenden Nachricht `message` sind sie mit sinnvollen Werten vorbelegt und können weggelassen werden.

Name	Typ und Wertebereich	Default	Bedeutung
`message`	Oktettfolge	–	Nachricht
`maxlayers`	0-32	0	maximale Größe; 0 = default
`ecclevel`	0-100	0	Mindestumfang der Korrekturinformationen; 0 = default
`x`	1-30	3	Elementgröße (Bildpixel)
`format`	gif|png|txt	gif	Ausgabeformat
`allbin`	Flag	nein	gesamte Nachricht im Binärmodus kodieren
`truncate`	Flag	nein	zu lange Nachrichten kürzen

Flags sind gesetzt, wenn sie als Attribut im `QUERY_STRING` vorkommen und mit einem Wert belegt sind.

Tab. 4.5: *Aufrufparameter des Programms azcgi*

Das Programm `azcgi` greift auf die Funktionen der `libaztec` und des `output`-Moduls zurück. Es beschäftigt sich deshalb hauptsächlich mit der Fehlerbehandlung und dem Lesen der Eingabe. Um aus der übergebenen Nachricht ein Symbol zu erzeugen, müssen lediglich die drei Hauptfunktionen der Bibliothek `libaztec` in der richtigen Reihenfolge aufgerufen werden.

```
az_buffer *message;
az_bitstr *bitstr;
az_symdata *sdta;
az_bitmap *symbol;

bitstr = az_bitstream( *message, (allbin | truncate) );
sdta = az_mksymbol( bitstr, maxlayers, ecclevel, AZ_FLAG_NONE );
symbol = az_drawsym( sdta );
```

Dieser verkürzte Quelltextausschnitt, die Fehlerbehandlung wurde weggelassen, ist selbsterklärend. Die nicht deklarierten Variablen entsprechen den Parametern aus Tabelle 4.5. Ebenso einfach wie die Erzeugung ist die Ausgabe des Symbols.

```
FILE *cgiOut;
```

```
wrtgif( cgiOut, symbol, x ); /* oder wrtpng(...) */
```

Hier ist `cgiOut` die bereits zum Schreiben geöffnete Ausgabedatei, in diesem Fall ein Alias für die Standardausgabe, der von der Bibliothek cgic zur Verfügung gestellt wird. Der Zeiger `symbol` enthält die oben von `az_drawsym` gelieferte Adresse. Der dritte Parameter `x` gibt die Elementgröße in Bildpixeln an.

4.5 Wie kommt das Symbol zum Käufer?

Die im World Wide Web eingesetzten Techniken und Standards wurden mit dem Ziel der Plattformunabhängigkeit entwickelt. Daraus ergab sich, daß die Kontrolle über die Darstellung von Inhalten weitgehend auf der Client-Seite, also beim Browser, liegt. Informationsanbieter liefern beispielsweise in HTML-Dokumenten lediglich Strukturinformationen (Markup) und unverbindliche Hinweise für die Darstellung (Stylesheets); alles weitere wird dem Browser überlassen. Auf diese Weise ist es - korrekte Anwendung vorausgesetzt - möglich, ein und denselben Inhalt sogar auf so verschiedenen Geräten wie Bildschirm, Drucker, Sprachsynthesizer und Braille-Zeile in der jeweils geeigneten Form auszugeben.

Die Flexibilität und Plattformunabhängigkeit wird zum Nachteil, wenn der Informationsanbieter aus irgendwelchen Gründen eine weitergehende Kontrolle über die Darstellung benötigt.

Beim Verkauf von Tickets treten deshalb Schwierigkeiten auf. Während des Kaufvorganges soll dem Nutzer ein Datenpaket übermittelt werden, welches, auf seinem Drucker zu Papier gebracht, ein gültiges Ticket darstellt. Enthielte das Ticket neben Text nur Graphikelemente, die der Dekoration dienen, könnte man die Umsetzung in eine Druckgraphik problemlos dem Browser überlassen. Tickets sähen dann je nach Hardware, Software und Systemkonfiguration des Käufers unterschiedlich aus, wären aber sicher lesbar.

Zur Prüfung des Tickets werden jedoch Daten benötigt, die maschinenlesbar als Aztec-Symbol auf dem Ticket abgelegt sind. Hier erweist sich die mangelnde Kontrolle des Anbieters über den Druckvorgang beim Nutzer als problematisch.

Die Lesbarkeit des Symbols hängt von der Druckqualität ab, die zum einen vom Papier, zum anderen vom Drucker beeinflußt wird. Drucker und Papier bestimmen die minimale Elementgröße, mit der das Symbol sicher lesbar ist. Andererseits ist durch das Papierformat und das Sichtfeld des Scanners eine Maximalgröße gegeben. Sie ist jedoch weniger kritisch. Erfahrungen mit dem IT 4400 der Firma Welch Allyn zeigen, daß ungefähr alles, was auf ein A4-Blatt paßt, auch vom Gerät erfaßt werden kann. Lediglich die Aufteilung eines Symbols auf mehrere Seiten ist zu verhinern.

Bei unserer Anwendung liegen die Datenmengen weit unter der Kapazität des größtmöglichen Symbols, auf dessen Größe der Scanner ohnehin ausgelegt ist. Deshalb bleibt genügend Spielraum, eine unter allen Umständen sichere Elementgröße zu wählen.

Wie aber kann der Anbieter sicher sein, daß seine Tickets ausreichend groß gedruckt werden? Ist die Darstellung von HTML-Dokumenten noch begrenzt steuerbar, solange der Nutzer keine eigenen Style-Sheets einsetzt, so fehlt bei Graphiken und Bildern jede

Möglichkeit der Einflußnahme. Der Browser des Nutzer kann etwa ein GIF-Bild nach Belieben skalieren. Es gibt keinen Standard, der den Druck solcher Bilder regelt.

Ideal wäre das Gegenteil, nämlich ein Format, das dem Anbieter volle Kontrolle über den Druck läßt. Unter diesem Gesichtspunkt erscheinen Postscript und PDF gut geeignet. Im ersten Kapitel haben wir jedoch aus gutem Grund gefordert, daß der Käufer möglichst nichts als einen Web-Browser benötigen soll. Um Postscript zu drucken, benötigte er jedoch zusätzliche Software, wenn er nicht gerade einen Postscript-Drucker besitzt. Die ist zwar kostenlos erhältlich, aber Download und Installation kosten Zeit und Geld und unter manchen Betriebssystemen ist jede Softwareinstallation eine heikle Operation, die das System unbenutzbar machen kann.

Das alles möchte man nicht in Kauf nehmen, nur um eine Kinokarte zu erwerben. Selbst wenn man es hinnähme, bliebe ein weiteres Problem. Postscript als ASCII-Format (vgl. 2.7) macht aus kleinen Bildern große Dokumente. Angesichts der knappen Bandbreite vor allem auf den letzten Kilometern zwischen dem Käufer und seinem Provider ist das ein Problem. Datenkompression ist zwar möglich und wird sogar von HTTP unterstützt, erfordert aber ein weitere Software beim Kunden.

Pixelformate bieten eine kompaktere Repräsentation der Symbolgraphik und können – insbesondere GIF – von fast jedem Web-Browser ohne Hilfmittel angezeigt und gedruckt werden. Aber woher wissen wir, wie die Graphik aus dem Drucker kommt, wenn es für die Darstellung keine Standards gibt?

Theoretisch bietet das Format PNG [Bou96b] eine Lösung. Es wird von neueren Browsern (IE und Netscape Navigator jeweils ab Version 4.xx) unterstützt. PNG wurd als GIF-Ersatz speziell für die Benutzung im WWW entwickelt. Unter anderem gestattet es die Angabe von Pixeldimensionen. Die Pixelgröße wird, für X- und Y-Richtung getrennt, durch die Angabe der Pixelanzahl pro Meter spezifiziert.

Damit lassen sich die meisten Barcodes auf eine recht elegante Weise als darstellen. Ihnen ist gemeinsam, daß sie aus Elementen fester Größe zusammengesetzt sind. In einer PNG-Datei kann man jedes Element durch einen Pixel darstellen und als Pixelgröße die gewünschte Elementgröße angeben.

Leider ist dieser Teil der PNG-Spezifikation fakultativ und in den gängigen Browsern nicht implementiert. Berücksichtigt wird allenfalls das Seitenverhältnis, das sich aus der angegebenen Pixelgröße ergibt. Schade.

Dennoch sind Pixelgraphikformate in der Praxis geeignet, das Aztec-Symbol zum Käufer zu übermitteln. Experimente zeigen, daß sich nahezu alle Browser trotz fehlender Standards gleich verhalten. Netscape Navigator, MS Internet Explorer und Opera drucken GIF-Bilder mit 100 dpi. Amaya, der Referenzbrowser des W3C, bringt nur 75 Bildpixel auf einem Zoll unter; das Symbol wird dann beim Druck etwas größer. Wir werden deshalb Pixelformate nutzen und dabei annehmen, daß sie mit höchstens 100 dpi gedruckt werden. Ein Bildpixel hat dann eine Kantenlänge von wenigstens $0,25$mm. In dieser Schrittweite kann die Elementgröße variiert werden; welche Dimensionen sinnvoll sind, untersuchen wir im nächsten Abschnitt.

Im Interesse der Plattformunabhängigkeit ist es sinnvoll, ausschließlich das Aztec-Symbol als Bild zu übermitteln und in ein HTML-Dokument einzubinden, das die Ticketparameter und weitere Informationen in natürlicher Sprache enthält. Der Browser hat

dann die Freiheit, alles bis auf das Symbol nach seinen Bedürfnissen und den Wünschen seines Benutzers darzustellen.

Nur eins muß man sich bei der Gestaltung des HTML-Dokuments verkneifen. HTML gestattet in der Version 4.0 Größenangaben für Objekte, die mittels des OBJECT-Elements oder seiner älteren Geschwister APPLET und IMG in HTML-Dokumente eingebunden werden. In Abschnitt 13.7 der HTML-Spezifikation [RLJ98] heißt es dazu:

> »When specified, the width and height attributes tell user agents to override the natural image or object size in favor of these values.
>
> When the object is an image, it is scaled. User agents should do their best to scale an object or image to match the width and height specified by the author. Note that lengths expressed as percentages are based on the horizontal or vertical space currently available, not on the natural size of the image, object, or applet.«

Prozentangaben beziehen sich auf den zur Verfügung stehenden Platz. Der Browser wird aufgefordert, das Bild zum Beispiel auf 50% der Seitenbreite zu skalieren. Die aber kennt der Anbieter nicht, wenn er das Ticket und das zugehörige HTML-Dokument erzeugt. Prozentuale Größenangaben können deshalb zu ungewollten Ergebnissen führen und sind zu unterlassen, wie überhaupt jeder Layout-Versuch mit HTML zu unterlassen ist.

4.6 Experimente

Der ganze Wert unserer Eintrittskarten aus dem Netz liegt im geheimnisvollen Aztec-Symbol.[33] Ist dieses nicht dekodierbar, ist das Ticket wertlos, denn Papierstückchen, die nur echt *aussehen*, kann jeder herstellen.

Dem Käufer soll möglichst kein Verlust entstehen, wenn er sich einigermaßen vernünftig verhält. Deshalb muß sicher sein, daß die online gekaufte Eintrittskarte benutzbar bleibt, ohne vom Käufer besondere Sorgfalt zu fordern.

Dieser Abschnitt untersucht die Fehlertoleranz von Aztec-Symbolen. Es soll geklärt werden, welche Elementgröße für ausreichende Druckqualität erforderlich ist, und ob es Beschädigungen gibt, gegen die die Symbole besonders empfindlich sind.

Die Resultate wurden experimentell, aber nicht systematisch gewonnen. Da nur ein einziger Scanner, ein Welch Allyn IT 4400, zur Verfügung stand, gelten sie streng genommen nur für diese Hardware. Jede Statistik wäre von vornherein wertlos, weil sie diesen Einfluß nicht näher bestimmen könnte. Ebenso lassen sich Einflüsse des Benutzers bei einem Handscanner kaum ausschließen. Wer falsch zielt, erhält schlechtere Ergebnisse, aber was ist »falsch« und woran erkennt man, ob man etwas falsch gemacht hat?

Der Autor hat deshalb keine Erstleseraten bestimmt, sondern beschränkt seine Beobachtungen auf die drei Kategorien *gut lesbar*, *schlecht lesbar* und *nicht lesbar*. Gut lesbare Symbole werden in der Regel bei einem der ersten Leseversuche erkannt. Schlecht

[33] Das ist ein ernstes Problem. Man muß den Veranstalter bzw. seine Mitarbeiter darauf dressieren, die Gültigkeit ausschließlich vom Computer beurteilen zu lassen. Menschen, die einem Computer blind vertrauen, sind aber selbst ein Problem. Vgl. [Neu95].

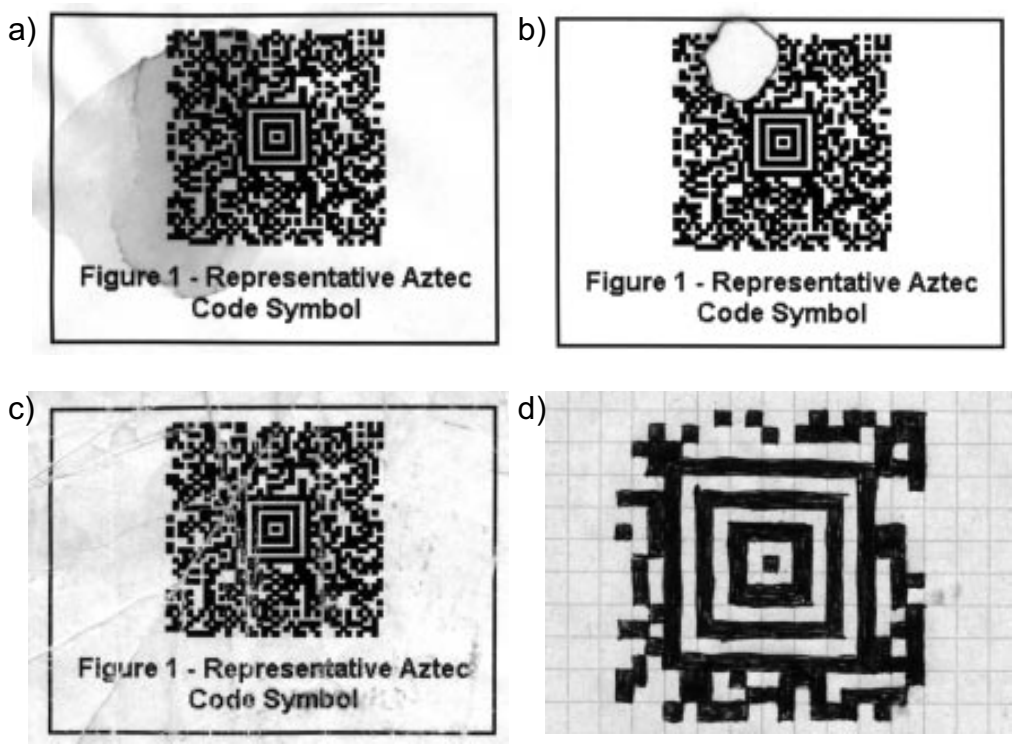

Abb. 4.4: *Beschädigte Symbole I. a) Teefleck; b) Brandloch; c) Mehrwöchiger Transport in der Hosentasche; d) Unbekannter Meister: "ABCDEFGH", Kugelschreiber auf Papier, um 1998.*

lesbare dagegen werden in der Regel nicht erkannt, mit etwas Glück und Geduld aber manchmal doch. Nicht lesbare Symbole sind die, die übrigbleiben.

Die Leistungsfähigkeit der Fehlerkorrektur ist bekannt; für jedes gegebene Symbol kann berechnet werden, wieviele Fehler korrigierbar sind. Mit einem Zeichenprogramm den Datenbereich eines Symbols zu verändern erschien daher wenig sinnvoll. Statt dessen wurden Beschädigungen simuliert, die im Alltag auftreten können, um herauszufinden, was man alles *nicht* tun darf, wenn ein Symbol lesbar bleiben soll.

Für die meisten Versuche wurde ein Beispielsymbol aus der Spezifikation [WAl97] verwendet. Es enthält 109 ASCII-Zeichen. Die Nachrichtenkodierung macht daraus 75 Acht-Bit-Zeichen, die um 81 Zeichen für die Fehlerkorrektur erweitert wurden.

Bild 4.4 zeigt Symbole, die trotz Beschädigung beziehungsweise eigenwilliger Ausgabegeräte gut lesbar sind.

Tee und Kaffee (4.4a) verringern den Kontrast zwischen den schwarzen Elementen und dem vormals weißen Hintergrund nur wenig. Sie stellen keine ernste Gefahr dar, es sei denn, der Druck erfolgt mit einer nicht wasserfesten Tinte.

Auch das Brandloch in Symbol 4.4b ist in dieser Größe und Position unproblematisch. Dafür sorgt die Fehlerkorrektur, die bei diesem Symbol fast schon übertrieben erscheint.

Das Symbol 4.4c trug der Autor wochenlang achtlos in der Hosentasche mit sich. Die Auswirkungen sind nicht zu übersehen, doch auch dieses Symbol konnte noch ohne Schwierigkeiten gelesen werden.

Bild 4.4d zeigt ein Symbol, das der Autor ohne Lineal selbst gezeichnet hat. Die Elementgröße beträgt ungefähr $2,5$mm. Die Ausgabequalität spielt bei dieser Größe offenbar nur noch eine geringe Rolle. Sie sollte daher mit allen denkbaren Druckern ausreichend gute Ergebnisse liefern. Allerdings wäre ein Symbol mit 256 Bytes Daten und genauso viel Korrekturinformation dann ungefähr 20cm groß und damit sehr unhandlich. Wie wir gleich sehen werden, muß es so groß denn doch nicht sein.

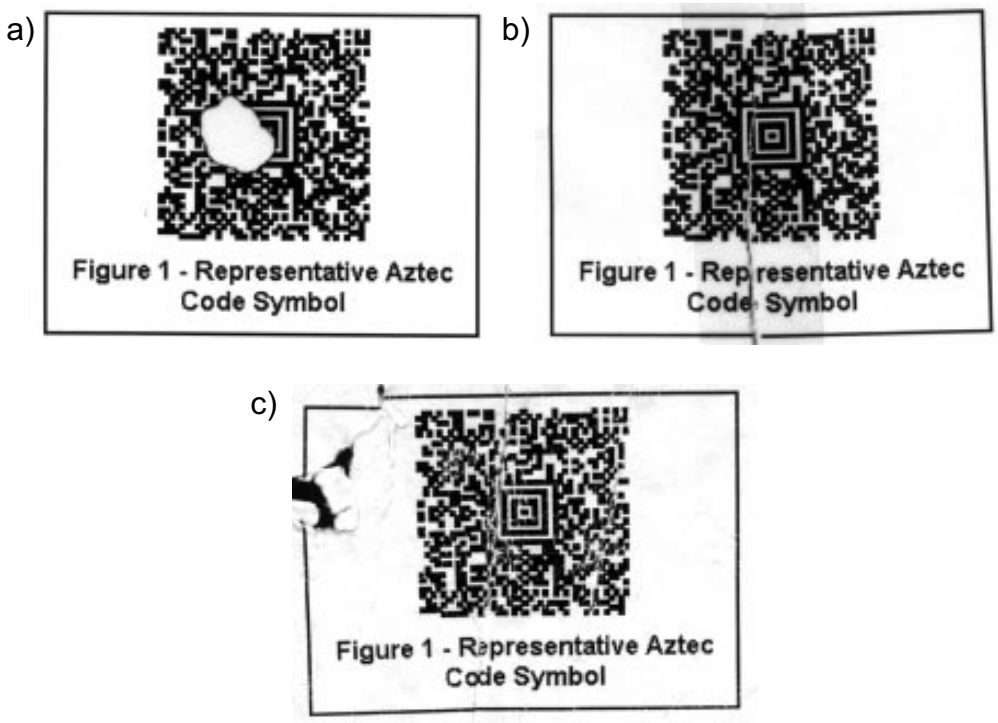

Abb. 4.5: *Beschädigte Symbole II. a) Brandloch; b) Mit Klebeband reparierter Riß; c) Handwäsche.*

Zum Vergleich zeigt Abbildung 4.5 Symbole, die wegen ihrer Beschädigung nicht mehr gelesen werden konnten. Das Loch in 4.5a ist nicht größer als das im vorhergehenden Bild, hat aber einen kritischen Teil des Symbols beschädigt. Das Finder-Muster, mit dessen Suche das Dekodieren beginnt, ist zur Hälfte ausgelöscht. Da es vom Scanner anhand seiner topologischen Eigenschaften gefunden wird [WAl97], kann er hier nichts ausrichten – von keinem einzigen der Quadrate ist noch ein geschlossener Linienzug erhalten. Eine noch viel kleinere Beschädigung mit gravierenden Auswirkungen wäre die Auslöschung der Orientation Bits an den Ecken des Finders. Sie wird in der Praxis aber kaum so isoliert vorkommen.

In Abbildung 4.5b ist ein Symbol zu sehen, das sehr schwer lesbar ist, obgleich es nur leicht beschädigt aussieht. Damit es überhaupt dekodiert werden kann, muß (beim Original) der Scanner schräg gehalten werden. Der Grund ist ein Stück Klebeband, mit dem ein Riß repariert wurde. Der Scanner beleuchtet das Symbol selbst, was zu störenden Reflexionen führt. Sie treffen genau die CCD-Kamera des Scanners, wenn das Symbol parallel zu ihr gehalten wird.

Das dritte Symbol in Abbildung 4.5 wurde – nur kurz! – im Wasser mechanisch belastet. Das Ergebnis unterscheidet sich optisch nicht sehr von 4.4c, kann im Gegensatz dazu aber nicht mehr dekodiert werden. Vermutlich liegt das daran, daß die Orientation Bits an zwei Ecken des Finders zerstört sind. Aber auch der Finder selbst sowie die Korrektur- und Nachrichtenbits sind erheblich beschädigt.

Für die Versuche wurden alle Symbole mit einem hochwertigen Laserdrucker zu Papier gebracht. Die Elementgröße lag bei ungefähr 0, 7mm. Welchen Einfluß hat die Druckqualität auf die Lesbarkeit?

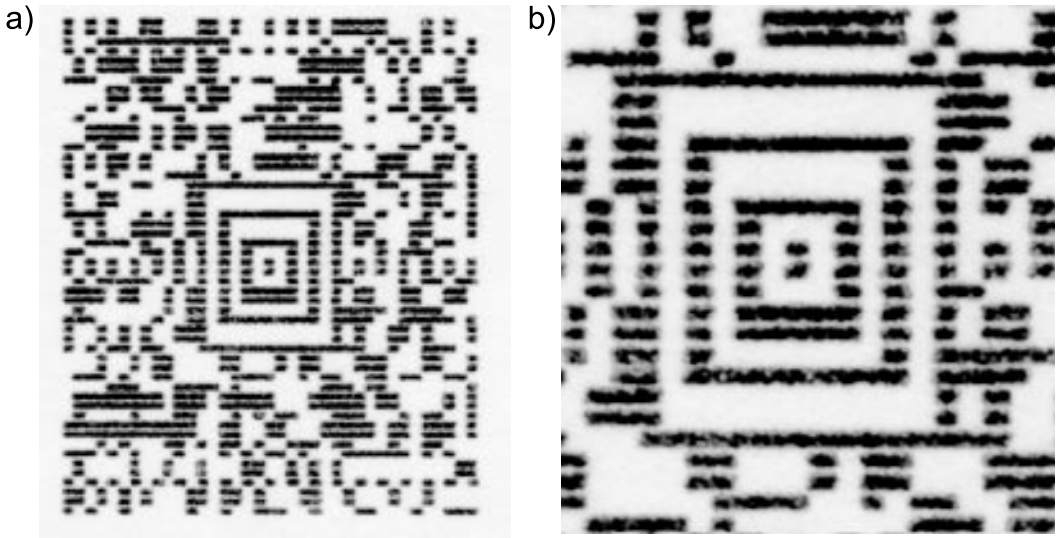

Abb. 4.6: *Aztec-Symbol aus dem Nadeldrucker (vergrößert). Der Ausschnitt rechts zeigt Verzerrungen, die vermutlich bei der Aufbereitung der Postscript-Datei für den Druck entstanden.*

Um das herauszufinden, wurden Aztec-Symbole mit einem 24-Nadeldrucker gedruckt, dessen Farbband bereits deutlich gealtert war. Bereits in Abschnitt 4.2 wurde erwähnt, daß mit einem solchen Drucker Strukturen ab ungefähr 0, 4mm in ausreichender Qualität erzeugt werden können. Testsymbole wurden deshalb mit Elementgrößen von 2, 3 und 4 Pixeln erzeugt, das entspricht 0,5 bis 1,0 Millimetern. Die Fehlerkorrektur wurde für jede Größe zwischen dem Minimum und 100% der Nachrichtenlänge variiert.

Das Ergebnis war erstaunlich, zumal die Software – Netscape kombiniert mit Ghostscript – auch noch für deutliche Verzerrungen sorgte (Bild 4.6b). Schon mit der kleinesten Elementgröße von 0, 5mm und minimaler Fehlerkorrektur ließen sich die (unbeschädigten)

Symbole gut lesen.

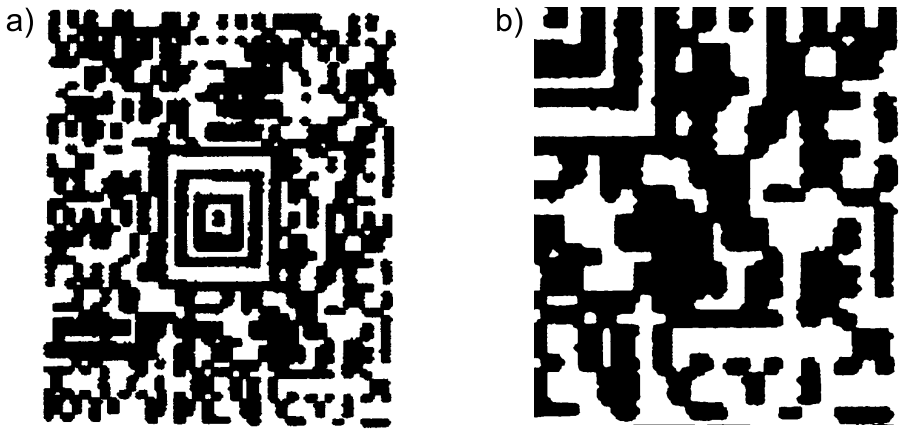

Abb. 4.7: *Faxkopie des Symbols aus Abb. 4.6 (vergrößert)*

Schwierigkeiten traten erst auf, nachdem die gedruckten Symbole auch noch mit einem Faxgerät kopiert worden waren. Dabei entstanden deutlich verwaschene Kopien, wodurch die kleinsten Symbole unabhängig vom Anteil der Fehlerkorrektur unlesbar wurden. Bei $0,75$ und $1,0$mm Elementgröße waren hingegen auch die Kopien noch leicht zu lesen.

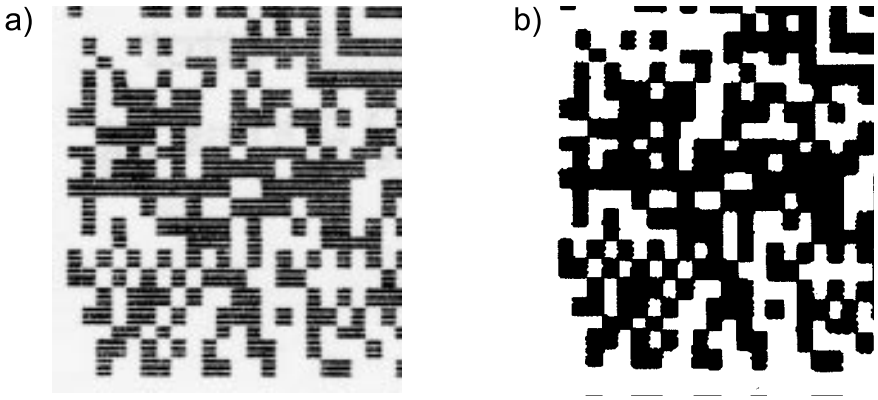

Abb. 4.8: *Zum Vergleich ein Symbol mit doppelter Elementgröße (vergrößerte Ausschnitte); a) Original aus dem Nadeldrucker, b) Faxkopie.*

Mit einer Elementgröße von 4 Pixeln und einem Korrekturanteil von 100% der Nachrichtenlänge werden wir für alle vorstellbaren Fälle gut gerüstet sein. Vor der Waschmaschine ist das digitale Billett genauso wenig geschützt wie das herkömmliche, aber den Weg von der Wohnung zum Veranstaltungsort wird es überstehen, auch wenn man

es achtlos in die Hosentasche stopft. Mehr muß es auch gar nicht verkraften, denn die Ticketdaten können gespeichert und bei Bedarf erneut gedruckt werden.

5 Implementierung

Pilotprojekt Früher hieß dieses schnittige Projekt »Versuchskarnickel«.
(Eckhard Henscheid: Dummdeutsch)

Der Aztec-Generator aus dem 4. Kapitel vollendet das Fundament. Wir wissen, daß wir Eintrittskarten im Internet verkaufen können, und wir wissen auch ungefähr, was wir dafür tun müssen. Der nächste und in dieser Arbeit letzte Schritt ist die Implementierung eines Prototypen. Er ist nicht Neben- sondern Hauptprodukt; Prototyping ist ein wichtiger Lernprozeß [Gan95]. Beim Prototypen können wir einerseits Kompromisse machen, wo es nur darum geht, die Funktionsfähigkeit zu zeigen. Andererseits läßt er Raum für Variationen und Experimente. Bei der Entwicklung richtete sich das Augenmerk deshalb auch mehr auf die Grundstruktur und die Erweiterbarkeit als auf die lästigen Details, die auf dem Weg vom Prototypen zum Produkt ohnehin noch $(n+1)$-mal geändert werden.

Was brauchen wir? Schon in 1.3 haben wir festgestellt, daß das wir das Ticket als Nachricht betrachten können. Wir benötigen ein System, das die Ticketparameter als Nachricht darstellt und diese manipulatonssicher über einen nicht vertrauenswürdigen Kanal übermittelt. Daraus ergibt sich fast zwangsläufig ein Schichtenmodell, wie es aus der Welt der Rechnernetze bekannt ist.

Die untere Schicht bildet ein beliebiges Übertragungsmedium. In unserem Fall sind das Aztec-Symbole, aber wir fordern nichts weiter, als daß es eine genügend lange Oktettfolge transportieren kann und dabei Übertragungsfehler vermeidet. So kann der Aztec-Kode leicht zum Beispiel durch Chipkarten ersetzt werden, ohne daß der Rest des Systems wesentlich verändert werden muß.

Die zweite Schicht sorgt für die Sicherheit. Sie erhält auf der Senderseite eine Nachricht als Eingabe und gibt eine veränderte an die Übertragungsschicht weiter. Beim Empfänger erhält sie umgekehrt Nachrichten von der Übertragungsschicht und muß entscheiden, ob sie authentisch sind. Nach oben weitergegeben werden nur diejenigen, für die diese Prüfung positiv ausfällt. Ihre kryptographischen Schlüssel soll die Sicherheitsschicht selbst verwalten.

Die obere Schicht schließlich hat eigentlich nichts weiter zu tun als Parametersätze auf Nachrichten abzubilden und umgekehrt. Wir ordnen ihr aber noch eine weitere Aufgabe zu, nämlich die Prüfung von Parametersätzen auf Gültigkeit, wie sie in Kapitel 2 behandelt wird.

Drei Programme nutzen dieses Gerüst. Eines ist für den Verkauf zuständig. Damit der Kunde auch sieht, wofür er sein Geld ausgegeben hat, soll er neben der gesicherten Nachricht in Form eines Aztec-Symbols auch noch eine natürlichsprachige Darstellung des Billetts bekommen. Sie wird direkt aus dem Parametersatz erzeugt, kann aber auch Informationen enthalten, die für die Kontrolle unbedeutend und deshalb nicht im Aztec-

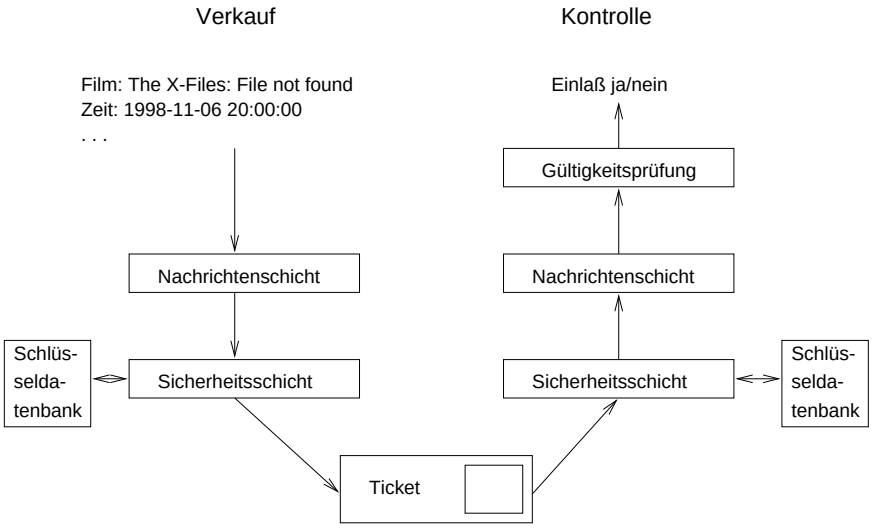

Abb. 5.1: *Schichtenmodell*

Symbol kodiert sind. Das Verkaufsprogramm erzeugt diese Darstellung als HTML-Text und bettet ein Aztec-Symbol mit der gesicherten Nachricht ein.

Das zweite Programm übernimmt die Ticketkontrolle. Es muß Symbole lesen, daraus einen Parametersatz gewinnen und diesen auf Gültigkeit prüfen. Dazu führt es eine Datenbank, in der die bereits kontrollierten Eintrittskarten vermerkt werden.

Ein drittes Programm dient nur als Hilfsmittel. Verkaufssystem und Kontrollstation benötigen kryptographische Schlüssel, die transportiert und verwaltet werden müssen.

Eigentlich ist eine weitere Komponente nötig, die die Veranstaltungen und ihre Parameter auf der Verkaufs- wie auf der Kontrollseite verwaltet. Sie fehlt im Prototypen; statt dessen werden alle Parameter beim Verkauf als CGI-Eingabe und bei der Kontrolle auf der Kommandozeile übergeben. Das ist unbequem und unsicher, genügt aber zum Ausprobieren.

5.1 Werkzeuge

Bevor wir zur Tat schreiten, wenige Worte zu den verwendeten Werkzeugen. Wie schon beim Aztec-Symbolgenerator erfolgt die CGI-Anbindung des Verkaufsprogramms mittels der Bibliothek cgic [Bou96a]. Weitere Hilfmittel sind SSLeay für die kryptographischen Funktionen und DBM zur Datenspeicherung.

5.1.1 DBM

(N)DBM ist eine primitive Datenbank, die mit jedem Unix-System mitgeiefert wird. Sie kann lediglich Schlüssel-Wert-Paare in einer einfachen Tabelle ablegen. Schlüssel und Wert können beliebige Oktettfolgen sein, jedoch darf ihre Länge zusammen 1024 Bytes

nicht überschreiten.

Die abgelegten Daten können über ihren Schlüssel jederzeit gelesen werden. Dazu sind nur zwei Zugriffe auf den Massenspeicher erforderlich. Das Speicherformat ist architekturabhängig, also nicht zwischen verschiedenen Plattformen übertragbar. Das stört hier nicht, denn DBM soll nur lokal benutzt werden.

Ein anderes Manko fällt stärker ins Gewicht. DBM ist nicht in der Lage, Datensätze zu sperren. Konkurrierende Zugriffe durch mehrere Prozesse können daher zu unsinnigen Resultaten führen, sobald einer davon eine Schreiboperation ist. Dieses Problem kann umgangen werden, indem man bei solchen Operationen die gesamte DBM-Datenbank zumindest für kooperative Prozesse sperrt. Das ist keine richtig gute Idee, weil es viel zu weit greift, aber besser geht es nicht.

Die Vorteile von DBM liegen darin, daß es einfach da und leicht zu benutzen ist. Wir werden es deshalb zum einen zur Schlüsselspeicherung benutzen, zum anderen in der Kontrollstation zum Ablegen der eindeutigen Billettnummern.

5.1.2 SSLeay

Kapitel 3 fordert den Einsatz bereits vorhandener Implementierungen der kryptographischen Algorithmen. Sie sollen im Quelltext erhältlich und ausgiebig getestet sein.

Die Suche nach einer geeigneten Kryptographiebibliothek war unerwartet schwierig. Amerikanische Software ist aufgrund von Exportbeschränkungen nicht ohne weiteres verfügbar. Peter Gutmanns Cryptlib[34] ist, hmm, idiotensicher, worunter die Flexibilität leidet. Das ist kein grundsätzliches Problem, aber die Schlüsselverwaltung dieser Bibliothek ist nur in Verbindung mit ausgewachsenen Datenbanksystemen brauchbar. Die sind für Kinokarten eine Nummer zu groß. SECUDE[35] machte auf den ersten Blick einen guten Eindruck, von der Benutzung wurde jedoch aufgrund schlechter Erfahrungen abgeraten [Hue98].

Die Wahl fiel schließlich auf SSLeay [HuYo98, Reif96] von Eric Young und Tim Hudson. SSLeay ist eine freie Implementierung des SSL-Protokolls. Die dafür benötigten kryptographischen Algorithmen sind in einer Bibliothek zusammengefaßt, die auch für andere Zwecke genutzt werden kann. Sie enthält alles, was wir für den Prototypen brauchen: symmetrische Blockchiffren (DES, IDEA, Blowfish), HMAC mit verschiedenen Hashfunktionen (MD5, SHA-1) sowie asymmetrische Signaturverfahren (RSA, DSA).

Die Dokumentation ist spärlich, doch dank sauberer, gut lesbarer Quellen ist das kein Problem. Die Programmierschnittstellen sind durchdacht und ähnlich einfach zu nutzen wie die der Cryptlib, lassen dem Programmierer aber mehr Freiraum. Insbesondere kann man die Schlüsselverwaltung selbst in die Hand nehmen. Unterstützung bieten dabei Funktionen, die Schlüssel für asymmetrische Verfahren plattformunabhängig gemäß ASN.1 darstellen.

[34] `http://www.cs.auckland.ac.nz/~pgut001/cryptlib/index.html`
[35] `http://www.darmstadt.gmd.de/secude/`

5.2 Sicherheitsschicht

Die Sicherheitsschicht schützt Nachrichten vor Manipulation. Dazu verwendet sie die drei Verfahren, die im 3. Kapitel diskutiert werden. Die Schlüsselverwaltung wird von der Sicherheitsschicht selbst übernommen. Anwendungsprogramme bekommen lediglich Schlüsselkennungen zu Gesicht. Das macht es leicht, sämtliche kryptographsichen Funktionen auf einen eigenen, besonders gesicherten Rechner auszulagern und die Funktionen der Sicherheitsschicht übers Netz aufzurufen.

Unabhängig vom kryptographischen Verfahren bietet die Sicherheitsschicht zwei Basisfunktionen. Auf der Senderseite erhält sie eine Nachricht in Form einer Bytefolge. Die Nachricht wird dabei verändert. Die Empfängerseite prüft die Authentizität, sofern das gewählte Verfahren dies zuläßt.[36] Dann werden die Veränderungen rückgängig gemacht. Ergebnis ist die gesendete Nachricht; werden Blockchiffren zur Sicherung genutzt, möglicherweise auch eine andere, unsinnige Nachricht.

Die entsprechenden Funktionen heißen `tck_protectmsg()` für den Sender und `tck_unprotectmsg()` für den Empfänger. Neben der zu sendenden beziehungsweise der empfangenen Nachricht erhalten sie als Parameter lediglich eine Schlüsselkennung.

Intern arbeitet die Sicherheitsschicht mit austauschbaren Protokollmodulen. Zusammen mit jedem Schlüssel ist in der Datenbank die Information abgelegt, für welches Protokollmodul er bestimmt ist. Jede Schlüsselkennung legt damit implizit das benutzte Verfahren fest.

Kryptographieschlüssel werden zusammen mit Metainformationen in einer DBM-Datenbank abgelegt. Neben den beiden Grundfunktionen bietet die Sicherheitsschicht Funktionen zur Erzeugung sowie zum Ex- und Import von Schlüsseln.

Funktion	Aufgabe
`tck_protectmsg()`	sichert Nachrichten gegen Manipulation
`tck_unprotectmsg()`	stellt gesendete Nachricht wieder her und prüft auf Veränderung
`tck_genkey()`	erzeugt Schlüssel und legt ihn in der Datenbank ab
`tck_exportkey()`	exportiert einen Schlüssel aus der Datenbank
`tck_importkey()`	importiert einen Schlüssel in die Datenbank
`tck_delkey()`	löscht einen Schlüssel aus der Datenbank
`tck_listkeys()`	listet Datenbankinhalt auf
`tck_slinit()`	öffnet die Datenbank und sonstige Initialisierung
`tck_slclose()`	schließt die Datenbank

Tab. 5.1: *Funktionen der Sicherheitsschicht*

Die Schlüsseldatenbank muß geöffnet werden, bevor die Sicherheitsschicht darauf zu-

[36] Bei symmetrischer Verschlüsselung können Manipulationen erst in den darüberliegenden Ebenen erkannt werden. Eine Ausnahme bilden Nachrichten, die kürzer sind als die Blocklänge des benutzen Verfahrens. Sie werden auf definierte Weise aufgefüllt (»Padding«), so daß Veränderungen mit einer gewissen Wahrscheinlichkeit schon beim Entschlüsseln durch falsches Padding auffallen.

greifen kann. Darüber hinaus sind weitere Initialisierungsschritte notwendig, etwa die Festlegung eines Startwertes für den Zufallszahlengenerator, mit dessen Hilfe Schlüssel erzeugt werden. Bevor irgendeine andere Funktion der Sicherheitsschicht genutzt werden kann, ist deshalb `tck_slinit()` aufzurufen, wo all dies erledigt wird. Die Datenbank bleibt geöffnet, bis das Gegenstück `tck_slclose()` aufgerufen wird. Zu jedem Zeitpunkt kann nur eine Datenbank geöffnet sein.

Die Funktion `tck_slinit` erwartet als Parameter den Pfad zur Schlüsseldatenbank und Flags. Er wird von DBM um die Endungen `.dir` und `.pag` ergänzt; außerdem dient er als Basisname für Sperrdateien mit der Endung `.LCK`.

Sind keine Flags angegeben (`TCK_FLG_NONE`), wird die Schlüsseldatenbank nur zum Lesen geöffnet. Das genügt zum Sichern und Prüfen von Nachrichten, da dabei nur vorhandene Schlüssel eingesetzt werden. Will man dagegen Funktionen zur Schlüsselverwaltung aufrufen, muß die Datenbank auch zum Schreiben geöffnet sein. Das wird durch das Flag `TCK_FLG_RDWR` erreicht.

5.2.1 Protokollmodule

Um die Nutzung unterschiedlicher Verfahren und Algorithmen zu ermöglichen und das System einfach erweiterbar zu machen, wurden sämtliche kryptographischen Funktionen in Protokollmodule verlegt. Entsprechend den drei Verfahren aus 3.1 gibt es drei Protokolltypen für symmetrische Verschlüsselung (`TCK_PTYPE_SYM`), symmetrische MACs (`TCK_PTYPE_MAC`) und für asymmetrische Signaturverfahren (`TCK_PTYPE_ASYM`).

Ein Protokollmodul implementiert einen bestimmten Algorithmus, beispielsweise IDEA im ECB-Modus oder RSA-Signaturen. Wie es das tut, ist allein Sache des Protokollmoduls. Lediglich die Schnittstelle, über die das Modul von den anderen Funktionen der Sicherheitsschicht aufgerufen wird, ist durch den Modultyp festgelegt.

Funktion	Parameter	Aufgabe
`genkey()`	Passwort	leitet einen Schlüssel aus dem Passwort ab oder erzeugt einen zufälligen, falls kein Passwort angegeben ist
`encrypt()`	Klartext, Schlüssel	verschlüsselt den Klartext mit dem übergebenen Schlüssel
`decrypt()`	Kryptogramm, Schlüssel	entschlüsselt das Kryptogamm mit dem übergebenen Schlüssel

Tab. 5.2: *Modulschnittstelle für symmetrische Verschlüsselung*

Zu jedem Modultyp ist eine Programmschnittstelle definiert. Sie ist unabhängig davon, welchen konkreten Algorithmus ein Modul implementiert. Wie das Modul seinen Algorithmus implementiert, ist ihm überlassen. Die Funktionen der Sicherheitsschicht müssen so nur die Schnittstellen der drei Modultypen kennen. So können später Protokollmodule hinzugefügt werden, ohne daß Änderungen am Quelltext erforderlich sind.

Jedes Protokollmodul verfügt über eine Funktion zur Schlüsselerzeugung. Für symmetrische Verfahren kann der Schlüssel auch von einem Passwort abgeleitet werden, bei Signaturverfahren ist das hingegen nicht sinnvoll.

Die erzeugten Schlüssel werden in einem linearen Puffer zurückgegeben. Das Format, in dem sie darin abgelegt sind, ist dem Modul überlassen, muß aber so gestaltet sein, daß der Schlüssel auch auf anderen Plattformen gelesen werden kann. Wichtig ist das in erster Linie bei den asymmetrischen Verfahren. Deren Schlüssel bestehen aus mehreren Komponenten, bei denen es sich um Zahlen handelt, so daß unterschiedliche Darstellungen möglich sind. Schlüssel für symmetrische Verfahren sind hingegen Bitfolgen, die nicht weiter interpretiert werden, so daß sich da kaum Schwierigkeiten ergeben.

Funktion	Parameter	Aufgabe
`genkey()`	Passwort	leitet einen Schlüssel aus dem Passwort ab oder erzeugt einen zufälligen, falls kein Passwort angegeben ist
`auth()`	Nachricht, Schlüssel	fügt MAC an die Nachricht an
`verify()`	gesicherte Nachricht, Schlüssel	prüft und entfernt MAC

Tab. 5.3: *Modulschnittstelle für MACs*

Signaturmodule liefern bei der Schlüsselerzeugung das komplette Schlüselpaar oder eine Repräsentation des geheimen Schlüssels, aus der der öffentliche leicht berechnet werden kann. Auch hier sind die Einzelheiten dem Modul überlassen. Es muß lediglich eine Funktion `sec2pub()` zur Verfügung stellen, die aus einem geheimen den öffentlichen Schlüssel gewinnt.

Ebenso ist jedes Modul selbst für die Darstellung der gesicherten Nachrichten zuständig. Auch hier wird aber eine plattformunabhängige Repräsentation gefordert, die von demselben Modul auf beliebigen Systemen verarbeitet werden kann.

Da wir SSLeay zur Implementierung der Protokollmodule benutzen, lösen sich die Darstellungsprobleme fast von selbst, denn SSLeay ist für den Einsatz in heterogenen Netzen bestimmt und bringt deshalb alle nötigen Fähigkeiten mit. Symmetrische Verfahren bereiten ohnehin wenig Probleme und für die asymmetrischen sind Funktionen zur ASN.1-Darstellung vorhanden.

Jedes Protokollmodul exportiert eine Datenstruktur vom Typ `tck_proto_t`. Sie enthält einen Satz Zeiger auf die Funktionen des Moduls sowie einen Zahlenkode für den Modultyp. Diese Strukturen werden für alle benutzten Module in ein Feld (`pmod` in `msglayer.c`) eingetragen. Jedes Protokollmodul (und damit jedes Kryptoverfahren) ist also eindeutig durch seinen Feldindex gekennzeichnet.

Die übrigen Teile der Sicherheitsschicht arbeiten nur noch mit diesen Indizes, über die sie die Funktionen eines jeden Moduls erreichen. Schlüssel werden dabei als Bytefolgen betrachtet, die den Modulen lediglich übergeben werden, deren Inhalt aber sonst bedeutungslos ist.

Für jeden Protokolltyp ist ein Beispiel implementiert, das seinerseits auf SSLeay zu-

Funktion	Parameter	Aufgabe
`genkey()`	—	erzeugt ein Schlüsselpaar
`sign()`	Nachricht, geheimer Schlüssel	signiert die Nachricht
`verify()`	signierte Nachricht, öffentlicher oder geheimer Schlüssel	prüft und entfernt die Signatur
`sec2pub()`	geheimer Schlüssel bzw. Schlüsselpaar	extrahiert den öffentlichen Schlüssel

Tab. 5.4: *Modulschnittstelle für Signaturen*

rückgreift, um kryptographische Operationen auszuführen. Zur symmetrischen Verschlüsselung wird IDEA im ECB-Modus verwendet, als MAC das HMAC-Verfahren mit MD5 als Hashfunktion und für digitale Signaturen RSA.

5.2.2 Schlüsselverwaltung

Die Schlüsselverwaltung des Prototypen weicht deutlich vom Entwurf aus 3.1.5 ab. Dort wird empfohlen, bei der Systeminstallation Schlüssel mit unbegrenzter Lebensdauer zu erzeugen. Der Prototyp implementiert statt dessen (andeutungsweise) Verfahren für die regelmässige Änderung der verwendeten Schlüssel. Sie werden auf einem Rechner – dafür bietet sich derjenige an, auf dem das Verkaufssystem läuft – erzeugt und können von dort signiert und verschlüsselt an andere Rechner übermittelt werden. Damit sind sie, wichtig bei symmetrischen Verfahren, vor unbefugter Kenntnisnahme geschützt und der Empfänger kann sich über die Herkunft vergewissern.

Der Grund für die Abweichung liegt eben im Prototypcharacter des Systems. In der Testphase müssen häufiger Schlüssel erzeugt werden, zumal der Prototyp verschiedene kryptographische Verfahren anbietet. Hat man sich später für ein Verfahren entschieden, kann man die Verwaltung immer noch vereinfachen oder die vorhandenen Methoden ungenutzt lassen.

Verwaltungsfunktionen dienen zur Bearbeitung der Schlüsseldatenbank. Dort sind alle Schlüssel unter einer eindeutigen Kennung abgelegt. Die Grundlage bilden die internen Funktionen `newid()` zum Erzeugen von Schlüsselkennungen, `storekey()` zum Ablegen eines Schlüssels und `fetchkey()` zum Lesen von Schlüsseln aus der Datenbank.

Schlüsselkennungen sind acht Bytes lang und können beliebige Werte annehmen. Erzeugt werden sie von der Funktion `newid()` aus der Prozeßnummer, der aktuellen Zeit und einem Zähler, der für jeden Prozeß neu gestartet wird. Das garantiert eindeutige Kennungen, solange alle Schlüssel auf ein und demselben System erzeugt werden. Zur Schlüsselerzeugung benutzen wir das Verkaufssystem.[37] Signaturschlüssel können ohenehin nur dort erzeugt werden und bei den symmetrischen Verfahren ist das keine große Einschränkung.

[37] Diese Festlegung wird von der Software nicht überwacht oder gar durchgesetzt.

Die Funktion `storekey()`, die nur innerhalb der Sicherheitsschicht nutzbar ist kann, legt einen Schlüssel unter seiner Kennung in der DBM-Datenbank ab. Zuammen mit dem Schlüssel werden vier Parameter gespeichert. Das sind die Protkollnummer (der Feldindex aus dem vorigen Abschnitt), der Protokolltyp (Verschlüsselung, MAC oder Signatur) und der Schlüsseltyp (geheim oder öffentlich), der nur bei asymmetrischen Verfahren eine Rolle spielt, aber überall angegeben wird. Der vierte Parameter ist eine Zeitangabe in `time_t`-Darstellung. Sie legt fest, bis wann der Schlüssel gültig ist. Ist das Verfallsdatum überschritten, kann dieser Schlüssel nicht mehr verwendet werden. Derart abgelegte Schlüssel werden mit `fetchkey()` wieder aus der Datenbank geholt.

Den Anwendungsprogrammen stehen fünf darauf aufbauende Funktionen zur Verfügung. Mit `tck_genkey()` werden Schlüssel erzeugt. Die Funktion erwartet eine Protokollnummer und ein Verfallsdatum. Über die Protokollnummer wird das Modul identifiziert, mit dem der Schlüssel später benutzt werden soll. Dessen `genkey()`-Funktion erzeugt einen Schlüssel und gibt ihn als Puffer im Speicher zurück. Sodann wird er unter einer neu erzeugten Kennung in der Datenbank abgelegt. Zurückgegeben wird nur die Kennung; der Aufrufer von `tck_genkey()` bekommt den erzeugten Schlüssel also nicht zu Gesicht.

Dazu muß er eine andere Funktion bemühen, nämlich `tck_exportkey()`. Sie stellt einen Schlüssel aus der Datenbank zur Übertragung auf einen anderen Rechner bereit. Dabei kann er mit einem Passwort verschlüsselt und, sofern ein entsprechender privater Schlüssel existiert, auch signiert werden. Dazu werden dieselben Protokollmodule eingesetzt wie zur Nachrichtensicherung. Das ist auch der (einzige) Grund, weshalb Protokollmodule für symmetrische Verfahren Schlüssel von Passworten ableiten können. Das Verschlüsselungsverfahren für den Schlüsselexport ist nicht frei wählbar, sondern wird beim Übersetzen der Quelltexte durch die Konstante `EXPT_CRYPT_PROTO` festgelegt.

Saubere Schlüsselverwaltung erfordert die Trennung der Schlüssel nach Aufgaben (Arbeitsschlüssel, Signaturschlüssel zum Schlüsseltausch etc.) Dies ist hier nicht implementiert, kann aber mit Bordmitteln realisiert werden. Man trage einfach ein Protokollmodul mehrfach unter verschiedenen Feldindizes ein, und schon sind Schlüssel für ein und dasselbe Verfahren nach Verwendungszweck unterscheidbar.

Exportierte Schlüssel werden zusammen mit Metainformationen (Schlüsselkennung, Verfallsdatum und Protokollnummer) in einem Speicherbereich abgelegt und können dann in einer Datei gespeichert oder über Netzverbindungen übertragen werden. Ob der Schlüssel beim Export signiert oder mit einem Passwort verschlüsselt wurde, ist dort nicht vermerkt. Diese Information zu übermitteln, obliegt dem Anwendungsprogramm beziehungsweise dem Anwender. Das mag zunächst unsinnig erscheinen. Ist es aber nicht. Das Passwort zum Entschlüsseln muß ohnehin auf der anderen Seite eingegeben werden; damit wird implizit die Information übermittelt, daß entschlüsselt werden muß. Die Information, daß das Paket signiert ist, ist wertlos, denn sie kann unterwegs manipuliert werden – die Signatur ließe sich unbemerkt entfernen.

Wird ein Schlüssel für ein asymmetrisches Verfahren exportiert, extrahiert `tck_exportkey()` zunächst die öffentlichen Komponenten des Schlüssels. Dazu wird die Funktion `sec2pub()` des jeweiligen Protokollmoduls mit dem geheimen Schlüssel als Parameter aufgerufen.

Mit der Funktion `tck_importkey()` werden exportierte Schlüssel in eine andere Datenbank eingelesen. Die Funktion erwartet eine Bytefolge, wie sie von `tck_exportkey()` erzeugt wird, sowie gegebenenfalls wieder ein Passwort und die Kennung des Schlüssels, mit dem die Signatur geprüft werden soll. Der Import verläuft nur erfolgreich, wenn die Signaturprüfung positiv ausfällt und der importierte Schlüssel noch nicht in der Datenbank enthalten sowie sein Verfallsdatum nicht überschritten ist.

Abgerundet wird die Schlüsselverwaltung durch die Funktionen `tck_delkey()`, die einen Schlüssel mit gegebener Kennung löscht, sowie `tck_listkeys()`, die die Metainformationen zu allen in der Datenbank gespeicherten Schlüsseln in einer Liste liefert. Die Reihenfolge der Listeneinträge ist unbestimmt und hängt nur von DBM ab, Anwendungen können sie aber nach Belieben sortieren. Die Funktion `tck_listkeys()` reserviert Speicher für die zurückgegebene Liste. Er kann mit `tck_freeklst()` freigegeben werden.

5.2.3 Zufallszahlen

SSLeay stellt einen Pseudozufallszahlengenerator (PRNG) bereit, der für kryptographische Zwecke geeignet ist. Er wird zur Schlüsselerzeugung herangezogen und muß deshalb mit echten Zufallszahlen initialisiert werden. Aufeinanderfolgende Programmläufe würden sonst gleiche oder leicht ratbare Zufallsfolgen und damit unsichere Schlüssel hervorbringen.

Doch woher soll die »Saat« kommen, mit der der Generator gestartet wird? Gute Zufallsquellen wie der radioaktive Zerfall sind in der Praxis unbrauchbar, weil sie nicht als handliche Steckkarte für den PC verfügbar sind. Statt dessen muß der Rechner selbst echten Zufall liefern. Das ist möglich, denn Rechner haben Benutzer und jeder Benutzer ist eine Zufallsquelle [ECS94]. Man kann die Quelle entweder explizit anzapfen, indem man den Benutzer zu einer Eingabe auffordert und beispielsweise die Zeitintervalle zwischen seinen Tastendrücken mißt, oder implizit durch Betrachtung des Systemzustandes, der als Folge aller Benutzungshandlungen entstanden ist.

Wir gehen den zweiten Weg. Die interne Funktion `seedrng()`, die von `tck_slinit()` aufgerufen wird, sammelt aktuelle Informationen über den Zustand des Betriebssystems und gibt sie an den PRNG von SSLeay weiter. Verwendet werden:

- die aktuelle Zeit in größtmöglicher Genauigkeit,[38]
- die eigene Prozessnummer,
- die Liste gerade laufender Prozesse, wie sie von `ps(1)` ausgegeben wird, sowie
- das Ergebnis von `getrusage(1)`, das Informationen über Page Faults, Context Switches und ähnliches enthält.

Der Wert der gewonnenen Zufallsinformationen ist allerdings zweifelhaft. Zeit und Prozessnummer fließen auch in die Schlüsselkennungen ein, die nicht ausdrücklich geheimgehalten werden. Unter ungünstigen Umstüden kann dieser Teil der Initialisierungswerte

[38] Die Unix-Funktion `gettimeofday(3)` kann Mikrosekunden liefern, aber die tatsächliche Genauigkeit hängt von der Hardware und Systeminterna ab.

also einem Angreifer bekannt werden, vor allem wenn die Systemuhr nur eine grobe Auflösung bietet. Der Zufallsgehalt der Prozeßliste hängt von der Laufzeit und der Auslastung des Systems ab.

Fortgeschrittene Techniken zur Zufallsgwinnung setzt Peter Gutmanns Cryptlib ein, deren Quelltext zur Lektüre empfohlen sei. Dort werden allerlei weitere Zufallsquellen genutzt und ihre Qualität bewertet.

Von der Möglichkeit, den internen Zustand des Zufallsgenerators bei Programmende in einer Datei abzulegen und später weiterzuverwenden – SSLeay unterstützt das –, wird hier kein Gebrauch gemacht.

5.2.4 Datenstrukturen

Anwendungsprogramme, die die Sicherheitsschicht nutzen, kommen im wesentlichen mit drei Datenstrukturen in Berührung. Nachrichten und exportierte Schlüssel werden zusammen mit Längenangaben in einer Struktur vom Typ `tck_buffer_t` übergeben. Speicher für diese Struktur kann mit `tck_bmalloc()` reserviert und mit `tck_bmfree()` freigegeben werden. Gibt eine Funktion einen Zeiger auf `tck_buffer_t` zurück, ist der Aufrufer für die Freigabe verantwortlich.

Metainformationen über Schlüssel werden in der Struktur `tck_keyinfo_t` abgelegt. Sie enthält neben der Schlüsselkennung den Schlüsseltyp (geheim oder öffentlich), die Protokollnummer, für die der Schlüssel geeignet ist, sowie den Protokolltyp (symmetrische Verschlüsselung, MAC oder asymmetrisches Signaturverfahren), außerdem das Verfallsdatum in `time_t`-Darstellung. Die Funktion `tck_listkeys()` gibt ein Array von Zeigern auf solche Strukturen zurück.

Schlüsselkennungen werden in einem Speicherbereich fester Länge als `unsigned char[TCK_KEYID_LEN]` übergeben.

5.3 Nachrichtenschicht

Die Nachrichtenschicht bildet Ticketparameter, die in einer einheitlichen Datenstruktur vorliegen, auf Nachrichten ab und umgekehrt. Sie definiert mehrere Nachrichtenformate, die grob den Parametersätzen aus Kapitel 2 entsprechen. Darüber hinaus enthält sie Funktionen zur Gültigkeitsprüfung für jedes Format. Damit kann sie auf der Empfängerseite zum einen die Ticketparameter liefern, zum anderen auch gleich die Information, ob diese Parameter ein gültiges Billett darstellen. Theoretisch ist jedes der Nachrichtenfor-

Funktion(en)	Aufgabe
`tck_param2msg()`	erzeugt eine Nachricht aus einem Parametersatz
`tck_msg2param()`	liest Parametersatz aus einer Nachricht
`tck_valid_...()`	prüfen die Gültigkeit von Parametersätzen
`tck_nextnum()`	liefert eine eineutige Billettnummer

Tab. 5.5: *Funktionen der Nachrichtenschicht*

mate mit jedem Protokoll der Sicherheitsschicht einsetzbar. Das ist das, wovor in 3.3.1 gewarnt wird. Die symmetrische Verschlüsselung bietet nur zusammen mit geeigneten Nachrichten genügend Sicherheit. Bei den beiden anderen Verfahren ist das jedoch nicht kritisch.

5.3.1 Parameter

Die Nachrichtenschicht stellt Anwendungsprogrammen eine einheitliche Datenstruktur für die Billettparameter zur Verfügung. Sie ist unabhängig vom benutzten Nachrichtenformat und dient zwei Zwecken. Aus ihr werden nicht nur die Nachrichten gewonnen, die an die Sicherheitsschicht weitergegeben werden und schließlich als maschinenlesbares Aztec-Symbol Ticket abgelegt werden. Auch die natürlichsprachige Darstellung, die der Information des Käufers dient, entsteht aus dieser Struktur. Für diesen Teil ist jedoch allein das Verkaufsprogramm zuständig; die Nachrichtenschicht liefert ausschließlich die Datenstruktur.

Parameter	Typ	Bedeutung und Einheit
Ticketnummer	unsigned long	eindeutige Nummer zur Erkennung von Kopien
Veranstaltungsort	`ushort`	Nummer des Veranstaltungsortes
Veranstaltungsort	`uchar[64]`	Name des Veranstaltungsortes
Veranstaltung	`ushort`	Nummer der Veranstaltung
Veranstaltung	`uchar[64]`	Name der Veranstaltung
Anfangszeit	`time_t`	Anfangszeit in Sekunden seit 1970-01-01 00:00:00Z
Dauer	`ushort`	Dauer der Veranstaltung bzw. maximale Geltungsdauer in Sekunden
Einlaß	`ushort`	Zeitdifferenz zwischen Einlaß und Veranstaltungsbeginn
Sitzplatz	`ushort`	
Reihe	`ushort`	
Preisklasse	`uchar`	frei wählbar; möglichst druckbares ASCII-Zeichen
Preis	`ushort`	Preis in 0,01 Euro
Freitext 1	`uchar[64]`	
Freitext 2	`uchar[64]`	
Freitext 3	`uchar[64]`	

Mit *short* sind 16-Bit-Ganzzahlen bezeichnet, mit *long* ihre doppelt so langen Geschwister. Ein 'u' als erstes Zeichen des Typs zeigt an, daß es sich um einen vorzeichenlosen Typ handelt.

Tab. 5.6: *Mögliche Billettparameter*

Welche Parameter tatsächlich mit einem Wert belegt sind, zeigt die Variable `fields` an. Jedem Parameter ist darin ein Bit zugeordnet, das gesetzt ist, sobald dem Parameter

ein Wert zugewiesen wurde. Zu jedem Bit ist eine symbolische Konstante definiert, mit deren Hilfe es gesetzt, gelöscht und abgefragt werden kann.

Neben den Parametern enthält die Struktur Verwaltungsinformationen, namentlich ein Verfallsdatum. Die Kontrollstation muß benutzte Tickets nur solange speichern, wie sie aufgrund ihrer Parameter gültig sind. Später können Kopien ohnehin nicht mehr eingesetzt werden. Um die Pflege der Ticketliste zu erleichtern, wird jedem Eintrag schon bei der Kontrolle ein Verfallsdatum zugewiesen. Alte Einträge können dann automatisch zum Beispiel jede Nacht beseitigt werden.

Veranstaltungsort und -bezeichnung sind zweifach enthalten, einmal als Nummer und einmal als Text. Nummern sind einfacher zu vergleichen als Text. Sie lassen kaum Mehrdeutigkeiten zu; ein und derselbe Text kann hingegen unterschiedlich – z.B. "ü" vs. "ue", ISO 8859-1 vs. Codepage 437 – dargestellt werden. Zudem benötigen sie deutlich weniger Speicherplatz. Den spart man zwar nur, wenn man unbedingt muß, doch wir müssen. Das Aztec-Symbol soll möglichst klein sein und seine minimale Elementgröße ist vorgegeben.

Die drei Zeitangaben definieren den Geltungszeitraum. Dauer und Einlaß sind in Sekunden angegeben, weil sie so zum `time_t` passen und ohne Umrechnungen verarbeitet werden können. Die vorgegaukelte Genauigkeit ist allerdings mit Vorsicht zu genießen. Uhren gehen ungenau und Menschen halten sich nicht exakt an die vorgegebenen Zeiten. Die Ticketparameter können wir nicht großzügig wählen, denn sie werden auch noch für den Käufer lesbar aufgedruckt. Da der Sicherheitsaufschlag ein fester Wert ist, genügt es aber, ihn in der Kontrollstation zu berücksichtigen.

Der Inhalt der drei Freitextfelder geht unter keinen Umständen in eine Nachricht ein. Er dient ausschließlich der Kundeninformation und wird dementsprechend nur in natürlicher Sprache angezeigt.

5.3.2 Nachrichtenformate

Definiert sind zwei feste und ein variables Nachrichtenformat. Die beiden festen Formate sind sehr kompakt und speziell für den Einsatz von Blockchiffren in der Sicherheitsschicht ausgelegt.

Das Format `simple1` enthält die eindeutige Ticketnummer, die Veranstaltungsnummer sowie die Preisklasse. Es soll vor allem dann eingesetzt werden, wenn nur wenige Veranstaltungen zu unterscheiden sind, zum Beispiel in einem Museum, wo Eintrittskarten für wechselnde Ausstellungen verkauft werden, oder für Sportveranstaltungen an einem bestimmten Ort. Eine Nachricht dieses Formats ist nur 7 Bytes lang, paßt also in einen Block gängiger Blockchiffren.

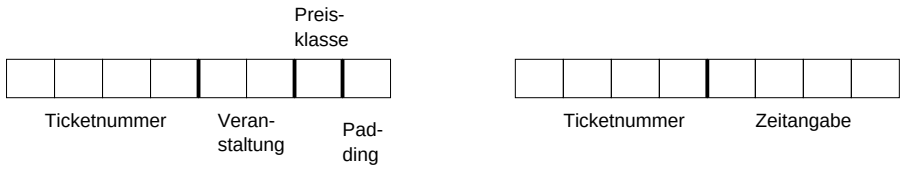

Abb. 5.2: *Die Formate* `simple1` und `simple2`

Das zweite feste Nachrichtenformat `simple2` enthält neben der Ticketnummer eine Zeitangabe im `time_t`-Format, die als Anfangszeit interpretiert werden kann, aber nicht muß. Auch dieses Format paßt mit 8 Bytes Nachrichtenlänge noch in einen Verschlüsselungsblock.[39]

Das dritte Format, `std`, hat einen variablen Inhalt. Es enthält genau die Parameter, die beim Erzeugen der Nachricht, in der Parameterdatenstruktur enthalten sind; sie werden einfach aneinandergehängt. Die beiden Stringvariablen werden dabei stets in voller Länge übernommen. Der Inhalt der Variablen `fields`, der anzeigt, welche Parameter mit Werten belegt sind, wird der Nachricht als Formatidentifikator vorangestellt. Zu jedem möglichen Formatidentifikator ergibt sich eine feste Nachrichtenstruktur, die unabhängig vom Inhalt ist. Auf diese Weise können beliebige Parameterkombinationen übermittelt werden, ohne daß Änderungen an der Software nötig sind. Zum Dekodieren der Nachricht wird lediglich der Identifikator ausgewertet, der das Format vollständig spezifiziert.

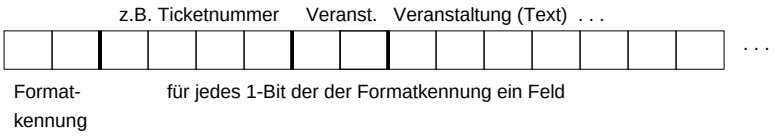

Abb. 5.3: *Das Format* `std`

Ein Spezialfall des `std`-Formates ist `notext`. Es ist analog aufgebaut, enthält aber die beiden Textfelder nicht, auch wenn sie in der Parameterstruktur belegt sind. Da die Prüfverfahren im nächsten Abschnitt ohnehin keine Strings auswerten, kann es genauso wie `std` eingesetzt werden und liefert deutlich kleinere Nachrichten.

Alle Zahlenwerte sind in den Nachrichten in *Network-Byteorder* (MSB zuerst) abgelegt, damit sie problemlos zwischen verschiedenen Rechnerplattformen austauschbar sind. Unix bietet zur Wandlung zwischen Host- und Network-Byteorder die Funktionen (bzw. Makros) `htonl(3)` und `ntohl(3)` sowie `htons(3)` und `ntohs(3)`.

Textfelder können bis zu 63 Zeichen aus dem Zeichenvorrat des *Latin alphabet No. 1* gemäß ISO 8859-1 (vgl. [Kor98]) enthalten. Auf das letzte Zeichen folgt ein Nullbyte; der Empfänger darf sich jedoch nicht darauf verlassen, daß es vorhanden ist.

5.3.3 Gültigkeitskontrolle

Jedem Nachrichtenformat ist eine Funktion zur Gültigkeitsprüfung zugeordnet, die gerade die darin enthaltenen Parameter berücksichtigt. Grundsätzlich können Formate und Kontrollverfahren aber frei kombiniert werden, wenn nur alle zur Kontrolle nötigen Parameter in der Nachricht enthalten sind. Die implementierten Verfahren sollen nur beispielhaft einige Möglichkeiten zeigen; andere können problemlos implementiert werden. Die

[39] SSLeay benutzt allerdings ein Padding-Verfahren, das in diesem Fall einen zweiten Block anfügt. Er enthält nur Füllbytes und bietet deshalb keine Manipulationsmöglichkeit. Man muß ihn nicht einmal mit übermitteln, sondern kann ihn auf Empfängerseite vor dem Entschlüsseln hinhalluzinieren.

Prüfung der eindeutigen Ticketnummern zur Erkennung von Kopien erfolgt an anderer Stelle (und nach der hier besprochenen Kontrolle).

Alle Kontrollfunktionen erwarten eine Parameterstruktur, wie sie beim Interpretieren einer Nachricht entsteht. Daneben sind abhängig vom Kontrollverfahren Werte oder Wertebereiche der geprüften Parameter anzugeben.

Mit den einfachen Nachrichtenformaten korrespondieren einfache Kontrollen. Die Funktion `tck_valid_simple1()` vergleicht die Veranstaltungsnummer des Tickets mit einem vorgegebenen Wert und die Preisklasse mit einer Liste der zulässigen Angaben. Das Ticket ist gültig, wenn beide Prüfungen positiv ausfallen. Die Preisklasse, die nach der erfolgreichen Prüfung sicher im vorgesehenen Wertebereich liegt, kann dann weiter ausgewertet werden, um beispielsweise zu signalisieren, daß noch der Studentenausweis o.ä. zu prüfen ist.

Für das zweite Nachrichtenformat, das lediglich eine Zeitangabe enthält, ist die Funktion `tck_valid_simple2()` vorgesehen. Sie erwartet neben dem Parametersatz eine Uhrzeit (Stunde, Minute, Sekunde). Geprüft wird, ob die Angabe auf dem Ticket zum einen dem aktuellen Datum entspricht, zum anderen der angegebenen Uhrzeit. Das ist nicht für Demonstrationszwecke praktischer als die Intervallösung aus Kapitel 2, sondern eignet sich auch zur Herstellung von Tickets, die einen ganzen Tag lang gelten sollen. Dazu wählt man einfach eine feste Uhrzeit, zum Beispiel 12:00 Uhr, und verkauft Tickets für diese Uhrzeit am jeweiligen Geltungstag.

Die Dritte im Bunde ist `tck_valid_std()`. Hier werden Preisklasse und Veranstaltungskennung genauso geprüft wie oben bereits beschrieben. Hinzu kommt die Nummer des Veranstaltungsortes, die ebenfalls mit einem festen Wert verglichen wird. Über ein Flag kann die Püfung der Zeitangaben veralaßt werden. In diesem Fall muß zusätzlich die aktuelle Zeit in dem Intervall (Anfangszeit-Einlaß, Anfangszeit+Dauer) liegen, damit der Parametersatz als gültiges Billett akzeptiert wird.

Bei allen Prüfverfahren bleiben der Preis sowie Reihe und Nummer des reservierten Platzes unberücksichtigt. Diese Parameter könnten noch mit der Preisklasse abgegelichen werden, aber das trägt kaum zur Sicherheit bei. Zudem hängt diese Prüfung von örtlichen Gegebenheiten ab und ist deshalb im Prototypen nicht implementiert.

Überhaupt lassen alle Parameter semantischen Spielraum und können bei konkreten Installationen verschieden interpretiert werden. So läßt sich die Veranstaltungsnummer beispielsweise in einem Kino entweder einem Film oder aber einer Vorstellung zuordnen. Bezeichnet sie eine Vorstellung, werden Zeitangaben unnötig. Veranstaltungsorte müssen nur angegeben sein, wenn ein Veranstalter über deren mehrere verfügt. Der Veranstalter selbst ist bereits implizit durch die Arbeit der Sicherheitsschicht festgelegt, denn nur er verfügt über die geheimen Schlüssel zur Billetterzeugung. Damit ist ausgeschlossen, daß Eintrittskarten eines anderen Veranstalters benutzt werden können.

5.3.4 Billettnummern

Der Nachrichtenschicht ist auch die Erzeugung eindeutiger zugeordnet, obgleich sie nicht automatisch bei der Erzeugung von Nachrichten erfolgt. Anwendungsprogramme müssen

selbst die Funktion `tck_nextnum()` aufrufen und ihren Rückgabewert in die Parameterstruktur kopieren.

Eindeutige Nummern können auf drei Arten erzeugt werden. Übernimmt ein einziger Prozeß den Verkauf, genügt ein einfacher Zähler, dessen Zustand zusätzlich in einer Datei abgelegt wird, damit er auch nach Abstürzen oder einem Stromausfall noch verfügbar ist. Zusammen mit der vorgeschlagenen Trennung von Verkaufssystem und Webserver läßt sich diese Lösung gut implementieren.

Der Prototyp soll jedoch für den Verkauf als CGI-Programm direkt vom Webserver aufgerufen werden. Das bedeutet, daß jedes Ticket von einem anderen Prozeß verkauft wird. Wie können diese Prozesse eindeutige Nummern generieren? Eine Möglichkeit ist in der Sicherheitsschicht für die Schlüsselkennungen implementiert. Sie nutzt die eindeutigen Prozeßnummern, die vom Betriebssystem vergeben werden. Leider werden die Kennungen dabei recht lang. Bei der Schlüsselverwaltung stört das nicht weiter, aber auf dem Ticket müssen wir Platz sparen.

Zur Erzeugung von Ticketnummern wurde daher ein anderer Weg beschritten, der kurze Nummern liefert. Eine Datei dient als globaler Zähler für alle Prozesse. Sie enthält stets die letzte vergebene Nummer. Ein Prozeß, der eine neue, noch nie benutzte Nummer benötigt, öffnet und sperrt zunächst diese Datei. Ist sie bereits gesperrt, wartet er solange, bis sie vom sperrenden Prozeß freigegeben wird. Dann erhöht er den Zähler und erhält so seine Nummer, die er in die Datei zurückschreibt, bevor die Sperrung aufgehoben wird. Das Sperren ist notwendig, weil Prozesse jederzeit vom Betriebssystem unterbrochen werden können, es sei denn, sie führen gerade eine atomare Operation aus. Die Systemrufe zum Sperren arbeiten atomar.

5.4 Anwendungprogramme

Die beiden vorigen Abschnitte behandeln die grundlegenden Funktionen, die von einem Parametersatz auf der Verkaufsseite über eine Nachricht zu einer (Teil)Entscheidung »gültig oder nicht« auf der Kontrollseite führen. Darauf aufbauend entstanden Programme für den Verkauf (oder jedenfalls eine Verkaufssimulation) und die Kontrolle sowie zur Schlüsselverwaltung.

5.4.1 Verkauf

Das Verkaufsprogramm `sale` des Prototypen ist stark vereinfacht. Es handelt sich um ein CGI-Programm, das im HTTP-Request sämtliche Ticketdaten erhält. Auch alle weiteren Parameter, etwa die Schlüsselkennung, das Nachrichtenformat und der Pfad zur Schlüsseldatenbank, werden auf diese Weise angegeben. Im richtigen Leben läse man sie selbstverständlich aus einer Konfigurationsdatei.

Die CGI-Anbindung erfolgt wieder mittels cgic [Bou96a]. Zunächst werden in der Funktion `gettparam` die Ticketparameter gelesen. Ihre Namen entsprechen denen in der Parameterstruktur `tck_tparam_t`. Danach wird eine Ticketnummer generiert.

Die eigentliche Billetterzeugung erfordert nur wenige Funktionsaufrufe. Zuerst entsteht aus dem Parametersatz eine Nachricht, wobei einer der oben vorgestellten Nach-

Parameter	Typ	Bedeutung
`keydbpath`	String	Pfad zur Schlüsseldatenbank
`countpath`	String	Pfad zur Zählerdatei für die Ticketnummern
`keyid`	16 Stellen hexadezimal in ASCII-Darstellung	Schlüsselkennung
`msgtype`	{1,...,4}	Nachrichtenformat; 1 ≙ `simple1`, 2 ≙ `simple2`, 3 ≙ `std`, 4 ≙ `notext`

Tab. 5.7: *Zusätzliche Parameter für das Verkaufsprogramm*

richtentypen ausgewählt wird. Diese Nachricht geht dann zusammen mit der Schlüsselkennung weiter an die Sicherheitsschicht. Das liefert die Daten für das Aztec-Symbol.

```
tck_tparam_t *tparam;
tck_buffer_t *msg, barmsg;

msg = tck_param2msg( tparam, msgtype );
tck_slinit( keydbpath, TCK_FLG_NONE );
barmsg = tck_protectmsg( msg, keyid );
tck_slclose();
```

Die Fehlerbehandlung ist weggelassen. Bevor die Sicherheitssschicht genutzt werden kann, ist sie noch zu initialisieren und danach zu schließen. Die drei letzten Funktionsaufrufe könnten auch per Remote Procedure Call auf einem anderen Rechner ausgeführt werden, der besonders gegen Angriffe und damit gegen Ausspähen der Schlüssel gesichert ist.

Aus der Parameterstruktur `tparam` und der gesicherten Nachricht `barmsg` wird schließlich die HTML-Darstellung der Eintrittskarte gebildet. Das Aztec-Symbol wird in Form eines IMG- oder OBJECT-Links auf den Aztec-Generator mit der Nachricht als Parameter eingebettet. Der Browser des Käufers kümmert sich dann selbständig um den Abruf des Symbols. Vorher ist noch ein wenig Nacharbeit nötig. Beim Lesen des Symbols liefert der Scanner einen Datenstrom, der mit `CR LF` endet. Um das Ende daran eindeutig erkennen zu können, darf diese Zeichenfolge nicht in den Daten vorkommen. Das Programmodul für den Barcodescanner stellt deshalb eine Funktion `tck_msg2url` bereit, die dies erledigt[40] und nebenbei gleich noch die URL-Kodierung gemäß RFC 1738 vornimmt.

Die Elementgröße des Aztec-Symbols wird abhängig von der Nachrichtenlänge zwischen 4 und 8 Pixeln variiert, so daß kurze Nachrichten zusätzlich gegen schlechte Druckqualität geschützt sind.

[40] Jedes vorkommende `CR` wird verdoppelt.

5.4.2 Kontrolle

Das Kontrollprogramm `checkpoint` erhält in der Kommandozeile alle nötigen Parameter. Danach liest es in einer Endlosschleife Symbole und prüft, ob es sich um gültige Tickets handelt. Die enthaltenen Parameter und die Entscheidung werden auf dem Bildschirm angezeigt.

Neben den bereits behandelten Grundfunktionen greift das Kontrollprogramm auf zwei weitere Programmodule zurück. Das eine liest Daten von einer seriellen Schnittstelle, an die ein Barcodescanner angeschlossen ist. Es soll hier nicht weiter besprochen werden, da es nicht besonders elegant arbeitet und für das Thema dieser Arbeit keine weitere Bedeutung hat. Informationen über die Programmierung serieller Schnittstellen unter Unix sind in den Manual-Seiten[41] sowie in [Gal94] zu finden. Wir müssen hier nur wissen, daß nach Initialisierung des Scanners mit der Funktion `tck_read_symbol()` genau ein Symbol gelesen wird und daß der einzige Parameter der Funktion den Scanner bezeichnet – es könnte ja mehrere geben.

Zum anderen sind die Numemrn der kontrollierten Tickets zu speichern und bei der Kontrolle zu prüfen, ob das Ticket bereits benutzt wurde. Dazu wird wieder DBM verwendet. Die enstprechenden Funktionen sind im Modul `cpdb` zusammengefaßt. Die wichtigste

Funktion	Aufgabe
`tck_cpdbinit()`	öffnet die DBM-Datenbank
`tck_close_cpdb()`	schließt die DBM-Datenbank
`tck_addtodb()`	speichert ein Ticket unter seiner Nummer
`tck_getticket()`	liest das Ticket aus der Datenbank
`tck_delticket()`	löscht ein Ticket
`tck_expire()`	entfernt alte Einträge

Tab. 5.8: *Funktionen für die Ticketdatenbank*

Funktion ist `tck_addtodb()`. Sie versucht, ein als Parametersatz (`tck_tparam_t`) übergebenes Ticket in der Datenbank abzulegen. Ist unter der Nummer dieses Tickets bereits ein anderes abgelegt, meldet die Funktion einen Fehler. Die Prüfung auf Einmaligkeit und das Speichern können so in einem Schritt erledigt werden.

Bevor ein Ticket in der Datenbank abgelegt wird, erhält es ein Verfallsdatum, das aus seinen Zeitparametern oder dem Kontrollzeitpunkt abgeleitet werden kann. Anhand des Verfallsdatums sortiert die Funktion `tck_expire()` später alle alten Einträge aus.

In der Datenbank wird die komplette Parameterstruktur abgelegt. In der vorliegenden Programmversion hat das keinen tieferen Sinn, schadet aber auch nicht. Zusammen mit einer Verkaufsprotokollierung könnte es zur Betrugserkennung eingesetzt werden.

Die Billettkontrolle umfaßt vier Schritte. Zuerst wird über den Barcodescanner ein Symbol gelesen. Das liefert eine Bytefolge, die zusammen mit einer Schlüsselkennung an die Sicherheitsschicht übergeben wird. Außer bei symmetrischer Verschlüsselung fallen

[41] `termios(3)` und `termio(7)`

Manipulationen hier auf. Die Sicherheitsschicht liefert entweder eine Nachricht oder aber einen Fehler.

Weiter geht es mit der Nachrichtenschicht, die aus der Nachricht einen Parametersatz generiert. An dieser Stelle werden die Parameter auf den Bildschirm ausgegeben. Der nächste Kontrollschritt ist die Parameterprüfung. Verläuft auch sie positiv, erfolgt als letztes der Speicherversuch in der Datenbank. Ist das Ticket kein Duplikat eines vorher benutzten, wird es als gültig akzeptiert. Im vereinfachten Quelltext sieht das so aus:

```
tck_buffer_t *bardta;
tck_buffer_t *msg;
tck_tparam_t *param;

bardta = tck_read_symbol( rdr );
msg = tck_unprotectmsg( bardta, keyid );
param = tck_msg2param( msg, msgtype );
/* ... Ausgabe der Parameter ...
 */
tck_valid_simple1( param, event, categ );
/* oder tck_valid_simple2( param, hour, min, sec )
 * oder tck_valid_std( param, location, event, categ, TRUE )
 */
/* ... Verfallsdatum berechnen ...
 */
tck_addtodb( param );
```

Tritt bei einem der Schritte ein Fehler auf, wird das Billett zurückgewiesen. Die Reihenfolge ist sinnvoll. Die ersten beiden Schritte lassen ohnehin keine Wahl und Einträge in die Datenbank sollen nur für ansonsten gültige Tickets versucht werden. Ein fehlerhaftes Billett könnte sonst seine Nummer ungültig machen, ohne benutzt worden zu sein.

Parameter	Bedeutung
`-l integer`	Veranstaltungsort (Nr.)
`-e integer`	Veranstaltung (Nr.)
`-c string`	zugelassene Preisklassen
`-b hh:mm`	Anfangszeit (für Format `simple2`)
`-i string`	Schlüsselkennung für die Kontrolle
`-t {1,2,3,4}`	Nachrichtentyp
`-k string`	Pfad zur Schlüsseldatenbank
`-d string`	Pfad zur Ticketdatenbank
`-1`	Programm nach einer Kontrolle beenden

Tab. 5.9: *Kommandozeilenparameter des Kontrollprogramms*

Abschließend sind in Tabelle 5.9 die Kommandozeilenparameter aufgeführt, mit denen das Kontrollprogramm aufgerufen werden kann. Die ersten vier Parameter dienen der

Kontrolle, die übrigen sind technischer Natur und müssen bis auf `-1` immer angegeben sein.

5.4.3 Schlüsselverwaltung

Der Schlüsselverwalter `keymgr` wird interaktiv benutzt und man kann mit ihm Schlüssel erzeugen, löschen, in eine Datei exportieren und aus einer Datei importieren sowie eine Liste der vorhandenen Schlüssel anzeigen. Als Kommandozeilenparameter verlangt er lediglich den Pfad zur Schlüsseldatenbank.

Da das Programm `keymgr` lediglich eine Benutzerschnittstelle zu den bereits besprochenen Verwaltungsfunktionen der Sicherheitsschicht bildet, wird er hier nicht in seine Bestandteile zerlegt. Er ist ohnehin nur ein Hilfsmittel.

5.5 Beispieltickets

Wie sehen die Eintrittskarten nun eigentlich aus, die der Käufer mit seinem Drucker zu Papier bringt? Weil sie nirgends so recht hinpassen, wird den Beispielen ein eigener Abschnitt gewidmet. Die Größe der Aztec-Symbole vermittelt jeweils einen Eindruck von der übermittelten Datenmenge.

Abbildung 5.4 zeigt die Minimalvariante. Eine kurze Nachricht ist durch Verschlüsselung mit IDEA gesichert. Die resultierenden acht Bytes könnten auch als eindimensionaler Strichkode gedruckt werden. Die Textzeile mit den Öffnungszeiten entstammt einem der

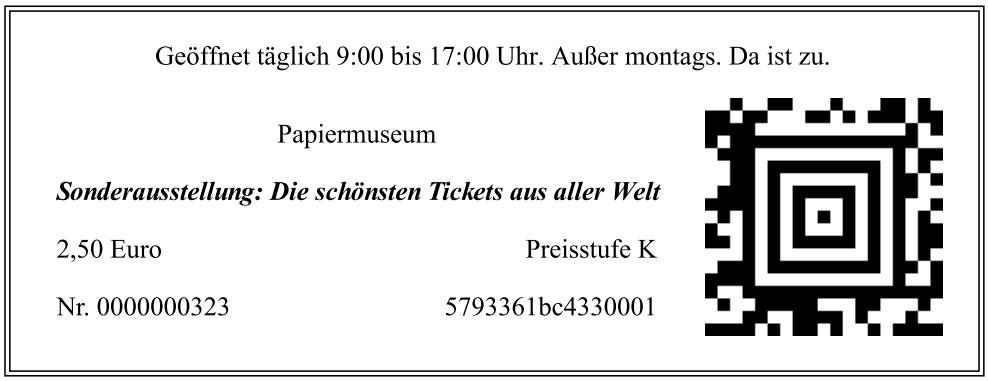

Abb. 5.4: *8 Bytes genügen*

drei Freitextfelder. In der letzten Zeile ist neben der Ticketnummer auch die Kennung des verwendeten Schlüssels angegeben. Gebraucht wird sie dort nicht; die Kontrollstation weiß, mit welchem Schlüssel sie die Eintrittskarten prüfen soll, falls sie überhaupt mehr als einen besitzt.

Das zweite Ticket (Bild 5.5) nutzt ein ebenso kompaktes Nachrichtenformat, ist aber mit einem MAC gesichert. Die Zeitangabe des Formats `simple2` wird hier nur für das Datum verwendet; für die Uhrzeit kann man einen festen Wert wählen.

Leipziger Messe

Fachmesse Ticket '98

Donnerstag
8.10.1998

Nr. 0000000331 5793361bc4620003

Abb. 5.5: *Eine kurze Nachricht mit MAC*

Das dritte Beispiel in Abbildung 5.6 zeigt fast die Maximallösung. Alle Parameter mit Ausnahme der Texte für Ort und Veranstaltung sind im Symbol kodiert und mit einer RSA-Signatur versehen. Nimmt man die beiden Texte hinzu, wird das Symbol noch etwas größer und recht unansehnlich.

Theater am Rande der Stadt

Aus dem Leben eines Billettabreißers

Donnerstag 8.10.1998	Beginn: 20:00 Uhr Einlaß: 19:30 Uhr
Reihe 3	Platz 11
15,00 Euro	Preisstufe F
Nr. 0000000329	5793361bc4540002

Abb. 5.6: *Ort und Veranstaltung sind nur als Nummer im Symbol kodiert*

6 Und nun?

Hinterfragen Hinterfragt wird heute z.B. das Atlantische Bündnis, der Marxismus sowie der hinterletzte Mindersinn. Dieser vor allem.
(Eckhard Henscheid: Dummdeutsch)

Geschafft. Programm läuft. Parameter gesammelt, Schlüssel verwaltet, Eintrittskarten gedruckt, Ecken abgerissen. Der Praxistest steht noch aus. War ich gut?

6.1 Da kann man nichts tun

Wir haben ein System gebaut, das allerlei Angriffen standhalten wird. Dem folgenden nicht. Es handelt sich um einen Denial-of-Service-Angriff, der zwar nicht unbedingt Betrugsmöglichkeit eröffnet, aber Schaden anrichten kann, indem er Kontrollstationen außer Betrieb setzt.

Die Spezifikation des Aztec-Codes erlaubt die Herstellung sogenannter *Menu*-Symbole. Ihr Inhalt wird vom Scanner selbst interpretiert und nicht an den Rechner übertragen. Solche Symbole dienen dazu, den Barcode-Leser zu konfigurieren. So enthält das Handbuch zum Welch Allyn Imageteam 4400 unzählige Seiten voller Aztec-Symbole, mit denen zum Beispiel die Übertragungsgeschwindigkeit und das Datenformat für die serielle Schnittstelle, das Verhalten des Gerätes und andere Einstellungen verändern lassen. Ein *Menu*-Symbol ist durch ein einzelnes Bit in der Mode Message gekennzeichnet und mit bloßem Auge nicht von anderen Symbolen zu unterscheiden.

Das macht den Umgang mit dem Scanner einfach - im Handbuch blättern, Symbol lesen, fertig. Da es sich jedoch um eine Form der Inband-Signalisierung handelt – Daten und Steuerinformationen werden auf ein und demselben Weg übertragen –, bietet sich damit auch ein Ansatzpunkt für Angriffe auf das System. Das sollen zunächst zwei Beispiele illustrieren.

In einem Telefonnetz werden Leitungen zwischen Teilnehmern vermittelt. Dazu müssen Teilnehmeranschlüsse und Ortsvermittlungsstellen sowie die Vermittlungsstellen untereinander Signalisierungsinformationen austauschen, etwa »benötige eine Verbindung zu Anschluß 0341-3385775«, »der gerufene Anschluß ist zur Zeit besetzt« oder »Verbindungsabbau«. In älteren Telefonnetzen erfolgte die Signalisierung zwischen den Vermittlungsstellen auf den gleichen Kanälen wie die Sprachübertragung. Zwischen den Vermittlungsstellen wurden einfach Töne in bestimmten Frequenzen gesendet. Damit hatte jeder Teilnehmer die Möglichkeit, während eines Gesprächs Signalisierungsinformationen an Vermittlungsstellen irgendwo auf dem Wege zum gerufenen Anschluss zu senden, indem er entsprechende Töne erzeugte.

Man konnte zum Beispiel eine gebührenfreie Nummer anrufen, die zu einem Anschluß im Ausland führte. Danach sendete man geeignete Signalisierungstöne und baute so die

Verbindung teilweise wieder ab. Ergebnis war ein offener Kanal bis zur Auslandsvermittlung, mit dem man eine neue Verbindung zu einer beliebigen Zielrufnummer aufbauen konnte. Der Gebührenzähler in der OVSt merkte davon nichts. Er zählte weiter brav die Einheiten für einen gebührenfreien Anruf, also gar keine. Auf diese Weise wurden über die Jahre erhebliche Schäden angerichtet, bis schließlich die Telefonnetze zumindest der bedeutendsten Industrieländer auf modernere Systeme umgestellt werden konnten, in denen für die Steuerinformationen besondere Kanäle zur Verfügung stehen.

Ein weiteres Beispiel liefert das Programm Crosspoint. Dabei handelt es sich um eine Software zur Teilnahme am E-Mail-Verkehr sowie an Diskussionsforen. Das Programm ist für den Offline-Betrieb ausgelegt. Von einem Serversystem werden Nachrichten gesammelt und in komprimierten Paketen abgelegt; Crosspoint holt diese Pakete selbständig ab, packt sie aus und sortiert die Nachrichten in eine lokale Datenbank ein.

Mit einem Update wurde die Software um ein Feature erweitert. Der Benutzer kann jetzt Filter einbinden. Jede empfangene Nachricht wird dann zunächst an ein Filterprogramm geleitet, das sie nach beliebigen Kritierien untersuchte und Crosspoint mitteilt, was mit der Nachricht geschehen soll (z.B. »lschen« oder »Wichtig! Hervorheben!«). Diese Information schreibt der Filter in eine Headerzeile der Nachricht, die Crosspoint dann auswertet.

Die Aufregung unter den Benutzern war groß, als sich herausstellte, daß diese nur lokal genutzte Haederzeile auch vom Absender einer Nachricht erzeugt werden konnte. Wer sich für besonders wichtig hielt, ließ seine Newsgroup-Postings von Crosspoint hervorgehoben darstellen. Das funktionierte sogar dann, wenn der Leser überhaupt keinen Filter installiert hatte.

Die Inband-Signalisierung birgt auch beim Einsatz eines Barcodelesers zur Ticketkontrolle Gefahren. Die gelesenen Symbole kommen nicht aus einer vertrauenswürdigen Quelle; erst durch Lesen des Symbols soll ja festgestellt werden, ob es sich um gültige Tickets handelt. Man muß also damit rechnen, daß dem Scanner beliebige Symbole vorgelegt werden.

Mittels eines *Menu*-Symbols ist ein Denial-of-Service-Angriff leicht auszuführen. Der Angreifer muß lediglich das Gerät so weit umkonfigurieren, daß es nicht mehr benutzbar ist. Das ist sehr einfach. Es genügt schon, die Übertragungsgeschwindigkeit an der seriellen Schnittstelle oder das Übertragungsprotokoll zu verändern.

Setzt ein Angreifer auf diese Weise sämtliche Lesegeräte einer Kontrollstation außer Betrieb, so kann er sich möglicherweise im folgenden ohne Bezahlung in eine Veranstaltung schleichen. Der Veranstalter wird seine Kunden im Falle einer offensichtlichen Computerpanne nicht abweisen, sondern im Zweifel lieber unkontrolliert einlassen. Ein solcher Betrug ist allerdings dermaßen frech, daß sich kaum einer trauen wird, ihn umzusetzen. Im Wiederholungsfall bringt er zudem ein hohes Entdeckungsrisiko mit sich.

Aus gleichem Grunde empfiehlt es sich, sämtliche Steuerinformationen, die der Kontrollstation übermittelt werden müssen (neue Schlüssel, Veranstaltungsdaten etc.) auf einem eigenen Kanal zu übertragen. Das kann eine temporäre Netzverbindung, eine Diskette oder eine Chipkarte sein.

6.2 Was fehlt

Integration in das Bezahlprotokoll Der Prototyp des Verkaufsprogramms beantwortet einen HTTP-Request, der sämtliche Parameter enthält, mit einem Ticket, und interessiert sich nicht für Geld. Sollen tatsächlich Tickets verkauft werden, muß dieser Vorgang in das Bezahlprotokoll integriert werden, und zwar so, daß der Kunde dabei nicht manipulieren kann.

Das ist einfach, wenn die Bezahlung auch nur einen HTTP-Request erfordert und die Parameter dabei vom Browser des Käufers zum Server übertragen werden können. Bei Kreditkartenzahlung zum Beispiel könnte der Kunde die gewählten Parameter zusammen mit Eingabefeldern für die Kartendaten angezeigt bekommen. Hat er seine Angaben eingetragen, wird alles zusammen zum Server geschickt und als Antwort kommt das Ticket zurück.

Wie das Problem bei anderen Zahlungsverfahren gelöst ist, wurde nicht untersucht. Vermutlich ist es aber für jedes auf irgendeine Weise vernünftig gelöst, denn Netzgeld soll gerade auch zum Kauf digitaler Waren benutzt werden.

Verhalten bei Netzstörungen Kommt es nach dem Bezahlen bei der Übertragung des HTML-Dokuments oder des Aztec-Symbols zu einem Verbindungsabbruch, ist der Käufer sein Geld los und bekommt keine Gegenleistung. Auch dieses Problem muß das Zahlungsprotokoll lösen, was nicht Thema der vorliegenden Arbeit ist.

Kreditkarten sind auch in dieser Hinsicht unproblematisch, denn der Käufer kann die Zahlung zumindest nachträglich rückgängig machen, sofern er seine Kartenabrechnung aufmerksam liest.

Integration des klassischen Verkaufs Der Verkauf von Eintrittskarten im Internet wird das herkömmliche Verfahren nicht so schnell ablösen, sondern nur ein Zusatzangebot sein. Das wirft zwei Probleme auf. Zum einen müssen sowohl das Verkaufssystem im Netz (das wahrscheinlich bei einem Internet-Provider steht) als auch zum Beispiel die altertümliche Kino- oder Theaterkasse Plätze reservieren können. Allzu hohe Kommunikationskosten dürfen dabei aber nicht anfallen.

Das Problem ist auf verscheidene Weise lösbar. Der Veranstalter kann für jede Verkaufsart ein Platzkontingent bereitstellen. Schwierigkeiten bringt das bei fast ausverkauftem Haus mit sich, wenn Karten nur noch an einer Stelle zu haben sind. Aber das passiert genauso, wenn es mehrere Vorverkaufsstellen gibt. Ist an der Kasse bereits ein Computersystem installiert und werden nur wenige Tickets im Netz verkauft, kann man Wählverbindungen nutzen, die bei jedem Online-Verkauf aufgebaut werden. Das kostet nur ein paar Pfennige pro Ticket.

Das andere Problem ist die Kontrolle. Sie sollte für alle Besucher auf die gleiche Weise erfolgen. Bei hohem Andrang ergeben sich sonst störende Verzögerungen, wenn beispielsweise ein Kontrolleur ständig zwischen Scannen und Abreißen wechseln muß. Auch hier gibt es mehrere Möglichkeiten. Entweder man verkauft auch an der herkömmlichen Kasse maschinenlesbare Eintrittskarten und kontrolliert alle Besucher mit dem Barcodescanner.

Oder aber man bietet eine Möglichkeit, das online gekaufte Ticket am Veranstaltungsort ohne Anstehen und Warten gegen ein herkömmliches einzutauschen.

Ein Billett für mehrere Personen Der eine oder andere fände es schrecklich cool, wenn auf dem online gekauften Ticket auch stehen könnte: »Gültig für vier Personen.« [Hue98]. Ist es aber nicht, jedenfalls dann nicht, wenn das Ticket am Eingang kontrolliert werden soll. Gilt es für mehr als eine Person, liefert die Kontrolle keine einfache Ja/nein-Entscheidung mehr. Das macht alles komplizierter.

Ticketprüfung für jedermann Werden digitale Signaturen zur Sicherung benutzt, kann der Kunde den öffentlichen Schlüssel bekommen und damit die Herkunft seines Tickets prüfen. Ein Argument gegen symmetrische Verfahren? Mitnichten. Der Käufer besitzt ohnehin keinen Barcodescanner und wird sich die Bytes auch nicht einzeln aus einer HTML-Seite zusammenklauben. Außerdem beweist die Signatur wirklich nur eines, die Herkunft. Aus einer gültigen Signatur kann nicht auf ein gültiges Ticket geschlossen werden. Und den Käufer vor Betrügern schützen kann die Signatur auch nicht. Wenn er beim Server seiner Betrüger kauft, wird er sich auch einen falschen Schlüssel unterjubeln lassen.[42]

Verkaufsprotokolle Der Prototyp führt beim Verkauf keine Protokolle. Das sollte schnellstens geändert werden, um die Mißbrauchserkennung zu ermöglichen. Protokolliert werden können neben den Verkaufsvorgängen auch alle kryptographischen Operationen (Schlüsselbenutzung, Schlüsselexport etc.).

Erneuter Druck Mancher [Hue98] hält es für einen Nachteil, daß sich die online gekauften Tickets nicht problemlos auf der Festplatte speichern lassen. Das Aztec-Symbol ist über einen Hyperlink auf den Symbolgenerator in ein HTML-Dokument eingebunden. Beim Speichern legen gängige Browser nur das HTML-Dokument im Dateisystem ab, nicht aber das Bild, das der Symbolgenerator liefert. Um das Billett im Falle einer Beschädigung erneut drucken zu können, ist erneute eine Online-Verbindung notwendig. (Der Käufer kann natürlich auch das Symbol gesondert speichern, aber das ist umständlich.)

Ist das wirklich ein Nachteil? Nein. Für ein paar Pfennige kann man eine Eintrittskarte wiederherstellen, die versehentlich in die Waschmaschine geraten ist. Wo gibt es das sonst? Jede vollständige Lösung des scheinbaren Problems brächte allerlei neue Schwierigkeiten mit sich.

6.3 Fehler und Verbesserungsmöglichkeiten

Kryptische Kennungen Die Schlüsselkennungen der Sicherheitsschicht – sechzehnstellige Hexadezimalzahlen – erwiesen sich schnell als unhandlich. Das Programm zur

[42] Genaugenommen werden sich die Kunden alles mögliche unterjubeln lassen, trotz SSL und Serverzertifikat. Sie sind nämlich keine Kryptologen und klicken im Zweifel immer auf »OK«.

Schlüsselverwaltung schafft Abhilfe, indem es Schlüssel numeriert auflistet und den Benutzer nur nach diesen Nummern fragt. Das sollte auch genügen, denn die implementierte Schlüsselverwaltung ist ohnehin eine Spezialität des Prototypen. Für den praktischen Einsatz schlägt Kapitel 3 langlebige Schlüssel vor.

Schlangenöl Überhaupt macht die Schlüsselverwaltung ihren Entwickler nicht glücklich. Sie ist reines Schlangenöl [Cur98]. Entweder man läßt sie ganz und gar weg und folgt den Empfehlungen des 3. Kapitels oder man implementiert sie richtig und unterscheidet beispielsweise die Schlüssel nach dem Verwendungszweck. Die derzeitige Lösung dient in erster Linie der Gewissensberuhigung.

Zeitprobleme Was den Banken ihr Y2K,[43] ist der Unix-Welt das Jahr 2038. Dann nämlich ist der Wertebereich des Typs `time_t` ausgeschöpft. Eine Vergrößerung des Wertebereichs seitens des Betriebssystems löst das Problem nur teilweise, denn die Nachrichtenformate der Tickets ändern sich nicht automatisch mit. Eigentlich genügte es auch, den Bezugszeitpunkt von 1970-01-01 auf ein späteres Datum zu verschieben. Der Prototyp bietet dafür jedoch keine Unterstützung.

Problematisch sind weiterhin Zeitzonen und Sommerzeit. Dem wurde bei der Entwicklung wenig Beachtung geschenkt. Abhängig von der Systemkonfiguration und den benutzten Funktionen werden Zeitangaben unterschiedlich interpretiert. Auf dem Ticket sollten sie deshalb grundsätzlich in UTC angegeben sein; für die Eingabe von Veranstaltungen sowie die natürlichsprachige Parameterdarstellung sind sie dann in Ortszeit umzurechnen.

Veranstaltungsnummern Die Kennzeichnung des Geltungsereichs durch Veranstaltungsnummern erfordert einige Sorgfalt. Die Nummern – im Prototypen sind 16-Bit-Ganzzahlen vorgesehen – dürfen sich nicht allzu oft wiederholen, sonst sind Replay-Angrife möglich, das heißt eine alte Eintrittskarte kann unter Umständen für eine spätere Veranstaltung mit gleicher Nummer wiederverwendet werden. Möglich wird das, weil die Ticketdatenbank der Kontrollstation regelmäßig von alten Einträgen befreit werden soll.

6.4 Ein Nebeneffekt

Da wir die Kopierbarkeit der online verkauften Eintrittskarten vorausgesetzt und hingenommen haben, ergibt sich eine interessante Nebenwirkung. Kapitel 4 zeigt, daß Aztec-Symbole bei genügender Elementgröße auch eine Faxkopie überstehen. Solche Tickets lassen sich also problemlos per Fax übertragen. Damit kann man sie auch ohne Internet Verkaufen, etwa per Telefon gegen Nennung der Kreditkartendaten. Auch der Kunde kann diese Möglichkeit nutzen, und beispielsweise seine Angebetete per Fax ins Kino einladen.[44]

[43] Y2K ist im englischen Sprachraum die gebräuchliche Abkürzung für das Jahr-2000-Problem.
[44] Wie das Objekt der Begierde darauf reagiert, wird hier nicht behandelt.

6.5 Weitere Anwendungen

Elektronische Schecks Digitales Geld ist toll. Leider läßt es sich schlecht verschenken. Geldgeschenke sind aber recht beliebt. Für geeignete Zahlungssysteme ist es denkbar, aus dem Geld, das zum Beispiel in irgendeiner Form durch Daten auf einer Festplatte repräsentiert wird, einen maschinenlesbaren Schek zu erzeugen. Den druckt man zum Verschenken aus. Der Empfänger reicht ihn bei seiner Bank ein, die die Daten liest und den Betrag einem Konto gutschreibt oder auszahlt. Im Prinzip kann das mit allen Zahlungssystemen funktionieren, die beim Bezahlen eine Online-Verbindung zur Bank erfordern und auf diese Weise Kopien erkennen.

Rechnungen Die Finanzämter tun sich schwer mit digitalen Rechnungen. Sie gelten nicht als Urkunde und werden deshalb nicht als Beleg anerkannt [Schw98]. Mit dem zunehmenden elektronischen Handel wird das zum Problem – Rechnungen müssen nachträglich mit der Post versandt werden.

Die Schwierigkeiten sind allerdings eher psychologischer Natur. Ein Fax erkennen die Finanzbehörden nämlich anstandslos an, obwohl es genauso leich zu fälschen ist wie eine im Netz übermittelte und beim Käufer gedruckte Rechnung.

Für den (langen?) Übergangszeitraum bis zur Akzeptanz neuer Techniken auch in den Finanzämtern bietet es sich daher an, Rechnungen für Online-Käufe ähnlich zu gestalten wie die hier vorgestellten Eintrittskarten. Neben einer für Menschen lesbaren Darstellung wird ein 2D-Symbol gedruckt, das die wesentlichen Rechnungsdaten zusammen mit einer digitalen Signatur des Händlers enthält. Die Signatur wird nach dem Signaturgesetz erstellt und kann bei Bedarf geprüft werden. Das ist nicht aufwendiger als die Prüfung einer herkömmlichen Unterschrift, für die sich das Amt immerhin ein Vergleichsmuster besorgen müßte, um Fälschungen erkennen zu können.

6.6 Schlußwort

Die Arbeit endet hier.

A Quelltexte der Sicherheitsschicht

Abgedruckt sind hier die Quelltexte der Sicherheitsschicht (`seclayer.h` und `seclayer.c`) zusammen mit dem Protokollmodul für IDEA-Verschlüselung (`idea.c`).

A.1 `seclayer.h`

```
/*
 * seclayer.h   header file for the security layer
 *
 * Best viewed with tab size 4.
 *
 * requires ticket.h, dblock.h
 */

#ifndef TCK_SECLAYER_H
#define TCK_SECLAYER_H

#include <sys/types.h>
#include <limits.h>


/* Key IDs are built from 2 bytes PID, 4 bytes creation time, and 2 counter
 * bytes. Wherever a key ID is needed, a memory buffer of TCK_KEYID_LEN
 * unsigned chars is used to store it.
 */
#define TCK_KEYID_LEN    8


/* The tck_keyinfo_t holds all metainformation available on a key. The
 * protocol member indicates the protocol type for which the key is to
 * be used. It corresponds to the pmod array in seclayer.c. The keytype
 * member is TCK_KEYTYPE_SECRET for all keys except for public keys of
 * asymmetric signature protocols.
 */
typedef struct {
    unsigned char protocol;
#       define TCK_FIRSTPROTO        0
#       define TCK_PROTO_IDEA_ECB    0
#       define TCK_PROTO_RSA_MD5     1
#       define TCK_PROTO_HMAC_MD5    2
#       define TCK_LASTPROTO         2

    unsigned char ptype;
#       define TCK_PTYPE_ASYM    1        /* asymmetric signature */
#       define TCK_PTYPE_SYM     2        /* symmetric encryption */
#       define TCK_PTYPE_MAC     3        /* symmetric message authentication */
#       define TCK_PTYPE_NONE    UCHAR_MAX

    unsigned char keytype;
#       define TCK_KEYTYPE_SECRET        1
#       define TCK_KEYTYPE_PUBLIC        2
#       define TCK_KEYTYPE_NONE          UCHAR_MAX

    time_t expire;                       /* second since 1970-01-01 00:00:00 */
    unsigned char keyid[TCK_KEYID_LEN];
} tck_keyinfo_t;


/* Function prototypes.
 */
extern int tck_slinit( unsigned char *keydbname, unsigned flags );
extern void tck_slclose( void );
extern tck_keyinfo_t * tck_genkey( unsigned char protocol, time_t expire );
extern int tck_delkey( const unsigned char *keyid );
extern tck_buffer_t * tck_exportkey( const unsigned char *exportid,
                                     const unsigned char *signid,
                                     const unsigned char *passwd );
extern tck_keyinfo_t * tck_importkey( const tck_buffer_t *impkey,
                                      const unsigned char *signid,
                                      const unsigned char *passwd );
extern tck_keyinfo_t ** tck_listkeys( void );
extern tck_buffer_t * tck_protectmsg( const tck_buffer_t *msg,
                                      const unsigned char *keyid );
extern tck_buffer_t * tck_unprotectmsg( const tck_buffer_t *pdu,
```

```
                                        const unsigned char *keyid );


/* The tck_proto_t structure is used internally in seclayer.c and the
 * protocol modules only. Each protocol module exports a tck_proto_t.
 * For each protocol valid number the pmod array in seclayer.c holds
 * a pointer to the corresponding tck_proto_t structure.
 */
typedef struct  {
    unsigned char ptype;            /* as defined in tck_keyinfo_t above */

    /* Interface to symmetric encryption modules. */
    struct {
        tck_buffer_t *(*genkey)( const unsigned char *passwd );
        tck_buffer_t *(*encrypt)( const tck_buffer_t *cleartext,
                                  const tck_buffer_t *key );
        tck_buffer_t *(*decrypt)( const tck_buffer_t *ciphertext,
                                  const tck_buffer_t *key );
    } sym;

    /* Interface to asymmetric signature modules. */
    struct {
        tck_buffer_t *(*genkey)( void );
        tck_buffer_t *(*sign)( const tck_buffer_t *inmsg,
                               const tck_buffer_t *keybuf );
        tck_buffer_t *(*verify)( const tck_buffer_t *inmsg,
                                 const tck_buffer_t *keybuf );
        tck_buffer_t *(*sec2pub)( const tck_buffer_t *seckey );
    } asym;

    /* Interface to symmetric message authentication modules. */
    struct  {
        tck_buffer_t *(*genkey)( const unsigned char *passwd );
        tck_buffer_t *(*auth)( const tck_buffer_t *plainmsg,
                               const tck_buffer_t *key );
        tck_buffer_t *(*verify)( const tck_buffer_t *authmsg,
                                 const tck_buffer_t *key );
    } mac;
} tck_proto_t;

#endif
```

A.2 seclayer.c

```
/*
 * seclayer.c   security layer and key database functions
 *
 * Best viewed with tab size 4.
 *
 * requires the dblock module
 *
 * The key database format is architecture-specific (as (N)DBM databases
 * are in whole) and thus not guaranteed to be portable across architectures.
 * However, everything that's visible to the outside, such as exported keys
 * or secured messages, should be usable anywhere where this module compiles
 * and runs without modification.
 */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <ndbm.h>
#include <assert.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/resource.h>
#include <time.h>
#include <netinet/in.h>          /* Yes, we need them all. */

#include <rand.h>                /* SSLeay header */

#include "ticket.h"
#include "msglayer.h"
#include "seclayer.h"
#include "dblock.h"


/* dbm's internal block size is 1024 */
#define DBMBLOCKSIZE 1024

#define PSCOMMAND "ps -cel"      /* that's for Solaris; used in seedrng() */


/* Each protocol module exports a tck_proto_t structure containing
 * pointers to its functions. The order in the pmod array corresponds
```

```
 * to the TCK_PROTO_* constants defined in seclayer.h. To add your own
 * module, simply add it here, give it the next spare number in
 * seclayer.h and increase TCK_LASTPROTO.
 */

extern tck_proto_t proto_idea_ecb;
extern tck_proto_t proto_rsa_md5;
extern tck_proto_t proto_hmac_md5;

tck_proto_t *pmod[TCK_LASTPROTO+1] = {
    &proto_idea_ecb,
    &proto_rsa_md5,
    &proto_hmac_md5
};


/* The EXPT_CRYPT_PROTO protocol is used when encrypting keys for export.
 * It must be a TCK_PTYPE_SYM protocol.
 *
 * Currently protocol 0 is IDEA in ECB mode since it is the only symmetric
 * cipher implemented. For key encryption one of the chaining modes should
 * be used, which could be done by adding an additional set of functions to
 * idea.c.
 */
#define EXPT_CRYPT_PROTO     0


/* Internal state information.
 */
static unsigned char keydbpath[PATH_MAX+1];
static DBM *keydb = NULL;
static unsigned globflags;


static void
newid( unsigned char *id )
/*
 * Returns a 64 bit ID string. The caller must provide a buffer of at
 * least TCK_KEYID_LEN bytes.
 *
 * A 64 bit ID is built from current time (in seconds from 1907-01-01
 * 00:00:00; 4 bytes on most older systems; will overflow in 2038),
 * process ID and a counter that is started once for each process. This
 * produces unique ID strings as long as the system clock works, a
 * process ID is not reused within one second of time and no process
 * calls this function more than 65.535 times within one second. ID
 * strings might repeat after 68 years.
 *
 * From the Solaris intro(2) man page:
 * A process ID may not be reused by the system until the process
 * lifetime, process group lifetime and session lifetime ends for
 * any process ID, process group ID and session ID equal to that
 * process ID.
 */
{
    static unsigned short count = 0;
    unsigned short pid;
    time_t sec;

    /* Never make assumptions. */
    assert( sizeof( unsigned short ) == 2 );
    assert( sizeof( time_t ) == 4 );
    assert( PID_MAX < USHRT_MAX );

    pid = (unsigned short) getpid();
    sec = time( NULL );
    count += 1;
    memcpy( id, &pid, 2 );
    memcpy( id + 2, &sec, 4 );
    memcpy( id + 6, &count, 2 );

    return;
}


static int
seedrng( unsigned flags )
/*
 * Gather some random bytes and seed the PRNG.
 * With some inspiration from Peter Gutmann's cryptlib
 *
 * This function is merely a rough sketch of random gathering. See
 * cryptlib source code for better (but slower) methods.
 *
 * If an Attacker has access to the machine where keys are generated,
 * he might be able to guess most of the information used to seed the
 * random number generator. But in that case he might be able to get
 * the key itself, too.
 *
 * Flags can be combined by logical or.
 *
 * When none of the flags is specified and thus the random number generator
```

```
 * would remain unseeded -1 is returned, 0 otherwise.
 */
#define FLG_RNDPS    1        /* process list */
#define FLG_RNDPID   2        /* process and parent process ID */
#define FLG_RNDRUS   4        /* ressource usage */
#define FLG_RNDTIME  8        /* current time */
#define FLG_RNDALL   16       /* all sources */
{
    FILE *pspipe;
    unsigned char psbuf[LINE_MAX];
    struct timeval tv;
    struct rusage ru;
    int i;

    if ( 0 == (flags & FLG_RNDALL) )
        return ( -1 );

    if ( 0 != (flags & FLG_RNDTIME) )    {
        /* add current time */
        gettimeofday( &tv, NULL );
        RAND_seed( (unsigned char *) &tv, sizeof( tv ) );
    }

    if ( 0 != (flags & FLG_RNDPS) ) {
        /* get a verbose list of processes currently running
         * For more random data, netstat -an, vmstat or who could be used
         */
        if ( (pspipe = popen( PSCOMMAND, "r" )) != NULL )
            while ( fgets( (char *) psbuf, LINE_MAX, pspipe) != NULL )   {
                RAND_seed( psbuf, strlen( psbuf ) );
                memset( psbuf, 0, LINE_MAX );
            }
        pclose( pspipe );
    }

    if ( 0 != (flags & FLG_RNDPID) )     {
        /* add process ID and parent process ID */
        i = getpid();
        RAND_seed( (unsigned char *) &i, sizeof( i ) );
        i = getppid();
        RAND_seed( (unsigned char *) &i, sizeof( i ) );
    }

    if ( 0 != (flags & FLG_RNDRUS) )     {
        /* add ressource usage information for current process */
        getrusage( RUSAGE_SELF, &ru );
        RAND_seed( (unsigned char *) &ru, sizeof( ru ) );
    }

    return ( 0 );
}


int tck_slinit( unsigned char *keydbname, unsigned flags )
/*
 * Open the NDBM database keydbname and perform various initialisation.
 * tck_slinit must be called prior to any other security layer operation.
 * Flags can be TCK_FLG_NONE for read-only access or TCK_FLG_RDWR for
 * read/write access. The latter is nessecary when keys are to be
 * generated, imported or deleted.
 *
 * Return value is 0 if no error occured and -1 in case of error.
 *
 * Under all circumstances the key database should be closed using
 * tck_slclose(), even after failed initialisation.
 */
{
    int rdwr;

    /* check parameters and internal state */
    if ( ((flags & ~TCK_FLG_RDWR) != 0) || (NULL == keydbname) )     {
        tck_errno = TCK_ERR_PARAM;
        return ( -1 );
    }
    if ( strlen( (char *) keydbname ) > PATH_MAX     )    {
        tck_errno = TCK_ERR_PARAM;
        return ( -1 );
    }

    if ( NULL != keydb )     {
        /* already open */
        tck_errno = TCK_ERR_ISOPEN;
        return ( -1 );
    }

    globflags = flags;
    rdwr = ((flags && TCK_FLG_RDWR) == 0) ? O_RDONLY : O_RDWR;

    /* open the key database */
```

```
    keydb = dbm_open( (char *) keydbname, rdwr|O_CREAT|O_SYNC, 0600 );
    if ( NULL == keydb )     {
        tck_errno = TCK_ERR_OPEN;
        return ( -1 );
    }
    strncpy( (char *) keydbpath, (char *) keydbname, PATH_MAX );
    keydbpath[PATH_MAX] = 0;

    /* seed random number generator */
    seedrng( FLG_RNDALL );

    return ( 0 );
}


void tck_slclose( void )
/*
 * Close the key database and reset internal state.
 */
{
    if ( NULL != keydb )
        dbm_close( keydb );
    keydb = NULL;
    memset( keydbpath, 0, PATH_MAX + 1 );
    globflags = 0;
    return;
}


static int
storekey( const tck_buffer_t *key,
          tck_keyinfo_t keyinfo )
/*
 * Stores a key and key information in a database opened by tck_slinit().
 * For internal use only.
 */
{
    datum dbk, dbd;
    unsigned char dbdbuf[DBMBLOCKSIZE];

    if ( NULL == keydb )     {
        tck_errno = TCK_ERR_NOTOPEN;
        return ( -1 );
    }

    dbm_clearerr( keydb );

    if ( tck_bempty( key ) || NULL == keyinfo.keyid )    {
        tck_errno = TCK_ERR_PARAM;
        return ( -1 );
    }

    /* make buffer with proto, expire, key
     *
     * format: 1 Byte protocol ID, 1 Byte key type, 4 bytes expiration date,
     *         rest is the key
     */
    if ( (key->len + 2 + sizeof( time_t ) + TCK_KEYID_LEN) > DBMBLOCKSIZE ) {
        tck_errno = TCK_ERR_UNSPEC;
        return ( -1 );
    }

    LOCKDB ( keydbpath, -1 );

    dbdbuf[0] = keyinfo.protocol;
    dbdbuf[1] = keyinfo.keytype;
    /* ptype does not have to be stored since it is available in pmod */
    memcpy( dbdbuf+2, &keyinfo.expire, sizeof( time_t ) );
    memcpy( (dbdbuf + 2 + sizeof( time_t )), key->buf, key->len );

    /* store in DBM database */
    dbd.dptr = dbdbuf;
    dbd.dsize = 2 + sizeof( time_t ) + key->len;

    dbk.dptr = (char *) keyinfo.keyid;
    dbk.dsize = TCK_KEYID_LEN;

    if ( 0 != dbm_store( keydb, dbk, dbd, DBM_INSERT ) )     {
        tck_errno = TCK_ERR_INSERT;
        UNLOCKRETURN( keydbpath, -1 );
    }

    UNLOCKRETURN( keydbpath, 0 );
}


static tck_buffer_t *
fetchkey( const unsigned char *keyid,
          tck_keyinfo_t *keyinfo )
/*
 * Fetches a key and key information from the database opened by tck_slinit().
```

```
 * For internal use only.
 */
{
    datum dbk, dbd;
    tck_buffer_t *ret;

    if ( NULL == keydb )     {
        tck_errno = TCK_ERR_NOTOPEN;
        return ( NULL );
    }

    if ( (NULL == keyid) )   {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }

    LOCKDB( keydbpath, NULL );

    /* retrieve key */
    dbm_clearerr( keydb );
    dbk.dptr = keyid;
    dbk.dsize = TCK_KEYID_LEN;
    dbd = dbm_fetch( keydb, dbk );
    if ( (dbm_error( keydb ) != 0) || (NULL == dbd.dptr)
         || (0 == dbd.dsize) )   {
        tck_errno = TCK_ERR_FETCH;
        UNLOCKRETURN(keydbpath, NULL );
    }

    unlockdb( keydbpath );

    /* buffer format as defined in storekey()
     */

    /* allocate memory and copy key */
    ret = tck_bmalloc( dbd.dsize - 6 );
    if ( NULL == ret )   {
        tck_errno = TCK_ERR_MEM;
        return ( NULL );
    }
    ret->len = dbd.dsize - 2 - sizeof( time_t );
    memcpy( ret->buf, (dbd.dptr + 2 + sizeof( time_t )),
            (dbd.dsize - 2 - sizeof( time_t )) );

    if ( NULL != keyinfo )   {
        keyinfo->protocol = dbd.dptr[0];
        keyinfo->keytype = dbd.dptr[1];
        keyinfo->ptype = pmod[keyinfo->protocol]->ptype;
        memcpy( &keyinfo->expire, (dbd.dptr + 2), sizeof( keyinfo->expire ) );
        memcpy( &keyinfo->keyid, keyid, TCK_KEYID_LEN );
    }

    return ( ret );

}


tck_keyinfo_t *
tck_genkey( unsigned char protocol, time_t expire )
/*
 * Generate a key suitable for the protocol specified. The newly generated
 * key is stored in the key database and a tck_keyinfo_t structure is
 * returned. The key (or its public components for asymmetric ciphers) can
 * then be retrieved using tck_expkey(). Protocol modules are responsible
 * for providing the generated key in a linear buffer that can be stored
 * and transported to other machines (possibly of different architecture).
 *
 * The expire parameter might contain any value of time in seconds since
 * 00:00:00 UTC, January 1, 1970. After expiration the key can no longer
 * be exported or used. If expire lies in the past, a key is generated
 * anyway, but expires immediately when created. (See
 * http://fsinfo.cs.uni-sb.de/~hirvi/ruesch/ for possible aplications of
 * this feature.)
 * As for Solaris, the time_t type currently is defined as long integer.
 * For a 'never-expiring' key, set expire to LONG_MAX. That's a date in
 * January 2038 for current time_t.
 *
 * In case of error, NULL is returned and tck_errno is set to an error
 * code. If a non-NULL pointer is returned, it points into static memory
 * that is reused on next call to tck_genkey().
 */
{
    tck_buffer_t *newkey = NULL;
    static tck_keyinfo_t keyinfo;

    /* check parameters and internal state */
    if ( time( NULL ) > expire )     {
        tck_errno = TCK_ERR_EXPIRE;
        return ( NULL );
    }
    if ( (TCK_FLG_RDWR & globflags) == 0 )   {
```

```
        /* cannot create key if database is read only */
        tck_errno = TCK_ERR_RDONLY;
        return ( NULL );
    }
    if ( TCK_LASTPROTO < protocol ) {
        tck_errno = TCK_ERR_NOTSUP;
        return ( NULL );
    }

    /* call key generator of specified protocol */
    switch ( pmod[protocol]->ptype )     {
    case TCK_PTYPE_ASYM:
        newkey = (*pmod[protocol]->asym.genkey)();
        break;
    case TCK_PTYPE_SYM:
        newkey = (*pmod[protocol]->sym.genkey)( NULL );
        break;
    case TCK_PTYPE_MAC:
        newkey = (*pmod[protocol]->mac.genkey)( NULL );
        break;
    default:
        tck_errno = TCK_ERR_UNSPEC;
        newkey = NULL;
    }

    if ( NULL == newkey )
        return ( NULL );              /* tck_errno set  by genkey function */


    /* generate key ID and store key in database */
    newid( keyinfo.keyid );
    keyinfo.protocol = protocol;
    keyinfo.ptype = pmod[protocol]->ptype;
    keyinfo.keytype = TCK_KEYTYPE_SECRET;
    keyinfo.expire = expire;
    if ( storekey( newkey, keyinfo )
         != 0 ) {
        tck_bfree( newkey );
        /* tck_errno set by storekey() */
        return ( NULL );
    }

    return ( &keyinfo );

}


int tck_delkey( const unsigned char *keyid )
/*
 * Deletes a key from the database.
 * 0 is returned If the key was successfully deleted. In case of error
 * -1 is returned and tck_errno ist set to an error code.
 */
{
    datum dbk;

    /* check parameters and internal state */
    if ( NULL == keydb )     {
        tck_errno = TCK_ERR_NOTOPEN;
        return ( -1 );
    }
    if ( NULL == keyid )     {
        tck_errno = TCK_ERR_PARAM;
        return ( -1 );
    }
    if ( (TCK_FLG_RDWR & globflags) == 0 )  {
        /* cannot delete key if database is read only */
        tck_errno = TCK_ERR_RDONLY;
        return ( -1 );
    }

    LOCKDB( keydbpath, -1 );

    dbk.dsize = TCK_KEYID_LEN;
    dbk.dptr = keyid;

    if ( 0 > dbm_delete( keydb, dbk ) ) {
        tck_errno = TCK_ERR_DELETE;
        UNLOCKRETURN( keydbpath, -1 );
    }

    UNLOCKRETURN( keydbpath, 0 );
}


tck_buffer_t *
tck_exportkey( const unsigned char *exportid,
               const unsigned char *signid,
               const unsigned char *passwd )
/*
 * Export a key from the database. For public key modules, only the
```

```
 * public parts of the key are exported.
 *
 * The second parameter can be NULL or a pointer to the ID of a private
 * key stored in the database. If it is not NULL, this key is used to
 * sign the exported key.
 *
 * The third parameter can be NULL or a pointer to a zero terminated string
 * containing a password which is used to encrypt the key to export.
 *
 * After retrieving the key, a header containing protocol (1 byte), expiration
 * date (4 bytes) and key ID (8 bytes) is added. If signid != NULL, the key
 * is signed. Then, if requested (i.e. passwd != NULL), these data are then
 * encrypted. [MPW92] states that it is important to first sign and then
 * encrypt, so don't change this order if you don't know what you do.
 *
 * The returned buffer must be released by the caller using tck_bfree().
 * In case of error a NULL pointer is returned and tck_errno is set to
 * error code.
 */
{
    tck_buffer_t *signkey;        /* signature key from database */
    tck_buffer_t *enckey;         /* encryption key derived from passwd */
    tck_buffer_t *fetchbuf;       /* key to export as fetched from database */
    tck_buffer_t *expbuf;         /* components of fetchbuf to be exported */
    tck_buffer_t *headbuf;        /* key to export with header */
    tck_buffer_t *signbuf;        /* the signed key before encryption */
    tck_buffer_t *retbuf;         /* exported key, possibly crypted & signed */
    time_t curtime;
    tck_keyinfo_t expkinfo, sigkinfo;

    if ( NULL == exportid ) {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }

    curtime = time( NULL );

    fetchbuf = fetchkey( exportid, &expkinfo );
    if ( NULL == fetchbuf )
        return ( NULL );

    if ( ( 0 >= expkinfo.expire ) || ( curtime >= expkinfo.expire ) )    {
        tck_errno = TCK_ERR_EXPIRE;
        tck_bfree( fetchbuf );
        return ( NULL );
    }

    if ( TCK_PTYPE_ASYM == pmod[expkinfo.protocol]->ptype ) {
        /* it's some asymmetric cryptosystem - extract public components
         * of key
         */
        expbuf = pmod[expkinfo.protocol]->asym.sec2pub( fetchbuf );
        tck_bfree( fetchbuf );
        if ( NULL == expbuf )
            return ( NULL );
    }
    else
        /* for MAC and symmetric encryption there are no public components
         * to extract
         */
        expbuf = fetchbuf;


    /* Now we have the components to export in *expbuf. Add protocol ID,
     * expiration date and key ID.
     * format: 1 byte protocol, 4 bytes expiration date, 8 bytes key id
     */
    headbuf = tck_bmalloc(
                    expbuf->len + 1 + sizeof( time_t ) + TCK_KEYID_LEN );
    if ( NULL == headbuf )  {
        tck_errno = TCK_ERR_MEM;
        tck_bfree( expbuf );
        return ( NULL );
    }
    headbuf->len = expbuf->len + 1 + sizeof( time_t ) + TCK_KEYID_LEN;
    headbuf->buf[0] = expkinfo.protocol;
    assert( sizeof( time_t ) == 4 );
    expkinfo.expire = htonl( expkinfo.expire );
    memcpy( (headbuf->buf + 1), &expkinfo.expire, sizeof( expkinfo.expire ) );
    expkinfo.expire = ntohl( expkinfo.expire );
    memcpy( (headbuf->buf + 5), exportid, TCK_KEYID_LEN );
    memcpy( (headbuf->buf + 5 + TCK_KEYID_LEN), expbuf->buf, expbuf->len );
    tck_bfree( expbuf );

    /* sign if requested */
    if ( NULL != signid )   {
        signkey = fetchkey( signid, &sigkinfo );
        if ( NULL == signkey )  {
            tck_bfree( headbuf );
            return ( NULL );
```

```
        }

        /* check for expiration and correct protocol/keytype */
        if ( ( 0 >= sigkinfo.expire ) || ( curtime >= sigkinfo.expire )
             || ( TCK_PTYPE_ASYM != sigkinfo.ptype)
             || ( TCK_KEYTYPE_SECRET != sigkinfo.keytype ) )   {
            tck_errno = TCK_ERR_KEYTYPE;
            tck_bfree( headbuf );
            tck_bfree( signkey );
            return ( NULL );
        }

        signbuf = (*pmod[sigkinfo.protocol]->asym.sign)( headbuf, signkey );
        tck_bfree( headbuf );
        tck_bfree( signkey );
        if ( NULL == signbuf )
            return ( NULL );
    }
    else
        signbuf = headbuf;

    /* encrypt if requested */
    if ( NULL != passwd )    {
        /* derive key from passwd and encrypt key */
        enckey = (*pmod[EXPT_CRYPT_PROTO]->sym.genkey)( passwd );
        /* encrypt() should complain when called with empty key */
        retbuf = (*pmod[EXPT_CRYPT_PROTO]->sym.encrypt)( signbuf, enckey );
        tck_bfree( enckey );
        tck_bfree( signbuf );
        if ( NULL == retbuf )
            return ( NULL );
    }
    else
        retbuf = signbuf;

    /* No information on wether the key is signed or encrypted or on
     * the key ID used for signing is transmitted with the key. The
     * tck_importkey() function completely relies on proper information
     * provided from the application.
     * Sounds stupid, but it isn't, since there is no way to prevent an
     * attacker from altering that information, which he could use to
     * import a key of his own marked as unsigned and unencrypted into
     * the checkpoints database. Proof left to the attacker.
     */

    return ( retbuf );

}


tck_keyinfo_t *
tck_importkey( const tck_buffer_t *impkey,
               const unsigned char *signid,
               const unsigned char *passwd )
/*
 * Imports a key that was exported from another key database using
 * tck_exportkey().
 *
 * The return value points into static memory that will be reused with
 * the next call to tck_importkey(). In case of error a NULL pointer is
 * returned and tck_errno is set to error code.
 *
 * signid and passwd work as described for tck_exportkey(). The caller
 * must provide proper signature key ID and password information.
 */
{
    tck_buffer_t *vrfykey;          /* signature verification key */
    tck_buffer_t *deckey;           /* decryption key */
    tck_buffer_t *decbuf;           /* decrypted but still signed key */
    tck_buffer_t *keybuf;           /* key + header after sig. verification */
    tck_buffer_t *strbuf;           /* storage buffer */
    static tck_keyinfo_t impkinfo;  /* returned on success */
    tck_keyinfo_t sigkinfo;         /* verification key information */
    int strres;

    if ( tck_bempty( impkey ) ) {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }

    if ( NULL != passwd )    {
        /* derive key from passphrase */
        deckey = (*pmod[EXPT_CRYPT_PROTO]->sym.genkey)( passwd );
        if ( NULL == deckey )
            return ( NULL );

        /* decrypt */
        decbuf = (*pmod[EXPT_CRYPT_PROTO]->sym.decrypt)( impkey, deckey );
        tck_bfree( deckey );
        if ( NULL == decbuf )
```

```
            return ( NULL );
    }
    else    {
        /* copy import buffer only to be able to tck_bfree() it later */
        decbuf = tck_bmalloc( impkey->len );
        if ( NULL == decbuf )   {
            tck_errno = TCK_ERR_MEM;
            return ( NULL );
        }
        decbuf->len = impkey->len;
        memcpy( decbuf->buf, impkey->buf, impkey->len );
    }


    if ( NULL != signid )   {
        /* retrieve key for signature verification */
        vrfykey = fetchkey( signid, &sigkinfo );
        if ( NULL == vrfykey )
            return ( NULL );
        if ( time( NULL ) > sigkinfo.expire )   {
            tck_errno = TCK_ERR_EXPIRE;
            tck_bfree( vrfykey );
            tck_bfree( decbuf );
            return ( NULL );
        }
        if ( TCK_PTYPE_ASYM != sigkinfo.ptype ) {
            tck_errno = TCK_ERR_PARAM;
            tck_bfree( vrfykey );
            tck_bfree( decbuf );
            return ( NULL );
        }

        /* check and remove signature */
        keybuf = (*pmod[sigkinfo.protocol]->asym.verify)( decbuf, vrfykey );
        tck_bfree( vrfykey );
        tck_bfree( decbuf );
        if ( NULL == keybuf )   {
            /* verify() function must provide TCK_ERR_SIG in tck_errno if
             * verification failed or other code when an error occured.
             */
            return ( NULL );
        }
    }
    else    {
        keybuf = decbuf;
    }


    /* get protocol, expiration date and key ID */
    impkinfo.protocol = keybuf->buf[0];
    if ( TCK_LASTPROTO < impkinfo.protocol )    {
        tck_errno = TCK_ERR_NOTSUP;
        tck_bfree( keybuf );
        return ( NULL );
    }
    if ( TCK_PTYPE_ASYM == pmod[impkinfo.protocol]->ptype )
        impkinfo.keytype = TCK_KEYTYPE_PUBLIC;
    else
        impkinfo.keytype = TCK_KEYTYPE_SECRET;

    memcpy( &impkinfo.expire, (keybuf->buf + 1), sizeof( impkinfo.expire ) );
    impkinfo.expire = ntohl( impkinfo.expire );
    if ( time( NULL ) > impkinfo.expire )   {
        tck_errno = TCK_ERR_EXPIRE;
        tck_bfree( keybuf );
        return ( NULL );
    }

    memcpy( impkinfo.keyid, (keybuf->buf + 1 + sizeof( time_t )),
            TCK_KEYID_LEN );


    /* copy the key to strbuf for storage
     */
    strbuf = tck_bmalloc( keybuf->len - 1 - sizeof( time_t ) - TCK_KEYID_LEN );
    if ( NULL == strbuf )   {
        tck_errno = TCK_ERR_MEM;
        tck_bfree( keybuf );
        return ( NULL );
    }
    strbuf->len = strbuf->buflen;
    memcpy( strbuf->buf, (keybuf->buf + 1 + sizeof( time_t ) + TCK_KEYID_LEN),
            strbuf->buflen );
    tck_bfree( keybuf );

    /* store key in database */
    strres = storekey( strbuf, impkinfo );
    tck_bfree( strbuf );
```

```
    if ( 0 != strres )
        return ( NULL );
    else
        return ( &impkinfo );

}


tck_keyinfo_t **
tck_listkeys( void )
/*
 * Returns a list of all keys in the database.
 *
 * The return value is an array of pointers to tck_keyinfo_t structures.
 * The last member of the array is set to NULL.
 *
 * The caller must release allocated memory using tck_freeklst() when
 * the list is no longer needed.
 *
 * The database is not locked here since the first call to the fetchkey
 * function would remove the lock. No, I don't have a solution.
 */
{
    tck_keyinfo_t **ret;
    datum dbk;
    int keycnt = 0;
    int i = 0;
    tck_buffer_t *tmpkey;

    if ( NULL == keydb )     {
        tck_errno = TCK_ERR_NOTOPEN;
        return ( NULL );
    }

    /* first count the keys in database */
    for ( dbk = dbm_firstkey( keydb ); dbk.dptr != NULL;
          dbk = dbm_nextkey( keydb ) )
        keycnt += 1;

    /* allocate memory for (keycnt + 1) pointers */
    ret = malloc( (keycnt + 1) * sizeof( tck_keyinfo_t * ) );
    if ( NULL == ret )  {
        tck_errno = TCK_ERR_MEM;
        return ( NULL );
    }
    memset( ret, 0, (keycnt + 1) * sizeof( tck_keyinfo_t * ) );

    /* now go through the database again and collect information for
     * each key
     */
    keycnt = 0;
    for ( dbk = dbm_firstkey( keydb ); dbk.dptr != NULL;
          dbk = dbm_nextkey( keydb ), keycnt++ )   {
        ret[keycnt] = malloc( sizeof( tck_keyinfo_t ) );
        tmpkey = fetchkey( dbk.dptr, ret[keycnt] );
        if ( (NULL == ret[keycnt]) || (NULL == tmpkey) )     {
            for ( i = 0; i < keycnt-1; i++ )
                free( ret[i] );
            free( ret );
            tck_errno = TCK_ERR_MEM;
            return ( NULL );
        }

        /* dbk.dptr points to the key ID */

        ret[keycnt + 1] = NULL;
        tck_bfree( tmpkey );
    }

    return ( ret );

}


void
tck_freeklst( tck_keyinfo_t **list )
/*
 * Releases a list of key information that was previously created by
 * tck_listkeys()
 */
{
    int i = 0;

    if ( NULL == list ) return;
    for ( i = 0; NULL != list[i]; i++ )
        free( list[i] );
    free( list );
}


tck_buffer_t *
```

```
tck_protectmsg( const tck_buffer_t *msg,
                const unsigned char *keyid )
/*
 * Protect a message using the key specified by keyid. Since every key
 * has one specific protocol associated with it, the protocol to use is
 * implicitly given with the key ID.
 *
 * The protocol modules are responsible for providing encrypted/signed/
 * authenticated/whatever data in a format that is portable to different
 * architectures.
 *
 * The returned buffer must be released by the caller using tck_bfree().
 * In case of error a NULL pointer is returned and tck_errno is set.
 */
{
    tck_buffer_t *key;
    tck_buffer_t *pdu;
    tck_keyinfo_t keyinfo;

    if ( tck_bempty( msg ) || ( NULL == keyid ) ) {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }

    /* fetch key from database and check for expiration */
    key = fetchkey( keyid, &keyinfo );
    if ( NULL == key )
        return ( NULL );
    if ( time( NULL ) > keyinfo.expire )     {
        tck_errno = TCK_ERR_EXPIRE;
        tck_bfree( key );
        return ( NULL );
    }

    /* sign/encrypt/authenticate */
    switch ( pmod[keyinfo.protocol]->ptype )     {
    case TCK_PTYPE_ASYM:
        pdu = (*pmod[keyinfo.protocol]->asym.sign)( msg, key );
        break;
    case TCK_PTYPE_SYM:
        pdu = (*pmod[keyinfo.protocol]->sym.encrypt)( msg, key );
        break;
    case TCK_PTYPE_MAC:
        pdu = (*pmod[keyinfo.protocol]->mac.auth)( msg, key );
        break;
    default:
        tck_errno = TCK_ERR_UNSPEC;
        pdu = NULL;
    }

    tck_bfree( key );

    return ( pdu );      /* NULL if any error occured */

}


tck_buffer_t *
tck_unprotectmsg( const tck_buffer_t *pdu,
                  const unsigned char *keyid )
/*
 * tck_unprotectmsg() is the counterpart of tck_protectmsg(). It takes a
 * protocol data unit created by tck_protectmsg() and applies the specified
 * key on it. That means that signatures and MACs are checked and removed
 * and encrypted data are decrypted depending on the protocol type associated
 * with key *keyid. In case of success this results in the message that was
 * put into tck_protectmsg() before. In case of error a NULL pointer is
 * returned if the error is detectable. With symmetric encryption
 * protocols it might happen that a message is returned but consists of
 * useless crap if the wrong key was applied to the PDU. (Actually even
 * then most errors are detectable due to incorrect padding.)
 *
 * The returned buffer must be released by the caller using tck_bfree().
 */
{
    tck_buffer_t *key;
    tck_buffer_t *msg;
    tck_keyinfo_t keyinfo;

    if ( tck_bempty( pdu ) || ( NULL == keyid ) ) {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }

    /* fetch key from database */
    key = fetchkey( keyid, &keyinfo );
    if ( NULL == key )
        return ( NULL );
    if ( time( NULL ) > keyinfo.expire )     {
        tck_errno = TCK_ERR_EXPIRE;
        tck_bfree( key );
```

```
        return ( NULL );
    }

    /* decrypt/verify */
    switch ( pmod[keyinfo.protocol]->ptype )     {
    case TCK_PTYPE_ASYM:
        msg = (*pmod[keyinfo.protocol]->asym.verify)( pdu, key );
        break;
    case TCK_PTYPE_SYM:
        msg = (*pmod[keyinfo.protocol]->sym.decrypt)( pdu, key );
        break;
    case TCK_PTYPE_MAC:
        msg = (*pmod[keyinfo.protocol]->mac.verify)( pdu, key );
        break;
    default:
        tck_errno = TCK_ERR_UNSPEC;
        msg = NULL;
    }

    return ( msg );
}
```

A.3 idea.c

```
/*
 * idea.c
 *
 * Best viewed with tab size 4.
 *
 * Interface to SSLeay's implementation of the IDEA algorithm.
 *
 * This module depends on a properly initialised random number generator.
 */

#include <evp.h>            /* SSLeay headers */
#include <err.h>
#include <rand.h>
#include "ticket.h"
#include "seclayer.h"

#define IDEA_KEYLEN 16


static tck_buffer_t *
genkey_idea_ecb( const unsigned char *passwd )
/*
 * Returns a 128 bit IDEA key either derived from passwd string or,
 * if passwd is absent, a random key. If passwd != NULL, it is expected
 * to point to a non-empty zero terminated string.
 * The returned tck_buffer_t is tck_bmalloc()ed and must be released by
 * the caller using tck_bfree(). In case of error a NULL pointer is
 * returned and tck_errno is set to an error code.
 */
{
    static tck_buffer_t *newkey;

    if ( NULL != passwd )    {
        if ( strlen( (char *) passwd ) == 0 )    {
            tck_errno = TCK_ERR_PARAM;
            return ( NULL );
        }
    }

    ERR_clear_error();

    /* allocate buffer for key */
    newkey = tck_bmalloc( IDEA_KEYLEN );
    if ( NULL == newkey )    {
        tck_errno = TCK_ERR_MEM;
        return ( NULL );
    }
    newkey->len = IDEA_KEYLEN;

    if ( NULL == passwd )    {
        /* generate random key */
        RAND_bytes( newkey->buf, IDEA_KEYLEN );
    }
    else     {
        /* derive key from passwd using MD5 */
        EVP_BytesToKey( EVP_idea_ecb(), EVP_md5(), NULL, passwd,
                        strlen( (char *) passwd ), 1, newkey->buf, NULL );
    }

    if ( ERR_get_error() != 0 ) {
        tck_errno = TCK_ERR_CRYPT;
        tck_bfree( newkey );
        return( NULL );
    }
    else
```

```
        return ( newkey );
}


static tck_buffer_t *
encrypt_idea_ecb( const tck_buffer_t *cleartext,
                  const tck_buffer_t *key )
/*
 * Encrypt cleartext with key using IDEA algorithm in electronic codebook
 * mode.
 * The returned buffer is tck_bmalloc()-ed and must be tck_bfree()-ed by
 * the caller.
 */
{
    EVP_CIPHER_CTX ectx;
    tck_buffer_t *ciphertext;
    int finlen;

    /* parameters ok? */
    if ( tck_bempty( cleartext ) || tck_bempty( key ) ) {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }
    if ( IDEA_KEYLEN != key->len )  {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }

    ERR_clear_error();

    /* Ciphertext might be longer than cleartext if last block is incomplete.
     */
    ciphertext = tck_bmalloc( cleartext->len + 8 );
    if ( NULL == ciphertext )   {
        tck_errno = TCK_ERR_MEM;
        return ( NULL );
    }

    EVP_EncryptInit( &ectx, EVP_idea_ecb(), key->buf, NULL );
    EVP_EncryptUpdate( &ectx, ciphertext->buf, &ciphertext->len,
                       cleartext->buf, cleartext->len );
    EVP_EncryptFinal( &ectx, (ciphertext->buf + ciphertext->len), &finlen );
    ciphertext->len += finlen;

    /* zero *ectx, because it contains the encryption key */
    memset( ectx, 0, sizeof( EVP_CIPHER_CTX ) );

    if ( ERR_get_error() != 0 ) {
        tck_errno = TCK_ERR_CRYPT;
        tck_bfree( ciphertext );
        return ( NULL );
    }
    else
        return ( ciphertext );
}


static tck_buffer_t *
decrypt_idea_ecb( const tck_buffer_t *ciphertext,
                  const tck_buffer_t *key )
/*
 * Decrypt cleartext with key using IDEA algorithm in electronic codebook
 * mode.
 * The returned buffer is tck_bmalloc()-ed and must be tck_bfree()-ed by
 * the caller. In case of error a NULL pointer is returned and tck_errno
 * is set to en error code as defined in ticket.h.
 */
{
    EVP_CIPHER_CTX dctx;
    tck_buffer_t *cleartext;
    int finlen;

    if ( tck_bempty( ciphertext ) || tck_bempty( key ) )    {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }
    if ( IDEA_KEYLEN != key->len )  {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }

    ERR_clear_error();

    cleartext = tck_bmalloc( ciphertext->len );
    if ( NULL == ciphertext )   {
        tck_errno = TCK_ERR_MEM;
        return ( NULL );
    }

    EVP_DecryptInit( &dctx, EVP_idea_ecb(), key->buf, NULL );
```

```
    EVP_DecryptUpdate( &dctx, cleartext->buf, &cleartext->len,
                       ciphertext->buf, ciphertext->len );
    EVP_DecryptFinal( &dctx, (cleartext->buf + cleartext->len), &finlen );
    cleartext->len += finlen;

    /* zero *dctx, it contains the encryption key */
    memset( dctx, 0, sizeof( EVP_CIPHER_CTX ) );

    if ( ERR_get_error() != 0 ) {
        tck_errno = TCK_ERR_CRYPT;
        tck_bfree( cleartext );
        return( NULL );
    }
    else
        return ( cleartext );
}


/* Protocol definition. Functions are accessed by seclayer.c through these
 * pointers.
 */
const tck_proto_t proto_idea_ecb = {
    TCK_PTYPE_SYM,
    { genkey_idea_ecb, encrypt_idea_ecb, decrypt_idea_ecb },
    { NULL, NULL, NULL, NULL },
    { NULL, NULL, NULL }
};
```

B Quelltexte der Nachrichtenschicht

B.1 msglayer.h

```
/*
 * msglayer.h   message layer header file
 *
 * Best viewed with tab size 4.
 *
 * requires seclayer.h
 */

#ifndef TCK_MSGLAYER_H
#define TCK_MSGLAYER_H

#include <sys/types.h>


/* fixed length for all strings in ticket */
#define TCK_STRLEN  64


/* The tck_tparam_t type collects all information needed to create a
 * ticket, including informational text that is only to be printed
 * but not encoded in the machine-readable part.
 */
typedef struct {
    /* The fields member indicates which other members are
     * actually used in the structure.
     */
    unsigned short  fields;
#       define TCK_TICK_NR          32768
#       define TCK_LOC_NR           16384
#       define TCK_LOC_TEXT          8192
#       define TCK_EVENT_NR          4096
#       define TCK_EVENT_TEXT        2048
#       define TCK_BEGIN             1024
#       define TCK_DURATION           512
#       define TCK_INLET              256
#       define TCK_SEAT               128
#       define TCK_ROW                 64
#       define TCK_CATEGORY            32
#       define TCK_PRICE               16

#       define TCK_TEXT1                8
#       define TCK_TEXT2                4
#       define TCK_TEXT3                2

    unsigned long   ticket_nr;
    unsigned short  location_nr;
    unsigned char   location_text[TCK_STRLEN];
    unsigned short  event_nr;
    unsigned char   event_text[TCK_STRLEN];
    time_t          begin;              /* seconds since 1970-01-01 00:00:00 */
    unsigned short  duration;           /* seconds */
    unsigned short  inlet;              /* sconds before begin */
    unsigned short  seat;
    unsigned short  row;
    unsigned char   category;           /* price or seat */
    unsigned short  price;              /* *0.01 Euro */

    /* additional info not to be included in barcode */
    unsigned char   text1[TCK_STRLEN];
    unsigned char   text2[TCK_STRLEN];
    unsigned char   text3[TCK_STRLEN];

    /* local management information */
    time_t          db_expire;
} tck_tparam_t;


/* There are four different message types.
 */
#define TCK_MSGTYPE_SIMPLE1     1
#define TCK_MSGTYPE_SIMPLE2     2
#define TCK_MSGTYPE_STD         3
#define TCK_MSGTYPE_NOTEXT      4


/* prototypes
 */
```

```
extern tck_buffer_t * tck_param2msg( tck_tparam_t *param,
                                     unsigned msgtype );
extern tck_tparam_t * tck_msg2param( const tck_buffer_t *msg,
                                     unsigned msgtype );
extern unsigned long * tck_nextnum( unsigned char *countfile );
extern int tck_valid_simple1( const tck_tparam_t *param,
                              unsigned short event,
                              unsigned char *categories );
extern int tck_valid_simple2( const tck_tparam_t *param,
                              unsigned short hour, unsigned short minute,
                              unsigned short second );
extern int tck_valid_std( const tck_tparam_t *param, unsigned short location,
                          unsigned short event, unsigned char *categories,
                          int timecheck );

#endif
```

B.2 msglayer.c

```
/*
 * msglayer.c    message layer
 *
 * Best viewed with tab size 4.
 *
 * The message layer provides a ticket parameter structure and means for
 * encoding it into a message, decoding it from a message, and validating
 * parameters against several requirements.
 * For creation of a ticket first a tck_tparam_t structure must be filled
 * in by rthe application with appropriate date. Then, these parameters
 * are encoded to a message which can be processed by the security layer.
 * On the checkpoint's side, the message revoked by the security layer
 * is decoded into a tck_tparam_t structure which then can be checked
 * against parameters using the tck_valid_* functions.
 */

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/uio.h>
#include <netinet/in.h>
#include <assert.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>
#ifdef LINUX
#   include <sys/file.h>
#endif   /* LINUX */

#include "ticket.h"
#include "seclayer.h"
#include "msglayer.h"


static tck_buffer_t *
param2msg_simple1( const tck_tparam_t *param )
/*
 * Message format: 4 bytes ticket_nr, 2 bytes event_nr, 1 byte
 * price_category. All integers are in network byte order.
 *
 * This message type could be used for instance in a museum.
 */
{
    tck_buffer_t *ret;
    unsigned long ticketnr;
    unsigned short eventnr;

    assert( sizeof( unsigned long ) == 4 );
    assert( sizeof( unsigned short ) == 2 );

    if ( NULL == param )     {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }

    /* are all required fields there? */
    if ( ((TCK_TICK_NR | TCK_EVENT_NR | TCK_CATEGORY) & param->fields)
         != (TCK_TICK_NR | TCK_EVENT_NR | TCK_CATEGORY) )     {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }

    ret = tck_bmalloc( 7 );
    if ( NULL == ret )   {
        tck_errno = TCK_ERR_MEM;
        return ( NULL );
    }

    /* fill the buffer */
```

```
    ticketnr = param->ticket_nr;
    ticketnr = htonl( ticketnr );
    memcpy( ret->buf, &ticketnr, 4 );

    eventnr = param->event_nr;
    eventnr = htons( eventnr );
    memcpy( (ret->buf + 4), &eventnr, 2 );

    ret->buf[6] = param->category;

    ret->len = 7;

    return ( ret );

}


static tck_tparam_t *
msg2param_simple1( const tck_buffer_t *msg )
/*
 * Message format as defined for param2msg_simple1.
 *
 * The pointer returned will be overwritten on next call to
 * tck_msg2param_simple1().
 */
{
    static tck_tparam_t ret;

    assert( sizeof( unsigned long ) == 4 );
    assert( sizeof( unsigned short ) == 2 );

    /* check parameter */
    if ( NULL == msg )   {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }
    if ( 7 != msg->len )     {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }

    memset( &ret, 0, sizeof( tck_tparam_t ) );
    ret.fields = TCK_TICK_NR | TCK_EVENT_NR | TCK_CATEGORY;
    ret.category = msg->buf[6];

    memcpy( &ret.ticket_nr, msg->buf, 4 );
    ret.ticket_nr = ntohl( ret.ticket_nr );

    memcpy( &ret.event_nr, (msg->buf + 4), 2 );
    ret.event_nr = ntohs( ret.event_nr );

    return ( &ret );

}


int
tck_valid_simple1( const tck_tparam_t *param,
                   unsigned short event,
                   unsigned char *categories )
/*
 * Check ticket parameter against event number and categories.
 *
 * If *param meets all requirements, 1 is returned. If it does not, the
 * return value is 0. In case of error -1 is returned and tck_errno set
 * to an error code.
 */
{
    unsigned int i;
    bool catok = FALSE;;

    if ( (NULL == param) || (NULL == categories) )   {
        tck_errno = TCK_ERR_PARAM;
        return ( -1 );
    }
    if ( ((TCK_EVENT_NR & param->fields) == 0)
         || ((TCK_CATEGORY & param->fields) == 0) )  {
        tck_errno = TCK_ERR_PARAM;
        return ( -1 );
    }

    for ( i=0; categories[i] != 0; i++ )     {
        if ( param->category == categories[i] )  {
            catok = TRUE;
            break;
        }
    }

    if ( !catok )    {
        tck_errno = TCK_ERR_CATEG;
        return ( 0 );
```

```
    }
    if ( event != param->event_nr ) {
        tck_errno = TCK_ERR_EVENT;
        return ( 0 );
    }

    return ( 1 );

}


static tck_buffer_t *
param2msg_simple2( const tck_tparam_t *param )
/*
 * Message format: 4 bytes ticket_nr, 4 bytes begin time. All integers are
 * in network byteorder.
 *
 * Ticket number should be shorter and price category be included.
 */
{
    tck_buffer_t *ret;
    unsigned long ticketnr;
    time_t begin;

    assert( sizeof( unsigned long ) == 4 );
    assert( sizeof( time_t ) == 4 );

    if ( NULL == param )     {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }

    /* are all required fields there? */
    if ( ((TCK_TICK_NR | TCK_BEGIN) & param->fields )
         != (TCK_TICK_NR | TCK_BEGIN) ) {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }

    ret = tck_bmalloc( 8 );
    if ( NULL == ret )   {
        tck_errno = TCK_ERR_MEM;
        return ( NULL );
    }
    ret->len = 8;

    /* fill the buffer */
    ticketnr = param->ticket_nr;
    ticketnr = htonl( ticketnr );
    memcpy( ret->buf, &ticketnr, 4 );

    begin = param->begin;
    begin = htonl( begin );
    memcpy( (ret->buf + 4), &begin, 4 );

    return ( ret );
}


static tck_tparam_t *
msg2param_simple2( const tck_buffer_t *msg )
/*
 * Message format as defined for param2msg_simple2().
 */
{
    static tck_tparam_t ret;

    assert( sizeof( unsigned long ) == 4 );
    assert( sizeof( time_t ) == 4 );

    /* check parameter */
    if ( NULL == msg )   {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }
    if ( 8 != msg->len )     {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }

    ret.fields = TCK_TICK_NR | TCK_BEGIN;
    memcpy( &ret.ticket_nr, msg->buf, 4 );
    ret.ticket_nr = ntohl( ret.ticket_nr );
    memcpy( &ret.begin, (msg->buf + 4), 4 );
    ret.begin = ntohl( ret.begin );

    return ( &ret );

}
```

```
int
tck_valid_simple2( const tck_tparam_t *param,
                   unsigned short hour,
                   unsigned short minute,
                   unsigned short second )
/*
 * The ticket parameters in *param are checked against the hh:mm:ss
 * requirements specified by the caller AND the date of today. By using
 * a fixed hh:mm:ss time (e.g. 12:00:00), tickets can be made valid for
 * a whole day.
 *
 * If *param meets all requirements, 1 is returned. If it does not, the
 * return value is 0. In case of error -1 is returned and tck_errno set
 * to an error code.
 */
{
    struct tm ptime, curtime;
    time_t ctme;

    if ( NULL == param )     {
        tck_errno = TCK_ERR_PARAM;
        return ( -1 );
    }
    if ( (TCK_BEGIN & param->fields) == 0 ) {
        tck_errno = TCK_ERR_PARAM;
        return ( -1 );
    }

    ctme = time( NULL );
    localtime_r( &ctme, &curtime );
    localtime_r( &param->begin, &ptime );

    /* the ticket is considered valid if its begin parameter is a time
     * within the range today 00:00:00 - today 24:00:00 and meets the
     * hh/mm/ss requirements specified by the caller.
     */
    if ( (ptime.tm_hour != hour) || (ptime.tm_min != minute)
          || (ptime.tm_sec != second) )   {
        tck_errno = TCK_ERR_TIME;
        return ( 0 );
    }
    if ( (ptime.tm_yday != curtime.tm_yday)
          || (ptime.tm_year != curtime.tm_year) )     {
        tck_errno = TCK_ERR_TIME;
        return ( 0 );
    }

    return ( 1 );

}


static tck_buffer_t *
param2msg_std( const tck_tparam_t *param )
/*
 * The standard message format. The message consists of the 2-byte value
 * param->fields and all fields of *param for which the corresponding flag
 * in param->fields is set concatenated together. All integers are in
 * network byteorder.
 */
{
    tck_buffer_t *ret;
    tck_tparam_t pcpy;  /* to save some declarations */

    assert( sizeof( unsigned short ) == 2 );
    assert( sizeof( unsigned long ) == 4 );
    assert( sizeof( time_t ) == 4 );

    if ( NULL == param )     {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }

    /* at least location, event, the time fields and price category must
     * be there
     */
    if ( (param->fields &
          (TCK_TICK_NR | TCK_LOC_NR | TCK_EVENT_NR | TCK_BEGIN | TCK_INLET))
         != (TCK_TICK_NR | TCK_LOC_NR | TCK_EVENT_NR | TCK_BEGIN | TCK_INLET)
        )    {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }

    /* copy *param to pcpy and do all the hton conversions
     */
    memcpy( &pcpy, param, sizeof( tck_tparam_t ) );
      /* remove flags for the three text fields at the end */
    pcpy.fields &= ~(TCK_TEXT1 | TCK_TEXT2 | TCK_TEXT3);
    pcpy.fields = htons( pcpy.fields );
```

```
    pcpy.ticket_nr = htonl( pcpy.ticket_nr );
    pcpy.location_nr = htons( pcpy.location_nr );
    pcpy.event_nr = htons( pcpy.event_nr );
    pcpy.begin = htonl( pcpy.begin );
    pcpy.duration = htons( pcpy.duration );
    pcpy.inlet = htons( pcpy.inlet );
    pcpy.seat = htons( pcpy.seat );
    pcpy.row = htons( pcpy.row );
    pcpy.price = htons( pcpy.price );

    /* malloc the buffer; max. size is sizeof( tck_tparam_t ) */
    ret = tck_bmalloc( sizeof( tck_tparam_t ) );
    if ( NULL == ret )  {
        tck_errno = TCK_ERR_MEM;
        return ( NULL );
    }

    /* For each field, if coressponding flag is set in param->fields
     * copy contents to buffer.
     */
    memcpy( ret->buf, &pcpy.fields, 2 );
    ret->len = 2;

    if ( (TCK_TICK_NR & param->fields) != 0 )   {
        memcpy( (ret->buf + ret->len), &pcpy.ticket_nr, 4 );
        ret->len += 4;
    }
    if ( (TCK_LOC_NR & param->fields) != 0 )    {
        memcpy( (ret->buf + ret->len), &pcpy.location_nr, 2 );
        ret->len += 2;
    }
    if ( (TCK_LOC_TEXT & param->fields) != 0 )  {
        memcpy( (ret->buf + ret->len), &pcpy.location_text, TCK_STRLEN );
        ret->len += TCK_STRLEN;
    }
    if ( (TCK_EVENT_NR & param->fields) != 0 )  {
        memcpy( (ret->buf + ret->len), &pcpy.event_nr, 2 );
        ret->len += 2;
    }
    if ( (TCK_EVENT_TEXT & param->fields) != 0 )    {
        memcpy( (ret->buf + ret->len), &pcpy.event_text, TCK_STRLEN );
        ret->len += TCK_STRLEN;
    }
    if ( (TCK_BEGIN & param->fields) != 0 ) {
        memcpy( (ret->buf + ret->len), &pcpy.begin, 4 );
        ret->len += 4;
    }
    if ( (TCK_DURATION & param->fields) != 0 )  {
        memcpy( (ret->buf + ret->len), &pcpy.duration, 2 );
        ret->len += 2;
    }
    if ( (TCK_INLET & param->fields) != 0 ) {
        memcpy( (ret->buf + ret->len), &pcpy.inlet, 2 );
        ret->len += 2;
    }
    if ( (TCK_SEAT & param->fields) != 0 )  {
        memcpy( (ret->buf + ret->len), &pcpy.seat, 2 );
        ret->len += 2;
    }
    if ( (TCK_ROW & param->fields) != 0 )   {
        memcpy( (ret->buf + ret->len), &pcpy.row, 2 );
        ret->len += 2;
    }
    if ( (TCK_CATEGORY & param->fields) != 0 )  {
        ret->buf[ret->len] = pcpy.category;
        ret->len += 1;
    }
    if ( (TCK_PRICE & param->fields) != 0 ) {
        memcpy( (ret->buf + ret->len), &pcpy.price, 2 );
        ret->len += 2;
    }

    /* The three text fields at the end are not included. */

    return ( ret );
}


static tck_tparam_t *
msg2param_std( const tck_buffer_t *msg )
/*
 * Message format as defined for param2msg_std().
 *
 * This is only an example; for real-world applications there should be
 * a fixed format and input should be checked against specification.
 */
{
    static tck_tparam_t ret;
```

```
unsigned offset = 0;

assert( sizeof( unsigned short ) == 2 );
assert( sizeof( unsigned long ) == 4 );
assert( sizeof( time_t ) == 4 );

/* check parameter */
if ( tck_bempty( msg ) )     {
    tck_errno = TCK_ERR_PARAM;
    return ( NULL );
}
if ( msg->len < 2 ) {
    tck_errno = TCK_ERR_PARAM;
    return ( NULL );
}

memset( &ret, 0, sizeof( ret ) );
memcpy( &ret.fields, msg->buf, 2 );
ret.fields = ntohs( ret.fields );
offset = 2;

/* read parameters from buffer
 * Must read in the same order as param2msg_std() writes!
 */
if ( ((TCK_TICK_NR & ret.fields) != 0)
        && ((msg->len - offset) >= 4 ) ) {
    memcpy( &ret.ticket_nr, (msg->buf + offset), 4 );
    offset += 4;
}
if ( ((TCK_LOC_NR & ret.fields) != 0) && ((msg->len - offset) >= 2 ) ) {
    memcpy( &ret.location_nr, (msg->buf + offset), 2 );
    offset += 2;
}
if ( ((TCK_LOC_TEXT & ret.fields) != 0)
     && ((msg->len - offset) >= TCK_STRLEN) )    {
    memcpy( &ret.location_text, (msg->buf + offset), TCK_STRLEN );
    offset += TCK_STRLEN;
}
if ( ((TCK_EVENT_NR & ret.fields) != 0) && ((msg->len - offset) >= 2 ) ) {
    memcpy( &ret.event_nr, (msg->buf + offset), 2 );
    offset += 2;
}
if ( ((TCK_EVENT_TEXT & ret.fields) != 0)
     && ((msg->len - offset) >= TCK_STRLEN) )    {
    memcpy( &ret.event_text, (msg->buf + offset), TCK_STRLEN );
    offset += TCK_STRLEN;
}
if ( ((TCK_BEGIN & ret.fields) != 0) && ((msg->len - offset) >= 4 ) ) {
    memcpy( &ret.begin, (msg->buf + offset), 4 );
    offset += 4;
}
if ( ((TCK_DURATION & ret.fields) != 0) && ((msg->len - offset) >= 2 ) ) {
    memcpy( &ret.duration, (msg->buf + offset), 2 );
    offset += 2;
}
if ( ((TCK_INLET & ret.fields) != 0) && ((msg->len - offset) >= 2 ) ) {
    memcpy( &ret.inlet, (msg->buf + offset), 2 );
    offset += 2;
}
if ( ((TCK_SEAT & ret.fields) != 0) && ((msg->len - offset) >= 2 ) ) {
    memcpy( &ret.seat, (msg->buf + offset), 2 );
    offset += 2;
}
if ( ((TCK_ROW & ret.fields) != 0) && ((msg->len - offset) >= 2 ) ) {
    memcpy( &ret.row, (msg->buf + offset), 2 );
    offset += 2;
}
if ( ((TCK_CATEGORY & ret.fields) != 0) && ((msg->len - offset) >= 1 ) ) {
    ret.category = msg->buf[offset];
    offset += 1;
}
if ( ((TCK_PRICE & ret.fields) != 0) && ((msg->len - offset) >= 2 ) ) {
    memcpy( &ret.price, (msg->buf + offset), 2 );
    offset += 2;
}

/* now there should be no byte left */
if ( (msg->len - offset) != 0 ) {
    tck_errno = TCK_ERR_DECODE;
    return ( NULL );
}

/* do all the ntoh conversions */
ret.ticket_nr = ntohl( ret.ticket_nr );
ret.location_nr = ntohs( ret.location_nr );
ret.event_nr = ntohs( ret.event_nr );
ret.begin = ntohl( ret.begin );
ret.duration = ntohs( ret.duration );
ret.inlet = ntohs( ret.inlet );
```

```
    ret.seat = ntohs( ret.seat );
    ret.row = ntohs( ret.row );
    ret.price = ntohs( ret.price );

    /* enforce zero termination of strings */
    ret.location_text[TCK_STRLEN - 1] = 0;
    ret.event_text[TCK_STRLEN - 1] = 0;

    return ( &ret );
}


int
tck_valid_std( const tck_tparam_t *param,
               unsigned short location,
               unsigned short event,
               unsigned char *categories,
               int timecheck )
/*
 * If timecheck is set to true, tck_valid_std() checks for current time
 * being in the range between (begin - inlet) and (begin + duration).
 * If not, only the location_nr, event_nr and category fields are checked
 * against the requirements specified by the caller.
 *
 * If *param meets all requirements, 1 is returned. If it does not, the
 * return value is 0. In case of error -1 is returned and tck_errno set
 * to an error code.
 */
{
    bool catok = FALSE;
    int i;

    if ( (NULL == param) || (NULL == categories) )  {
        tck_errno = TCK_ERR_PARAM;
        return ( -1 );
    }

    if ( NULL != categories )   {
        if ( ((TCK_CATEGORY & param->fields) == 0) && (0 != categories[0]) ) {
            tck_errno = TCK_ERR_PARAM;
            return ( -1 );
        }
        if ( 0 != categories[0] )   {
            for ( i = 0; categories[i] != 0; i++ )  {
                if ( param->category == categories[i] )  {
                    catok = TRUE;
                    break;
                }
            }
        }
        else
            catok = TRUE;
    }
    else
        catok = TRUE;
    if ( ((TCK_LOC_NR & param->fields) == 0)
         || ((TCK_EVENT_NR & param->fields) == 0) )  {
        tck_errno = TCK_ERR_PARAM;
        return ( -1 );
    }
    if ( timecheck && (((TCK_BEGIN & param->fields) == 0)
         || ((TCK_INLET & param->fields) == 0)) )  {
        tck_errno = TCK_ERR_PARAM;
        return ( -1 );
    }

    if ( !catok )   {
        tck_errno = TCK_ERR_CATEG;
        return ( 0 );
    }
    if ( location != param->location_nr )   {
        tck_errno = TCK_ERR_LOC;
        return ( 0 );
    }
    if ( event != param->event_nr ) {
        tck_errno = TCK_ERR_EVENT;
        return ( 0 );
    }

    if ( timecheck )    {
        if ( (param->begin - param->inlet) > time( NULL ) ) {
            tck_errno = TCK_ERR_TIME;
            return ( 0 );
        }
        if ( (param->begin +
              ((param->fields & TCK_DURATION) ? param->duration : 0)
             ) < time( NULL ) ) {
            tck_errno = TCK_ERR_TIME;
            return ( 0 );
        }
    }
```

```
    return ( 1 );

}


tck_buffer_t *
tck_param2msg( tck_tparam_t *param, unsigned msgtype  )
/*
 * Encode parameters. This is only a wrapper around the tck_param2msg_*
 * finctions.
 */
{
    tck_buffer_t *ret;
    unsigned short tmpfields;

    /* check parameter */
    if ( NULL == param )     {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }

    /* there mus be a ticket number */
    if ( ( param->fields & TCK_TICK_NR) == 0 )    {
        /* none of them */
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }

    /* if present, category must not be 0 */
    if ( (( param->fields & TCK_CATEGORY) != 0) && (param->category == 0) )   {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }

    switch( msgtype )    {
    case TCK_MSGTYPE_SIMPLE1:
        return ( param2msg_simple1( param ) );
        break;  /* Yes, I *am* paranoid! */
    case TCK_MSGTYPE_SIMPLE2:
        return ( param2msg_simple2( param ) );
        break;
    case TCK_MSGTYPE_STD:
        return ( param2msg_std( param ) );
        break;
    case TCK_MSGTYPE_NOTEXT:
        /* The NOTEXT type is the same as STD except that the event_text
         * and location_text fields are not included in the message.
         */
        tmpfields = param->fields;
        param->fields &= ~(TCK_LOC_TEXT | TCK_EVENT_TEXT);
        ret = param2msg_std( param );
        param->fields = tmpfields;
        return ( ret );
        break;
    default:
        tck_errno = TCK_ERR_NOTSUP;
        return ( NULL );
    }

    tck_errno = TCK_ERR_UNSPEC;
    return ( NULL );
}


tck_tparam_t *
tck_msg2param( const tck_buffer_t *msg, unsigned msgtype )
/*
 * Decode parameters. This is a wrapper around the tck_msg2param_*
 * functions. The returned tck_tparam_t structure is malloc()ed and
 * must be free()ed by the caller.
 */
{
    tck_tparam_t *param;
    tck_tparam_t *ret;

    if ( tck_bempty( msg ) )     {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }

    /* for API consistency, copy parameter structs into malloc()-ed
     * memory
     */
    if ( ( ret = malloc( sizeof( tck_tparam_t ) )) == NULL )  {
        tck_errno = TCK_ERR_MEM;
        return ( NULL );
    }

    switch( msgtype )    {
```

```
    case TCK_MSGTYPE_SIMPLE1:
        param = msg2param_simple1( msg );
        break;
    case TCK_MSGTYPE_SIMPLE2:
        param = msg2param_simple2( msg );
        break;
    case TCK_MSGTYPE_STD:
    case TCK_MSGTYPE_NOTEXT:
        param = msg2param_std( msg );
        break;
    default:
        tck_errno = TCK_ERR_NOTSUP;
        param = NULL;
    }
    if ( NULL == param )     {
        free( ret );
        return ( NULL );
    }
    /* copy to a malloc()ed buffer for interface consistency */
    memcpy( ret, param, sizeof( tck_tparam_t ) );

    /* there must be a ticket number */
    if ( (param->fields & TCK_TICK_NR) == 0 )    {
        /* none of them */
        tck_errno = TCK_ERR_DECODE;
        free( ret );
        return ( NULL );
    }

    /* if present, category must not be 0 */
    if ( ((param->fields & TCK_CATEGORY) != 0) && (param->category == 0) )   {
        tck_errno = TCK_ERR_DECODE;
        free( ret );
        return ( NULL );
    }

    return ( ret );
}


unsigned long *
tck_nextnum( unsigned char *countfile )
/*
 * Increase the shared counter in countfile by one and return a pointer to
 * the new value. If the counter does not exist, it is created and started
 * with a value of one.
 * It is ensured by lockf() record locking that multiple processes using
 * this function can share a counter without messing it up.
 * This function is used to get unique ticket ID numbers in a multiprocessed
 * environment.
 * In case of error a NULL pointer is returned and tck_errno is set.
 */
{
    static unsigned long ret;
    int fd = 0;
    ssize_t bts;

    if ( NULL == countfile )     {
        tck_errno = TCK_ERR_PARAM;
        return ( NULL );
    }

    /* open the file; if it does not exist it is created */
    if ( ( fd = open( (char *) countfile,
                      O_RDWR|O_SYNC|O_CREAT, 0600 )) == -1 ) {
        tck_errno = TCK_ERR_OPEN;
        return ( NULL );
    }

    /* No try to lock the whole file. The process will sleep until this
     * operation can be performed.
     */
#ifdef LINUX
    if ( flock( fd, LOCK_EX ) == -1 )    {
        close( fd );
        tck_errno = TCK_ERR_LOCK;
        return ( NULL );
    }
#else    /* LINUX */
    if ( lockf( fd, F_LOCK, 0 ) == -1 ) {
        close( fd );
        tck_errno = TCK_ERR_LOCK;
        return ( NULL );
    }
#endif   /* LINUX */

    /* now try to read sizeof( ret ) bytes into ret */
    if ( ( bts = read( fd, &ret, sizeof( ret ) )) == -1 )     {
        close( fd );          /* this should release the lock automagically */
        tck_errno = TCK_ERR_READ;
        return ( NULL );
```

```
    }

    if ( ( sizeof( ret ) != bts) && (0 != bts) )      {
         /* too many or too few bytes - can't be a counter */
         close( fd );          /* this should release the lock automagically */
         tck_errno = TCK_ERR_READ;
         return ( NULL );
    }

    /* set file pointer back to start of file */
    if ( lseek( fd, 0, SEEK_SET ) ) {
         close( fd );          /* this should release the lock automagically */
         tck_errno = TCK_ERR_UNSPEC;
         return ( NULL );
    }

    /* if we read 0 bytes, we must have created the file
     * (is this true?)
     */
    if ( 0 == bts )
         ret = 1;
    else
         ret += 1;

    if ( write( fd, &ret, sizeof( ret ) ) != sizeof( ret ) )     {
         /* could leave file in an inconsistent state */
         close( fd );
         tck_errno = TCK_ERR_WRITE;
         return ( NULL );
    }

    /* close file, releasing all locks */
    close( fd );

    return ( &ret );

}
```

C Zur beigefügten Diskette

Die Diskette enthält die Archive `aztec.tgz` (Aztec-Symbolerzeugung) und `ticket.tgz` (Ticketerstellung und -kontrolle). Sie sind mit `gunzip` zu dekomprimieren und mit `tar` auszupacken. Die enthaltenen Quelltexte lassen sich unter Solaris problemlos übersetzen – just type `make` –; für andere Unices sind unter Umständen Anpassungen nötig.

Der Aztec-Generator benötigt die Biliotheken cgic [Bou96a], GD [Bou98] und Libpng [PDG98]. Die Ticketprogramme werden gegen die libcrypto aus SSLeay [HuYo98] gelinkt, das Verkaufsprogramm außerdem gegen cgic. Die Pfade zu den Bilbliotheken und ihren Headerfiles sind in den Makefiles einzutragen.

Beim Übersetzen des Aztec-Generators entstehen die Bibliothek `libaztec.a`, das Programm `azdemo` zur Symbolerzeugung auf der Kommandozeile und das CGI-Programm `azcgi`.

Die Ticketsoftware besteht aus den drei Programmen `sale` für den Verkauf, `checkpoint` zur Kontrolle sowie `keymgr` für die Schlüsselverwaltung.

Zusammenfassung

Die Arbeit wendet sich der Distributionsphase des elektronischen Handels zu. Verkauf und Bezahlung sind in unsicheren Netzen möglich, aber die gekauften Waren oder Dienstleistungen können nur im Netz übermittelt werden, wenn sie vollständig digitalisierbar sind. Untersucht wird, ob und wie Fahr- und Eintrittskarten zur Übertragung in digitaler Form dargestellt werden können, ohne daß die leichte Kopierbarkeit solcher Daten Betrugsmöglichkeiten eröffnet.

Es wird gezeigt, daß Eintrittskarten, die nur an einem Ort gültig sind, im Netz verkauft und vom Käufer selbst gedruckt werden können. Sie werden dazu mit einem 2D-Barcode versehen, der kryptographisch gesicherte Daten in maschinenlesbarer Form enthält. Durch eindeutige Nummerierung kann sichergestellt werden, daß von mehreren Kopien einer solchen Eintrittskarte nur eine einzige benutzbar ist.

Weiter wird ausgeführt, warum Fahrkarten nicht auf diese einfache und auch sonst auf keine praktisch brauchbare Weise in unsicheren Netzen verkauft werden können, jedenfalls dann nicht, wenn die Käufer anonym und die Kommunikationskosten gering bleiben sollen. Solche Tickets lassen sich nur mit Chipkarten realisieren; die Arbeit nennt Gründe, das lieber nicht zu tun.

Neben der Anwendbarkeit kryptographischen Verfahren untersucht die Arbeit Fragen der praktischen Sicherheit sowie die Robustheit der gewählten Ticketdarstellung durch 2D-Matrixkodes unter Alltagsbedingungen.

Für den Verkauf und die Kontrolle von Eintrittskarten wurde ein Prototyp implementiert. Als Nebenprodukt entstand Software zur Kodierung von Daten in Symbolen des Aztec-Kodes, die auch für andere Zwecke genutzt werden kann.

Literatur und andere Quellen

[Ada98] Russ Adams: *BarCode1 - A Web Of Information About Bar Code.* URL: `http://www.adams1.com/`; 1998-10-24.

[AGF98] Urteil des Amtsgerichtes Frankfurt am Main; Aktenzeichen 30 C 2119 / 97 - 45; 1998-09-01. URL: `https://www.ccc.de/CRD/CRD210998.html`; 1998-10-21.

[AIM94] AIM Europe: *Uniform Symbology Specification - PDF417.*

[AIM97] AIM Europe: *Description of ISS Aztec Code.* URL: `http://www.aim-europe.org/standards/azcodspc.htm`; 1998-10-28.

[AnBe96] Ross J. Anderson; S. Johann Bezuidenhoudt: *On the Reliability of Electronic Payment Systems.* in: IEEE Transactions on Software Engineering, Vol. 22, No. 5, May 1996. URL: `http://www.cl.cam.ac.uk/users/rja14/meters.ps.gz`; 1998-08-22.

[And93] Ross J Anderson: *Why Cryptosystems Fail.* URL: `http://www.cl.cam.ac.uk/ftp/users/rja14/wcf.ps.gz`; 1998-10-21.

[And94] Ross Anderson: *Liability and Computer Security - Nine Principles.* in: Computer security: proceedings / ESORICS 94; LNCS 875. Berlin; Heidelberg; New York [u.a.]: Springer, 1994. URL: `http://www.cl.cam.ac.uk/ftp/users/rja14/liability.ps.gz`; 1998-10-21.

[Auf98] Kleiner Aufmerksamkeitstest: *Herzlichen Glückwunsch, Sie haben bestanden!* Darmstadt, 1998.

[BaRi96] R. Baldwin; R. Rivest: *The RC5, RC5-CBC, RC5-CBC-Pad, and RC5-CTS Algorithms.* Request for Comments 2040. URL: z.B. `ftp://ftp.tu-chemnitz.de/pub/documents/rfc/rfc2040.txt`; 1998-10-24.

[BCK96] Mihir Bellare; Ran Canetti; Hugo Krawczyk: *The HMAC Construction.* in: CryptoBytes Vol. 2, No. 1 - Spring 1996. URL: `http://www.rsa.com/rsalabs/pubs/cryptobytes/`; 1998-10-20.

[BDP97] Antoon Bosselaers; Hans Dobbertin; Bart Preneel: *The RIPEMD-160 Cryptographic Hash Function: A secure replacement for MD4 and MD5.* in: Dr. Dobb's Journal, January 1997.

[Beu94] Albrecht Beutelspacher: *Kryptologie.* 4., verbesserte Auflage; Braunschweig/Wiesbaden: Vieweg, 1994.

[Bie84] Gudrun Biermann: *Bildschirmtext: Neue Wege der Kommunikation.* Bergisch Gladbach: Lübbe, 1984.

[Bih96] Eli Biham: *How to Forge DES-Encrypted Messages in 2^{28} Steps.* Technion - Israel Institute of Technology; Technical Report CS 884; 1996. URL: `http://www.cs.technion.ac.il/~biham/publications.html`; 1998-10-11.

[Bla96] M. Blaze; W. Diffie; R. L. Rivest; B. Schneier; T. Shimomura; E. Thompson; Michael Wiener: *Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security.* URL: `ftp://ftp.research.att.com/dist/mab/keylength.txt`; 1998-10-20.

[Bou96a] Thomas Boutell: *cgic: an ANSI C library for CGI Programming.* URL: `http://www.boutell.com/cgic/`; 1998-10-31.

[Bou96b] Thomas Boutell (ed.): *PNG (Portable Network Graphics) Specification.* Version 1.0; URL: `http://www.w3.org/TR/REC-png-multi.html`; 1998-10-31.

[Bou98] Thomas Boutell: *gd 1.3: A graphics library for fast GIF creation.* URL: `http://www.boutell.com/gd/`; 1998-10-31.

[Bra98] Stefan Brands: *Electronic Cash.* in: Michael Atallah (ed.): Handbook on Algorithms and Theory of Computation. Erscheint im November 1998 bei CRC Press. URL: `http://www.xs4all.nl/~brands/draft.pdf`; 1998-10-27.

[Bre85] Klaus Brepohl: *Lexikon der neuen Medien: Von Abonnement-Fernsehen bis Zweiwegkommunikation.* Bergisch Gladbach: Lübbe, 1985.

[BSW95] Albrecht Beutelspacher; Jörg Schwenk; Klaus-Dieter Wolfenstetter: *Moderne Verfahren der Kryptographie.* Braunschweig/Wiesbaden: Vieweg, 1995.

[CFN90] David Chaum; Amos Fiat; Moni Naor: *Untraceable Electronic Cash.* in: Advances in Cryptology / CRYPTO '88; LNCS 403. Berlin; Heidelberg; New York [u.a.]: Springer, 1990.

[Cur98] Matt Curtin: *Snake Oil Warning Signs: Encryption Software to Avoid.* URL: `http://www.interhack.net/people/cmcurtin/snake-oil-faq.html`; 1998-10-24.

[DPA98a] *Post warnt vor Nutzung von Briefkästen in Seattle.* DPA-Meldung, 1998-10-22.

[DPA98b] *WM-Ticketbetrüger in Frankreich verhaftet.* DPA-Meldung, 1998-07-10.

[Dud95] Underwood Dudley: *Mathematik zwischen Wahn und Witz: Trugschluesse, falsche Beweise und die Bedeutung der Zahl 57 fuer die amerikanische Geschichte.* Basel; Boston; Berlin: Birkhaeuser, 1995.

[ECS94] D. Eastlake; S. Crocker; J. Schiller: *Randomness Recommendations for Security.* Request for Comments 1750. URL: z.B. `ftp://ftp.tu-chemnitz.de/pub/documents/rfc/rfc1750.txt`; 1998-10-26.

[EFF98a] Electronic Frontier Foundation: *EFF DES Cracker Project.* URL: `http://www.eff.org/descracker/`; 1998-10-19.

[EFF98b] Electronic Frontier Foundation: *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design.* O'Reilly & Associates, 1998.

[ESta98] E-Stamp Corporation: *Frequently Asked Questions.* URL: `http://www.e-stamp.com/products/faq.html`; 1998-10-13.

[Fox98] Dirk Fox: *Zu einem grundsätzlichen Problem digitaler Signaturen.* in: DuD - Datenschutz und Datensicherheit 22 (1998), Heft 7. Wiesbaden: Vieweg.

[FRW98] Bernhard Fastenrath; Thomas Reiners; Sven-Holger Wabnitz: *Smart is beautiful.* in: iX 1998, Heft 9.

[FrYu93] M. Franklin; M. Yung: *Secure and Efficient Off-line Digital Money.* in: Automata, languages and programming: proceedings / ICALP 93; LNCS 700. Berlin; Heidelberg; New York [u.a.]: Springer, 1993.

[FuWr96] Andreas Furche; Graham Wrightson: *Computer Money: A systematic overview of electronic payment systems.* Heidelberg: dpunkt, Verl. für Digitale Technologie, 1996.

[Gal94] Jirka Gallas: *Serial Programming Guide for POSIX Compliant Operating Systems.* URL: `http://home.eunet.cz/jirig/serial/`; 1998-11-05.

[Gan95] Mike Gancarz: *The Unix Philosophy.* Digital Press, 1995.

[Gra90] Timm Grams: *Denkfallen und Programmierfehler.* Berlin; Heidelberg; New York [u.a.]: Springer, 1990.

[Har94] Craig K. Harmon: *Lines of Communication. Bar Code and Data Collection Technology for the 90s.* Peterborough: Helmers Publishing, 1994.

[Hen93] Eckhard Henscheid: *Dummdeutsch.* Stuttgart: Reclam 1993.

[Hue98] Stephan Hüttinger, persönliche Mitteilung.

[HuYo98] T. J. Hudson; E. A. Young: *SSLeay and SSLapps FAQ.* URL: `http://www.cryptsoft.com/ssleay/`; 1998-10-24.

[JuMe97] Aleksandar Jurišić; Alfred J. Menezes: *Elliptic Curves and Cryptography: Strong digital signature algorithms.* in: Dr. Dobb's Journal, April 1997.

[Kab96] Nikola Kabel: *Die branchenübergreifende elektronische Geldbörse - technische Analyse und Einsatzmöglichkeiten im Zahlungsverkehr.* Diplomarbeit; Technische Universität Berlin, Institut für Informatik und Gesellschaft, 1996. URL: `http://ig.cs.tu-berlin.de/DA/042/diplom.html`; 1998-10-12.

[Karn96] Phil Karn: *Reed-Solomon coding/decoding package v1.0.* URL: `http://people.qualcomm.com/karn/dsp.html#4`; 1998-10-30.

[KaRo95] Burt Kaliski; Matt Robshaw: *Message Authentication with MD5.* in: CryptoBytes Vol. 1, No. 1 - Spring 1995. URL: `http://www.rsa.com/rsalabs/pubs/cryptobytes/`; 1998-10-20.

[KBC97] H. Krawczyk; M. Bellare; R. Canetti: *HMAC: Keyed-Hashing for Message Authentication.* Request for Comments 2104. URL: z.B. `ftp://ftp.tu-chemnitz.de/pub/documents/rfc/rfc2104.txt`; 1998-10-20.

[Kor98] Jukka Korpela: *A tutorial on character code issues.* URL: `http://www.hut.fi/u/jkorpela/chars.html`; 1998-10-28.

[Kuhn97] Markus G. Kuhn: *Probability Theory for Pickpockets - ec-PIN Guessing.* URL: `http://www.cl.cam.ac.uk/~mgk25/ec-pin-prob.pdf`; 1998-10-20.

[Lei92] Jerry Leichter: *Latest (?) credit card scams.* in: Forum on Risks to the Public in Computers and Related Systems Volume 14, Issue 20. URL: `http://catless.ncl.ac.uk/Risks/14.20.html`; 1998-10-15.

[Luk97] Sylvia Lukas: *Cyber money: künstliches Geld in Internet und elektronischen Geldbörsen.* Neuwied; Kriftel/Ts.; Berlin: Luchterhand, 1997.

[Lyp97] Hugo Lyppens: *Reed-Solomon Error Correction: A fast implementation in assembly language and C.* in: Dr. Dobb's Journal, January 1997.

[MaBo94] Wenbo Mao; Colin Boyd: *Classification of Cryptographic Techniques in Authentication Protocols.* in: Selected Areas in Cryptography. Kingston, Ontario, Canada: 1994. URL: `http://www.hpl.hp.co.uk/people/wm/papers/refine.ps`; 1998-10-09.

[Mei97] Reinhard Meindl: *Arbeitsweise und Einsatz kontaktloser Chipkarten.* in: it+ti - Informationstechnik und Technische Informatik 39 (1997) 5.

[Mil98] Barton P. Miller et al.: *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services.* URL: `ftp://grilled.cs.wisc.edu/technical_papers/fuzz-revisited.ps`; 1998-10-23.

[Moe97] Ulf Möller: *FAQ: Sicherheit von EC-Karten.* URL: `http://www.fitug.de/ulf/faq/pin.html`; 1998-10-20.

[MOV96] Alfred J. Menezes; Paul C. van Oorschot; Scott A. Vanstone: *Handbook of Applied Cryptography.* CRC Press, 1996. Einige Kapitel sind kostenlos abrufbar unter URL: `http://www.dms.auburn.edu/hac/`; 1998-10-21.

[MPW92] C. J. Mitchell; F. Piper; P. Wild: *Digital Signatures.* in: Contemporary Cryptology: the science of information integrity. IEEE Press, 1992.

[MWSl77] F. J. MacWilliams; N. J. A. Sloane: *The Theory of Error-Correcting Codes.* Amsterdam [u.a.]: North-Holland, 1977.

[Net98] Netscape Communications Corporation: *Secure Sockets Layer.* URL: `http://sitesearch.netscape.com/products/security/ssl/`; 1998-10-24.

[Neu95] Peter G. Neumann: *Computer-Related Risks.* Reading, Mass.: Addison-Wesley, 1995.

[NIST98] National Institute of Standards and Technology: *Advanced Encryption Standard (AES) Development Effort.* URL: `http://www.nist.gov/aes`; 1998-10-26.

[Nor90] Donald A. Norman: *The Design of Everyday Things.* New York: Currency and Doubleday, 1990.

[OkOh91] Tatsuaki Okamoto; Kazuo Ohta: *Universal electronic cash.* in: Advances in cryptology: proceedings / CRYPTO '91; LNCS 576. Berlin; Heidelberg; New York [u.a.]: Springer, 1992.

[Ope98] Opera Software: *Our marvellous browser: so many features – so little code!* URL: `http://www.operasoftware.com/features.html`; 1998-10-12.

[OS98] *Open Source: the Future is Here.* URL: `http://www.opensource.org/`; 1998-10-23.

[Pal95] Roger C. Palmer: *The Bar Code Book.* 3rd edition; Helmers Publishing, 1995.

[PDG98] PNG Development Group: *PNG library (libpng).* URL: `http://www.hensa.ac.uk/png/png/src/`; 1998-10-31.

[RaFu89] T. R. N. Rao; E. Fujiwara: *Error-Control Coding for Computer Systems.* Englewood Cliffs, NJ: Prentice-Hall, 1989.

[Ray96] Eric S. Raymond (ed.): *The New Hacker's Dictionary.* Cambridge, Mass.: MIT Press, 1996.

[Ray98] Eric S. Raymond (ed.): *The on-line hacker Jargon File, version 4.0.0.* URL: `http://www.ccil.org/jargon/`; 1998-10-11.

[Rei98] Nicolas Reichelt: *www.crypto.de: Verhindert ein Kryptographie-Gesetz!* URL: `http://www.crypto.de/`; 1998-10-18.

[Reif96] Holger Reif: *Schlüsselfertig. Secure Socket Layer: Chiffrieren und Zertifizieren mit SSLeay.* in: iX 1996, Heft 6. URL: `http://www.heise.de/ix/artikel/9606128/`; 1998-11-03.

[Rit98] Terry Ritter: *Ciphers by Ritter: New Encryption Technologies for Communications Designers.* URL: `http://www.io.com/~ritter/`; 1998-10-26.

[Riv92] R. Rivest: *The MD5 Message-Digest Algorithm.* Request for Comments 1321. URL: z.B. `ftp://ftp.tu-chemnitz.de/pub/documents/rfc/rfc1321.txt`; 1998-10-20.

[Riv97] Ronald L. Rivest: *Perspectives on Financial Cryptography.* in: Financial cryptography: first international conference; proceedings / FC '97; LNCS 1318. Berlin; Heidelberg; New York [u.a.]: Springer, 1997.

[RLab98] RSA Laboratories: *Frequently Asked Questions About Today's Cryptography.* Version 4.0. RSA Data Security, Inc; 1998. URL: `http://www.rsa.com/rsalabs/faq/`; 1998-10-18.

[RLJ98] Dave Raggett; Arnaud Le Hors; Ian Jacobs: *HTML 4.0 Specification.* W3C Recommendation, revised on 24-Apr-1998. URL: `http://www.w3.org/TR/REC-html40/`; 1998-11-03.

[Schn96] Bruce Schneier: *Angewandte Kryptographie: Protokolle, Algorithmen und Sourcecode in C.* Bonn; Reading, Mass. [u.a.]: Addison-Wesley, 1996.

[Schn97] Bruce Schneier: *Why Cryptography Is Harder Than It Looks.* URL: `http://www.counterpane.com/whycrypto.html`; 1998-10-21.

[Schn98] Bruce Schneier: *Security Pitfalls in Cryptography.* URL: `http://www.counterpane.com/pitfalls.html`; 1998-10-21.

[Schu91] Ralph-Hardo Schulz: *Codierungstheorie: Eine Einführung.* Braunschweig/Wiesbaden: Vieweg, 1991.

[Schu98] Christiane Schulzki-Haddouti: *Grobschnitt. Das Online-Recht muß nachgebessert werden.* in: c't 1998, Heft 16.

[Schw98] Holger Schwichtenberg: *Überweisungsträger: Rechnungen per Internet versenden und bezahlen.* in: iX 1998, Heft 9.

[ScKe98] Bruce Schneier; John Kelsey: *Cryptographic Support for Secure Logs on Untrusted Machines.* in: The Seventh USENIX Security Symposium Proceedings; USENIX Press: 1998. URL: `http://www.counterpane.com/secure-logs.html`; 1998-10-13.

[Sel97] Frank Seliger: *Informationen zur ec-GeldKarte.* URL: `http://www.darmstadt.gmd.de/~seliger/GeldKarte/ecindex.html`; 1998-10-16.

[Sim92] G. J. Simmons: *A Survey of Information Authentication.* in: Contemporary Cryptology: the science of information integrity. IEEE Press, 1992.

[Swe92] Peter Sweeney: *Codierung zur Fehlererkennung und Fehlerkorrektur.* München: Hanser; London: Prentice Hall, 1992.

[TYH96] J. D. Tygar; Bennet S. Yee; Nevin Heintze: *Cryptographic Postage Indicia.* in: Concurrency and parallelism, programming, networking, and security: proceedings / ASIAN '96; LNCS 1179. Berlin; Heidelberg; New York [u.a.]: Springer, 1996. URL: `http://www.cs.cmu.edu/afs/cs.cmu.edu/user/tygar/www/recommend.html`; 1998-10-13.

[USPS98] U.S. Postal Service: *Lick, Stick and now Click, First PC-generated Postage Unveiled.* Press Release No. 31; March 31, 1998. URL: `http://www.usps.gov/news/press/98/98031new.htm`; 1998-10-13.

[WAl97] Welch Allyn Inc.: *Aztec Barcode Symbology Specification Rev 3.0.* URL: `http://dcd.welchallyn.com/techover/dcdwhite.htm`; 1997-10-24.

[Wai88] Michael Waidner: *Betrugssicherheit durch kryptographische Protokolle beim Wertetransfer über Kommunikationsnetze.* in: Datenschutz und Datensicherung DuD/9 (1988) 448-459. URL: `http://www.semper.org/sirene/publ/Wai1_88DFG.DuD.ps.gz`; 1998-08-27.

[WFN] Wirtschaftsverband der Filmtheater Nord-West e.V.: *Beanstandungen von Spielfilmabrechnungen durch die Abrechnungskontrolle.* in: Der Filmtheaterkaufmann, Band 3, Heft 13.

[Wob97] Reinhard Wobst: *Abenteuer Kryptologie: Methoden, Risiken und Nutzen der Datenverschlüsselung.* Bonn; Reading, Mass. [u.a.]: Addison-Wesley-Longman, 1997.

[Yeh9?] Chung-Tsai Yeh: *A Study on Two Dimensional Code and its Application.* Institute of Computer and Information Science, National Chiao Tung University, Taiwan. URL: `http://debut.cis.nctu.edu.tw/~ctyeh/thesis.htm`; 1998-10-24.

[Zar97] Robert Morelos-Zaragoza: *The Error Correcting Codes (ECC) Home Page.* URL: `http://hideki.iis.u-tokyo.ac.jp/~robert/codes.html`; 1998-10-24.

Hinweis: Zu allen URLs von Netzressourcen ist das Datum des letzten Abrufs angegeben. Inhalte und Adressen können sich zwischenzeitlich geändert haben.

Erklärung

Ich versichere, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Leipzig, 1998-11-06