KAPITEL 1

Grundlagen

Blockchains setzen zwei kryptografische Grundbausteine voraus, die wir am Anfang dieses Kapitels kennenlernen: kryptografische Hashfunktionen und digitale Signaturen. Beide sind verhältnismäßig einfach und gut erforscht. Leser mit entsprechendem Vorwissen sind eingeladen, die entsprechenden Abschnitte zu überspringen.

Dann werden wir sehen, wie man auf das berühmte Double-Spending-Problem stößt, wenn man versucht, mithilfe digitaler Signaturen digitales Bargeld zu schaffen. Das Double-Spending-Problem besteht darin, dass ein dezentrales Netzwerk bei zwei widersprüchlichen Transaktionen nicht ohne Weiteres zu einem Konsens darüber kommen kann, welche der beiden Transaktionen gelten soll. Die Blockchain ist in erster Linie eine Lösung dieses Problems.

Danach betrachten wir digitale Zeitstempel oder Timestamping. Zeitstempel sind zum einen eine gute Illustration der Sicherheitseigenschaften von Hashfunktionen und digitalen Signaturen. Zum anderen ist eine Blockchain im Kern eben eine dezentrale Timestamping-Methode: Wenn das Netzwerk die zeitliche Reihenfolge der beiden widersprüchlichen Transaktionen kennen würde, dann könnte es einfach die erste gelten lassen und die zweite verwerfen. Dann wäre das Double-Spending-Problem gelöst.

Dieser dezentrale Timestamping-Mechanismus beruht auf einem weiteren kryptografischen Konzept: *Proof-of-Work*. Proof-of-Work ist integraler Bestandteil einer Blockchain und bezeichnet eine Methode, mit der man beweisen kann, dass man eine bestimmte Rechenarbeit geleistet hat.

Proof-of-Work wurde schon vor der Erfindung einer Blockchain genutzt, zum Beispiel im sogenannten Hashcash-Protokoll, das dazu

dient, Spam zu bekämpfen. Eine Einführung von Proof-of-Work beendet das Kapitel zu den kryptografischen Grundlagen.

Hashfunktionen

Eine *Hashfunktion* bildet eine Bitfolge beliebiger Länge auf eine Bitfolge fester Länge ab und ist effizient berechenbar.

Eine vielfach verwendete Hashfunktion ist *SHA-256*. Eine Implementation finden wir zum Beispiel in der Python-Standardbibliothek.

```
$ python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
>>> import hashlib
>>> print(hashlib.sha256(b"Satoshi Nakamoto").hexdigest())
a0dc65ffca799873cbea0ac274015b9526505daaaed385155425f7337704883e
>>>
```

Im gegebenen Beispiel wird der Hash der zugrunde liegenden Bitfolge des Strings `Satoshi Nakamoto` berechnet. Die Funktion `sha256()` gibt ein Objekt zurück, das den 256-bittigen binären Ausgabewert der Hashfunktion enthält. Auf diesem wird dann die Funktion `hexdigest()` aufgerufen, die diesen Ausgabewert in Hexadezimaldarstellung zurückgibt.

Änderungen am Eingabewert führen mit an Sicherheit grenzender Wahrscheinlichkeit zu Änderungen am Ausgabewert. Wenn wir zum Beispiel das letzte Zeichen unseres Eingabestrings ändern, erhalten wir einen völlig anderen Hashwert:

```
>>> print(hashlib.sha256(b"Satoshi Nakamot0").hexdigest())
73d607aab917435d5e79857769996c95027d4e42172698e0776e1295e285730e
```

Eine mögliche Anwendung von Hashfunktionen ist das Erkennen von Übertragungsfehlern. Alice hat wenig Platz auf ihrem Laptop, sie will deshalb eine große Datei auf dem Server von Bob speichern und sie lokal löschen. Bevor sie die Datei auf den Server von Bob hochlädt, berechnet sie den Hashwert der Datei und speichert ihn

lokal. So kann sie bei späterem Herunterladen der Datei erkennen, ob es zu Übertragungsfehlern gekommen ist: Alice berechnet den Hash der heruntergeladenen Datei und vergleicht ihn mit ihrem lokal gespeicherten Hash. Wenn beide gleich sind, kann Alice davon ausgehen, dass die Datei korrekt übertragen wurde.

Kryptografische Hashfunktionen. Eine *kryptografische Hashfunktion* ist eine Hashfunktion, die gewisse Sicherheitseigenschaften hat. Die beiden wichtigsten typischerweise geforderten Sicherheitseigenschaften sind die *Einwegfunktionseigenschaft* (englisch: preimage-resistance) und die *Kollisionsresistenz* (englisch: collision-resistance). Die Hashfunktion SHA-256 ist eine Einwegfunktion und auch kollisionsresistent.

Definition: Einwegfunktion

Eine Hashfunktion h ist eine *Einwegfunktion*, wenn es praktisch unmöglich ist, zu einem gegebenen Ausgabewert y einen Eingabewert x zu finden, den die Hashfunktion auf y abbildet: $h(x) = y$.

Diese Definition lässt offen, was mit »praktisch unmöglich« gemeint ist. Eine mathematisch exakte Definition ist für unsere Zwecke zu aufwendig, aber wir bemerken Folgendes:

Es ist *nicht* unmöglich, einen Eingabewert zu finden, der auf einen gegebenen Ausgabewert abgebildet wird. Eingabewerte sind Bitfolgen. Wir können also wie folgt alle Eingabewerte aufzählen:

(leere Bitfolge), 0, 1, 00, 01, 10, 11, 000, ... und so weiter.

Für jeden Eingabewert berechnen wir dabei seinen Hash und vergleichen diesen mit dem gegebenen Ausgabewert. Auf diese Weise finden wir mit Sicherheit irgendwann den passenden Eingabewert.

Aber wie praktikabel ist diese Methode? Für jeden Versuch liegt die Wahrscheinlichkeit, den richtigen Ausgabewert zu treffen, bei $2^{-256} \approx 10^{-77}$. Selbst wenn eine GPU[1] 10^{10} Hashwerte pro Sekunde be-

1 Graphical Processing Unit: die Recheneinheit einer Grafikkarte. Eine GPU ist im Allgemeinen schneller im Berechnen vieler Hashwerte als der Hauptprozessor (CPU) eines PCs.

rechnet und alle 10^{10} GPUs der Welt 100 Jahre lang rechnen (rund 10^{10} Sekunden), ist die Wahrscheinlichkeit, dass sie diesen Ausgabewert treffen, immer noch praktisch gleich null (10^{-47}).

»Praktisch unmöglich« in der obigen Definition heißt nun, dass kein wesentlich besseres Verfahren zum Finden eines passenden Eingabewerts bekannt ist als das soeben beschriebene.

Sofern Alice in unserem Beispiel eine Einwegfunktion benutzt, um den lokal gespeicherten Hashwert zu berechnen, kann sie sicher sein, dass niemand aus dem Hashwert den Dateiinhalt rekonstruieren kann. Wenn also zum Beispiel Charlie durch Zugriff auf Alice' Laptop Kenntnis dieses Hashwerts erlangt, kann Alice sicher sein, dass der Dateiinhalt trotzdem vor ihm geheim bleibt.

Achtung!

Wenn der Raum der möglichen Eingabewerte klein ist, dann ist es auch bei einer Einwegfunktion trivial, den entsprechenden Eingabewert zu finden! Bei dem Eingabewert »Satoshi Nakamoto« aus unserem Beispiel ist das leider der Fall, er besteht lediglich aus zwei Wörtern. Die Anzahl möglicher Wörter liegt nur in der Größenordnung von etwa 10^6, bei zwei Wörtern haben wir also 10^{12} Möglichkeiten. Eine einzelne GPU durchsucht diesen Raum vollständig in nur etwa 100 Sekunden! Einwegfunktionen sind also nur dann sinnvoll anwendbar, wenn ihre Eingabewerte aus einem genügend großen Raum stammen. In der Praxis werden deshalb meist Zufallswerte an die Eingabewerte angefügt, bevor Hashfunktionen darauf angewendet werden.

Definition: Kollisionsresistenz

Eine Hashfunktion h ist *kollisionsresistent*, wenn es praktisch unmöglich ist, zwei verschiedene Eingabewerte x und y zu finden, sodass $h(x) = h(y)$.

Ein Paar zweier solcher Eingabewerte x und y mit gleichem Ausgabewert wird auch *Kollision* genannt.

Ähnlich wie bei der Einwegfunktionseigenschaft ist es auch hier zwar einfach, einen Algorithmus anzugeben, der eine Kollision fin-

den kann, aber es ist kein Algorithmus bekannt, der bei sinnvoller Begrenzung der Rechenkapazität auch tatsächlich in nützlicher Zeit eine Kollision findet.

Kollisionsresistente Hashfunktionen sind im Allgemeinen auch Einwegfunktionen, aber nicht umgekehrt. Kollisionsresistent zu sein, ist also eine stärkere Anforderung an eine Hashfunktion, als eine Einwegfunktion zu sein. In wiederum anderen Worten: Die Kollisionsresistenz einer Hashfunktion zu brechen, ist einfacher, als ihre Einwegfunktionseigenschaft zu brechen. Intuitiv liegt das daran, dass ein Angreifer beim Finden einer Kollision mehr Freiheit hat als beim Finden eines zum Ausgabewert passenden Eingabewerts: Er kann beide Eingabewerte frei wählen.

Eine kollisionsresistente Hashfunktion lässt sich genau so zum Erkennen von Übertragungsfehlern anwenden wie eine beliebige Hashfunktion. Zusätzlich schützt eine kollisionsresistente Hashfunktion aber auch vor gezielten bösartigen Manipulationen. Wenn Bob Alice anstelle ihrer Datei eine Fälschung unterschieben will, muss die Fälschung ja denselben Hashwert haben wie Alice' Datei, sonst erkennt Alice die Fälschung. Dazu muss Bob also eine Kollision in der Hashfunktion finden – was aufgrund ihrer Kollisionsresistenz praktisch unmöglich ist.

Es ist eine spannende Frage, *wie* denn nun Hashfunktionen designt werden, damit sie die genannten Sicherheitseigenschaften erfüllen. Interessierte Leser seien dazu an [14] verwiesen. Wir gehen darauf hier nicht ein, denn wir brauchen nur den Fakt, *dass* sie diese Eigenschaften erfüllen.

Digitale Signaturen

Digitale Signaturen sollen die Eigenschaften von physischen Unterschriften auf Papier für digitale Dokumente nachbilden. Eine digitale Signatur ist grundsätzlich einfach eine Bitfolge, die vom Absender mithilfe eines Signaturschemas für eine Nachricht erzeugt wurde. Typischerweise wird diese Signatur an die Nachricht ange-

hängt und mit ihr verschickt, damit der Empfänger überprüfen kann, dass die Nachricht tatsächlich vom Absender stammt und nicht auf dem Übertragungsweg geändert wurde.

Ein Signaturschema besteht aus drei Funktionen:

1. `generate`: Erzeugt ein Schlüsselpaar, bestehend aus einem Signierschlüssel, der nötig ist, um Nachrichten zu signieren, sowie einem Verifikationsschlüssel, der nötig ist, um Signaturen zu überprüfen. Der Signaturschlüssel muss natürlich geheim gehalten werden. Der Verifikationsschlüssel wird typischerweise öffentlich bekannt gegeben. Deswegen heißt der Signaturschlüssel auch *geheimer Schlüssel* oder *private key*, und der Verifikationsschlüssel heißt *öffentlicher Schlüssel* oder *public key*.
2. `sign`: Erzeugt für eine gegebene Nachricht mit einem gegebenen Signaturschlüssel eine Signatur.
3. `verify`: Überprüft die Gültigkeit einer gegebenen signierten Nachricht bezüglich eines gegebenen Verifikationsschlüssels.

Eine Implementation eines Signaturschemas finden wir in der NaCl-Bibliothek [11, 15]. Alice und ihr Sekretär Bob werden diese Bibliothek nun von Python aus nutzen, um sicher miteinander zu kommunizieren.

Alice möchte Bob anweisen, einem Lieferanten Geld zu überweisen. Natürlich wird Bob diese Anweisung nur ausführen, wenn er sicher sein kann, dass sie von Alice stammt. Er wird sie insbesondere daher nicht ausführen, wenn er sie unsigniert per E-Mail erhält.

Alice muss also die Nachricht signieren. Zuerst erzeugt Alice ein Schlüsselpaar:

```
>>> import nacl.encoding
>>> import nacl.signing
>>> signing_key = nacl.signing.SigningKey.generate()
>>> verify_key = signing_key.verify_key
>>> verify_key.encode(encoder=nacl.encoding.HexEncoder)
b'2b03da5ee034ab1399d29e2535e2220d275fa3a879faee250ff666c599187414'
```

Den Signierschlüssel hält Alice geheim. Den Verifikationsschlüssel lässt sie Bob auf eine Weise zukommen, bei der Bob sicher sein

kann, dass er tatsächlich von Alice kommt. Sie kann ihn persönlich übergeben oder auch öffentlich zur Verfügung stellen, zum Beispiel auf der Webseite ihrer Firma. In jedem Fall muss Bob sichergehen, dass er wirklich den Verifikationsschlüssel von Alice erhält und sich nicht etwa einen anderen Verifikationsschlüssel unterschieben lässt.

Nun kann Alice eine Nachricht wie folgt signieren:

```
>>> message = "Bitte überweise 10000 EUR an Konto 0815"
>>> signed_message = signing_key.sign(message.encode())
>>> signed_message
b'K\x9d\xa7\xc9\x1d\xe1e\x00\x7f\x92\xeb\x04\xffx\xbcS\xd9Q\x8b]]
\x80/\x83K\\\x8b\x96w\xe2\xc0A\xb0v\xac\xe5\x14\xa6\x16h\xaaPl
\x11D\xe4w1\x14\xfa\xbe\xda\x94\xe6\xc0\xf7h3\xc6\xa2E*\xca\x05
Bitte \xc3\xbcberweise 10000 EUR an Konto 0815'
```

Die signierte Nachricht kann sie jetzt an Bob schicken, zum Beispiel per E-Mail. Bob muss nun anhand des Verifikationsschlüssels die Nachricht verifizieren:

```
>>> message_bytes = verify_key.verify(signed_message)
>>> message_bytes
b'Bitte \xc3\xbcberweise 10000 EUR an Konto 0815'
>>> message_bytes.decode()
'Bitte überweise 10000 EUR an Konto 0815'
>>>
```

Die Verifikation hat funktioniert. Bob ist sich also sicher, dass die Nachricht von Alice stammt, und er kann das Geld überweisen.

Wenn die Nachricht auf dem Weg modifiziert wurde, zum Beispiel von einem Angreifer, der das Geld auf sein eigenes Konto umlenken will, wirft die Funktion verify einen Ausnahmefehler:

```
>>> tampered = signed_message.replace(b'0815',b'4711')
>>> tampered
b'K\x9d\xa7\xc9\x1d\xe1e\x00\x7f\x92\xeb\x04\xffx\xbcS\xd9Q\x8b]]
\x80/\x83K\\\x8b\x96w\xe2\xc0A\xb0v\xac\xe5\x14\xa6\x16h\xaaPl
\x11D\xe4w1\x14\xfa\xbe\xda\x94\xe6\xc0\xf7h3\xc6\xa2E*\xca\x05
Bitte \xc3\xbcberweise 10000 EUR an Konto 4711'
>>> verify_key.verify(tampered)
Traceback (most recent call last):
...
nacl.exceptions.BadSignatureError: Signature was forged or corrupt
```

In diesem Fall überweist Bob das Geld also nicht, sondern informiert Alice darüber, dass jemand versucht, in ihrem Namen Nachrichten zu verschicken.

An ein Signaturschema werden zwei wesentliche Anforderungen gestellt: Korrektheit und Unfälschbarkeit. Dass ein Signaturschema *korrekt* ist, heißt einfach Folgendes: Wenn eine Nachricht mit einem bestimmten Signierschlüssel signiert wurde und dann diese signierte Nachricht mit dem zugehörigen Verifikationsschlüssel verifiziert wird, ist die Verifikation erfolgreich. Diese Anforderung ist offensichtlich. Ein nicht korrektes Signaturschema ist unbrauchbar – auch bei einer korrekt vom Absender signierten Nachricht könnte die Verifikationsfunktion dem Empfänger signalisieren, dass die Nachricht gefälscht ist.

Genauso wichtig ist die Unfälschbarkeit eines Signaturschemas. Intuitiv ist ein Signaturschema unfälschbar, wenn niemand außer dem Inhaber des Signaturschlüssels Nachrichten signieren kann. Bei genauerer Betrachtung müssen wir diese Bedeutung aber präzisieren. Erstens ist ja der Verifikationsschlüssel öffentlich bekannt – jeder kann also für eine beliebige Nachricht Signaturen so lange zufällig raten und anhand des Verifikationsschlüssels überprüfen, bis er eine richtige Signatur erraten hat. Genau wie bei den kryptografischen Hashfunktionen werden wir also nur verlangen, dass das Finden einer Signatur »praktisch unmöglich« ist. Zweitens kann der Empfänger einer signierten Nachricht natürlich trivialerweise eine Nachricht im Namen des Absenders signieren: Er kann ja einfach die empfangene Nachricht weiterverschicken. Bekannte signierte Nachrichten müssen wir also ausschließen. Damit kommen wir zu folgender Definition:

Definition: Unfälschbares Signaturschema

Wir nehmen einen Angreifer an, der den Verifikationsschlüssel kennt, der eine kleine Menge von signierten Nachrichten kennt, der aber den Signaturschlüssel nicht kennt. Ein Signaturschema ist *unfälschbar*, wenn es einem solchen Angreifer praktisch unmöglich ist, eine korrekt signierte Nachricht zu erzeugen – mit Ausnahme der ihm bekannten signierten Nachrichten.

Das vorgestellte Signaturschema aus der NaCl-Bibliothek ist korrekt und unfälschbar.

Sind Hashfunktionen und digitale Signaturen sicher?

Sicherheitsaussagen über kryptografische Funktionen spiegeln nur unseren derzeitigen Wissenstand über sie wider. Eine Hashfunktion gilt als kollisionsresistent, wenn *zurzeit* kein Algorithmus bekannt ist, der mit *derzeitig* verfügbarer Rechenkapazität erlauben würde, in nützlicher Zeit Kollisionen zu finden. Genauso gilt ein Signaturschema als unfälschbar, wenn *zurzeit* kein Algorithmus bekannt ist, der einem Angreifer mit *derzeitig* verfügbarer Rechenkapazität erlauben würde, Nachrichten in fremdem Namen zu signieren. Beides kann sich mit der Zeit ändern. Mit besserem Verständnis von kryptografischen Funktionen werden Attacken möglich, die weniger Rechenkapazität benötigen. Früher sichere kryptografische Hashfunktionen, wie z. B. MD5 und SHA-1, sind mittlerweile angreifbar. Auch SHA-256 wird möglicherweise irgendwann angreifbar, weshalb bereits Nachfolgekandidaten bereitstehen [20]. Erfahrungsgemäß werden weitverbreitete kryptografische Funktionen nicht überraschend von heute auf morgen gebrochen, sondern es werden mit der Zeit immer effizientere (aber immer noch nicht praktikable) Angriffe auf sie bekannt. Es sollte dann also genug Zeit sein, um auf sicherere Funktionen umzusteigen.

Digitales Bargeld

Eine Bargeldzahlung hat gegenüber einer elektronischen Zahlung den Vorteil, dass sie direkt vom Zahlenden an den Zahlungsempfänger geht. Es gibt also keine dritte Partei, die zwischen den beiden Parteien steht, wie etwa einen Zahlungsdienstleister. Ein solcher Zahlungsdienstleister hat prinzipiell die Möglichkeit, die Zahlung zu blockieren, Gebühren dafür zu erheben oder die Zahlung im Nachhinein rückgängig zu machen. Auch besteht die Gefahr, dass der Zahlungsdienstleister selbst insolvent wird. Daher kommt das Interesse an einem digitalen Äquivalent zu Bargeld.

Mit der Hilfe von digitalen Signaturen können wir nun schon fast digitales Bargeld schaffen. Zwei Hauptanforderungen an Bargeld sind:

1. Jeder kann die Echtheit von erhaltenem Bargeld überprüfen, und
2. nur der Herausgeber (z. B. die Zentralbank) darf Bargeld erschaffen.

Eine vom Herausgeber digital signierte Nachricht, wie etwa:

Diese Geldnote hat Seriennummer 0815 und den Wert von EUR 100.

erfüllt beide Anforderungen. Da der Verifikationsschlüssel des Herausgebers öffentlich bekannt gemacht wird, kann jeder die Gültigkeit der Signatur überprüfen, und da das Signaturschema fälschungssicher ist, kann niemand außer dem Herausgeber Geldnoten erzeugen.

Es gibt nur ein Problem: Im Gegensatz zu physischem Bargeld lassen sich digitale Nachrichten beliebig kopieren. Wenn Alice also die besagte digitale Geldnote im Wert von 100 EUR erhalten hat, kann sie einfach eine Kopie davon an Bob und eine zweite Kopie an Charlie weiterverschicken und so Waren im Wert von 200 EUR einkaufen. Davor schützt uns das Signaturschema nicht: Die Verifikation der Nachricht gelingt sowohl Bob als auch Charlie. Das ist das berühmte *Double-Spending-Problem* [19].

Natürlich kann man das Problem lösen, indem auf einem zentralen Server Buch über diese Geldnoten geführt wird und die Zahlungsempfänger ihre Zahlungseingänge überprüfen ... aber damit ist der Betreiber dieses Servers ein Zahlungsdienstleister, und die Vorteile von digitalem Bargeld sind dahin.

Das Double-Spending-Problem blieb lange ungelöst: Erst 2008 publizierte Satoshi Nakamoto im Bitcoin-Whitepaper eine Lösung dafür [12]. Das zentrale Konzept in dieser Lösung ist die Blockchain, die wir im nächsten Kapitel entwickeln werden.

Die Blockchain löst das Double-Spending-Problem, indem sie eine zeitliche Reihenfolge von Transaktionen festlegt. Bei zwei wider-

sprüchlichen Transaktionen wird so die erste Transaktion akzeptiert und die zweite verworfen.

Die Blockchain ist also eine Art Zeitstempelmechanismus. In frühen Versionen der Bitcoin-Software hieß die Blockchain auch *timechain*.[2] Deshalb schauen wir uns im folgenden Abschnitt zunächst an, was ein digitaler Zeitstempel ist.

Digitale Zeitstempel

Digitale Zeitstempel (englisch: timestamps) dienen dazu, die Existenz von bestimmten Daten zu einer bestimmten Zeit nachzuweisen.

Ein Dokument aus Papier kann man beispielsweise in einem Briefumschlag per Post an sich selbst schicken. Auf diese Weise kann man später beweisen, dass das Dokument bereits am Datum des Poststempels existiert hat – zumindest gegenüber jenen, die der Unversehrtheit des Umschlags und der Echtheit des Poststempels vertrauen.

Auch digitale Dokumente kann man mit Zeitstempeln versehen. Das ist häufig sinnvoll, wenn Software oder digitale Kunstwerke veröffentlicht werden. Man kann damit vermeiden, dass später andere Parteien Urheberschaft vorgeben und Copyright oder Patente für sich beanspruchen.

Nehmen wir an, Alice hat bei AirBnB eine Wohnung von Vera gemietet. Sie kommt in der Wohnung an und sieht, dass das Parkett beschädigt ist. Vera ist gerade nicht erreichbar. Um nicht selbst in Verdacht zu geraten, fotografiert Alice sofort den Schaden und lässt sich für das Foto einen Zeitstempel ausstellen. Der Zeitstempel beweist, dass das Foto schon bei ihrer Ankunft entstanden ist und nicht erst danach. So kann sie nachweisen, dass sie den Schaden nicht verursacht hat.

2 main.cpp, Zeile 1681, *https://github.com/bitcoin/bitcoin/archive/v0.3.2.zip*

Trusted Timestamping. Zeitstempel werden von Timestamp-Servern ausgestellt. Diese werden von vertrauenswürdigen Stellen betrieben, zum Beispiel von einem großen Telekommunikationsanbieter oder einer Regierungsbehörde. In unserem Beispiel betreibt Tim einen Timestamp-Server. Tim ist im Besitz eines Schlüsselpaars, das er sich mit der Funktion `generate` eines Signaturschemas erzeugt hat. Den Signaturschlüssel hält Tim geheim. Den Verifizierungsschlüssel hat er öffentlich bekannt gegeben.

Um einen Zeitstempel für ihr Foto zu bekommen, schickt Alice den Hash des Fotos an den Server. Der Server fügt das aktuelle Datum sowie die aktuelle Zeit hinzu und signiert das Resultat. Das ergibt dann den Zeitstempel, den der Server zurück an Alice schickt. Alice speichert diesen und kann später damit beweisen, dass das Foto am Datum des Zeitstempels existiert hat. Die Ausstellung eines Zeitstempels ist in Abbildung 1-1 dargestellt.

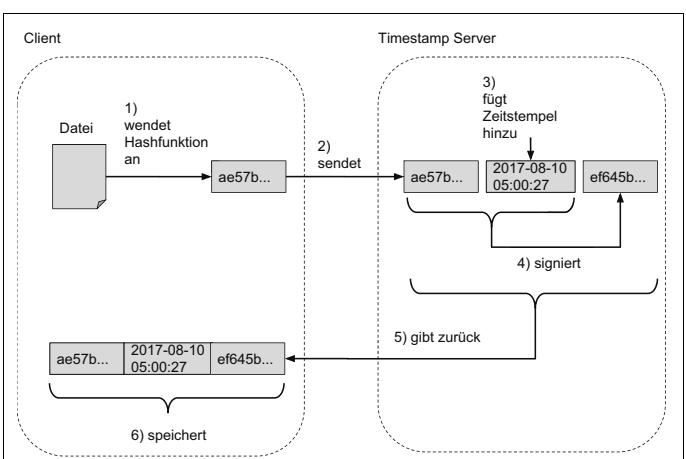

Abbildung 1-1: Ausstellung eines Zeitstempels im Trusted Timestamping

Prinzipiell funktioniert es auch, wenn Alice direkt ihr Foto an den Timestamp-Server schickt (und nicht nur einen Hash davon). Das hat aber den Nachteil, dass sie damit das Foto gegenüber Tim offenlegt. Prinzipiell geht der Zustand von Veras Parkett Tim nichts

an. Da die Hashfunktion eine Einwegfunktion ist, kann Tim das Foto nicht ohne Weiteres aus dem Hash rekonstruieren. Außerdem ist es natürlich effizienter, nur den Hash zu schicken, da dieser meist deutlich kürzer ist als das Foto selbst.

Wenn Vera später den erzeugten Zeitstempel überprüft, dann prüft sie Folgendes:

- Ist der Hashwert von Alice' Foto gleich dem im Zeitstempel enthaltenen Hashwert?
- Ist die Signatur des Timestamp-Servers echt? (Sie ruft also die Funktion `verify` des Signaturschemas auf.)

Wenn diese Prüfungen erfolgreich sind, geht sie davon aus, dass das Foto zum im Zeitstempel angegebenen Zeitpunkt existiert hat und dass demzufolge Alice den Schaden nicht verursacht haben kann.

Implizit vertraut Vera damit

- der Unfälschbarkeit des verwendeten Signaturschemas (denn sonst könnte Alice sich selbst den Zeitstempel ausgestellt haben),
- der Kollisionsfreiheit der Hashfunktion (denn sonst könnte Alice ihr ein anderes Foto vorlegen als das, dessen Hash sie damals zum Timestamp-Server geschickt hat) sowie
- der Fähigkeit und dem Willen von Tim,
  - immer die richtige Zeit in seinen Zeitstempeln einzutragen und
  - seinen Signaturschlüssel geheim zu halten.

Aufgrund des Vertrauens, das Vera Tim entgegenbringen muss, heißt diese Timestamping-Methode *Trusted Timestamping*. Das Wort *trust* (deutsch: Vertrauen) klingt positiv, bezeichnet hier aber nicht etwa *gegebenes* Vertrauen, sondern *erfordertes* Vertrauen. Das Verfahren erfordert, dass Vera Tim vertraut. Dieses Vertrauen kann verletzt werden, z. B. wenn Tim von Alice ein Bestechungsgeld annimmt und ihr dafür einen rückdatierten Zeitstempel ausstellt. Das geforderte Vertrauen ist also ein Nachteil des Verfahrens.

Das Vertrauen in die Geheimhaltung des Signaturschlüssels ist besonders problematisch. Sobald ein Angreifer in den Besitz des

Signaturschlüssels kommt, kann er Zeitstempel für beliebige Dokumente mit beliebigem Datum ausstellen. Im Fall der Kompromittierung des Signaturschlüssels verlieren also alle existierenden Zeitstempel auf einen Schlag ihre Beweiskraft!

Um dieses Problem zu lösen, wurde *Linked Timestamping* [9] entwickelt. Die wesentliche Idee dahinter ist einfach: Der Timestamping-Server fügt bei jedem ausgestellten Timestamp einen Hash des vorhergehenden Timestamps hinzu.

Linked Timestamping. Der Ablauf mehrerer Anfragen an einen Timestamp-Server mit Trusted Timestamping ist in Abbildung 1-2 dargestellt. Alice sendet einen Hash y_1 an den Timestamp-Server, der Server konkateniert[3] ihn mit der aktuellen Zeit t_1, wendet die Funktion `sign` des Signaturschemas darauf an und schickt den resultierenden Zeitstempel $sign(y_1||t_1)$ zurück. Das Gleiche passiert für Bob, Charlie und so weiter.

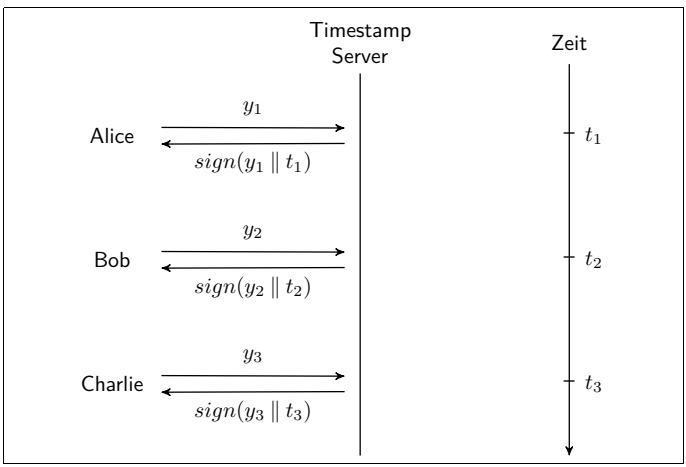

Abbildung 1-2: Trusted Timestamping

3 Zwei Strings werden konkateniert, indem sie hintereinandergeschrieben werden. Ist beispielsweise a der String "Hello" und b der String "World", so ist ihre Konkatenation $a||b$ der String "HelloWorld"

Wir sehen insbesondere, dass keinerlei Zusammenhang zwischen den zurückgegebenen Zeitstempeln besteht. Ganz anders im Linked Timestamping aus Abbildung 1-3: Darin beinhaltet jeder Zeitstempel den Hash des vorherigen Zeitstempels. Mit h ist hier eine kryptografische Hashfunktion bezeichnet.

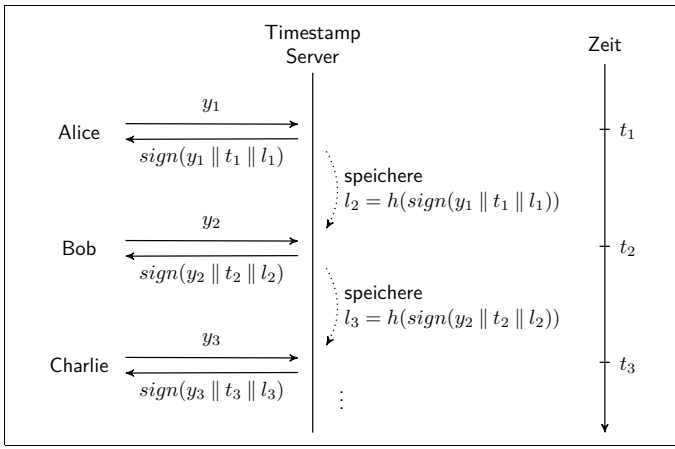

Abbildung 1-3: Linked Timestamping

Der resultierende Hashwert wird hier mit l_i bezeichnet, weil er als *Link*, also als Verbindung zwischen den Zeitstempeln, fungiert. Für den Link l_1 des ersten Zeitstempels wird einfach der leere String verwendet.

Die Einbettung des Hashwerts des vorigen Timestamps im aktuellen Timestamp legt die zeitliche Reihenfolge der beiden Timestamps fest: Es steht fest, dass der vorige Timestamp vor dem aktuellen Timestamp existiert haben muss – sonst hätte der Timestamp-Server seinen Hashwert nicht berechnen können. Das ist völlig unabhängig von den Zeiten, die in den Timestamps eingetragen sind. Auch wenn im aktuellen Timestamp, beispielsweise durch einen Fehler, eine frühere Zeit als im vorigen Timestamp eingetragen ist, so ist die wahre Reihenfolge der Timestamps leicht zu erkennen.

Nehmen wir nun an, dass Tims Timestamp-Server Linked Timestamps ausstellt und dass Alice sich von ihm einen Timestamp hat ausstellen lassen, so wie in Abbildung 1-3 dargestellt. Vera will nun den Timestamp von Alice überprüfen, aber sie vertraut Tim nicht. Was kann sie tun?

Sie fordert den nächsten ausgestellten Timestamp an, also den von Bob. Wenn Vera verifizieren kann, dass der Hash des ersten Timestamps im zweiten enthalten ist, und wenn sie Bob vertraut, kann sie immerhin davon ausgehen, dass Alice' Dokument vor dem Zeitpunkt t_2 existierte. Und wenn sie Bob nicht vertraut, fordert sie den Timestamp von Charlie an. Wenn sie verifizieren kann, dass der Hash von Bobs Timestamps in Charlies Timestamp enthalten ist und sie Charlie vertraut, kann sie immerhin noch davon ausgehen, dass Alice' Dokument vor dem Zeitpunkt t_3 existierte – und so weiter.

Noch nützlicher ist diese Methode, wenn Vera prinzipiell Tim vertraut, jedoch zu einem bestimmten bekannten Zeitpunkt Tims Signaturschlüssel kompromittiert wurde. Sagen wir, der Signaturschlüssel sei nach Zeitpunkt t_2 kompromittiert worden. Wenn Vera Bob vertraut und Bob ihr versichert, dass er seinen Timestamp zum Zeitpunkt t_2 erhalten hat, und wenn sie verifiziert, dass der Hash von Alice' Timestamp in Bobs Timestamp enthalten ist, dann weiß sie, dass Alice' Dokument zum Zeitpunkt t_1 bereits existierte – denn der Timestamp wurde ja zu einem Zeitpunkt signiert, als der Schlüssel noch nicht kompromittiert war.

Linked Timestamps bilden eine Kette – oder eine Hash-verkettete Liste. Jeder Timestamp legt durch die Einbettung des Hashwerts den Inhalt des vorigen Timestamps fest. Und wenn der aktuelle Timestamp den vorigen Timestamp festlegt und auch der vorige Timestamp den vorvorigen Timestamp festlegt und so weiter, dann legt jeder Timestamp alle vorigen Timestamps fest.

Wenn im Nachhinein in dieser Liste etwas geändert, gelöscht oder eingefügt wird, kann diese Manipulation durch die Überprüfung der Hashwerte erkannt werden. Und mit jedem neu ausgestellten Timestamp wird die Fälschungssicherheit der bereits ausgestellten Timestamps erhöht.

Genau wie eine Folge von Linked Timestamps, so ist auch eine Blockchain eine Hash-verkettete Liste: Jeder Block beinhaltet den Hash des vorherigen Blocks. Es gibt sogar Literatur, die jede Hash-verkettete Liste als Blockchain bezeichnet. Das kann jedoch zu Verwirrung führen: Hash-verkettete Listen gibt es schon lange, vermutlich seit den Siebzigerjahren. Sie allein können das Double-Spending-Problem nicht lösen. Dazu brauchen wir eine weitere wichtige Komponente: Proof-of-Work.

Proof-of-Work

Als Proof-of-Work bezeichnet man einen Beweis dafür, dass man eine bestimmte Arbeit geleistet hat. Eine Anwendung davon ist das Hashcash-Protokoll, ein Protokoll zur Bekämpfung von Spam [3]. Die grundlegende Idee in Hashcash ist, dass ein Empfänger einer E-Mail diese nur annimmt, wenn sie einen Beweis dafür enthält, dass der Absender eine gewisse Arbeit beim Erstellen der E-Mail geleistet hat. Die Arbeit besteht darin, eine Rechenaufgabe zu lösen, zu deren Lösung beispielsweise etwa eine Sekunde auf derzeit aktueller Hardware benötigt wird. Einem normalen Anwender macht es nichts aus, beim Verschicken einer Nachricht eine Sekunde zu warten. Einem Spammer ist es so aber nicht ohne Weiteres möglich, Millionen von Nachrichten zu verschicken.

Welche Art Rechenaufgabe wird in Hashcash verwendet? Es gibt ein paar Anforderungen: Sie muss verhältnismäßig schwierig zu lösen sein – mit anpassbarer Schwierigkeit, da sich Hardware ständig verbessert. Sie muss von der E-Mail-Adresse des Empfängers abhängen, damit ein Spammer nicht mit einer Lösung mehreren Empfängern Spam schicken kann. Außerdem muss der in der E-Mail enthaltene Beweis leicht vom Spamfilter des Empfängers überprüft werden können.

Die Aufgabe besteht im Wesentlichen darin, eine Bitfolge zu finden, die, wenn sie mit der E-Mail-Adresse des Empfängers konkateniert und einer kryptografischen Hashfunktion übergeben wird, einen Ausgabewert liefert, der mit einer bestimmten Anzahl Nullen anfängt. Genauer:

Definition: Hashcash-Puzzle

Gegeben sei eine Hashfunktion h, eine Bitfolge p (der Puzzle-String) sowie eine natürliche Zahl d (für difficulty). Das *Hashcash-Puzzle* ist dann die Aufgabe, eine Bitfolge s (für solution) zu finden, sodass

$h(p||s)$

mit d Nullen anfängt.

Ähnlich wie bei den besprochenen Sicherheitseigenschaften kryptografischer Hashfunktionen ist auch für das Hashcash-Puzzle mit aktuellen Hashfunktionen keine bessere Methode zur Lösung bekannt, als systematisch Werte für s auszuprobieren.

Wir wollen Satoshi Nakamoto unter `satoshin@gmx.com` eine E-Mail schicken. Satoshis Spamfilter nutzt Hashcash, um Spam abzuwehren. Um den Spamfilter zu überwinden, müssen wir also ein Hashcash-Puzzle lösen und dessen Lösung im E-Mail-Header eintragen.

Zur Lösung des Puzzles übergeben wir dem Hashcash-Tool den Befehl `-m` (das steht für *mint* bzw. *prägen* in Anlehnung an das Prägen einer Münze) sowie die E-Mail-Adresse und bekommen als Resultat ein sogenanntes Hashcash-Token, das die Puzzle-Lösung beinhaltet:

```
$ hashcash -m satoshin@gmx.com
hashcash token: 1:20:170710:satoshin@gmx.com::lQeWK51q1JcTwphK:01l/x
```

Was ist hier im Detail passiert? Zuerst hat Hashcash den Puzzle-String erzeugt. Er besteht aus folgenden mit Doppelpunkt separierten Daten:

- Hashcash-Version: `1`
- Anzahl Null-Bits im Hash »difficulty«: `20`
- Datum: `170710` (10.07.2017)
- E-Mail-Adresse: `satoshin@gmx.com`
- eine Base64-codierte Zufallszahl: `lQeWK51q1JcTwphK`

Dann hat Hashcash Rechenarbeit geleistet: Es hat das durch den Puzzle-String und die Difficulty definierte Puzzle gelöst. Es hat, angefangen bei null, laufend einen Zähler s inkrementiert und den

Puzzle-String, konkateniert mit *s*, so lange gehasht, bis der resultierende Hashwert mit 20 Null-Bits begann.

Dann hat es das gefundene *s* Base64-codiert:

- `O1l/x`

und an den Puzzle-String angefügt. Das Resultat ist das Hashcash-Token. Das Token ist Proof-of-Work: Indem wir es vorweisen, beweisen wir, dass wir die Rechenarbeit geleistet haben, die nötig war, es zu finden.

Mit der Standardschwierigkeit von 20 Bits sind Hashcash-Puzzles sehr schnell zu lösen (unter einer Sekunde). Wir erhöhen die Schwierigkeit des Puzzles etwas:

```
$ hashcash -m -b 25 satoshin@gmx.com
hashcash token: 1:25:170710:satoshin@gmx.com::Tyr6iRYL7BeQFnij:1C2vf
```

Das dauert auf aktueller Hardware im Durchschnitt ein paar Sekunden. Wir tragen das Token in den Header unserer Nachricht an Satoshi ein und schicken die Nachricht ab.

Satoshis Spamfilter wird dann mit der Hashcash-Funktion `-c` (check) die Gültigkeit des Hashcash-Tokens überprüfen und feststellen, dass der Absender tatsächlich die entsprechende Rechenarbeit geleistet hat:

```
$ hashcash -c -y 1:25:170710:satoshin@gmx.com::Tyr6iRYL7BeQFnij:1C2vf
matched token: 1:25:170710:satoshin@gmx.com::Tyr6iRYL7BeQFnij:1C2vf
check: ok
```

Statt Hashcash zu benutzen, könnte Satoshis Spamfilter auch einfach die Hashfunktion selbst auf das Token anwenden, um sich zu überzeugen, dass der Hash mit der entsprechenden Anzahl Null-Bits anfängt:

```
$ echo -n 1:25:170710:satoshin@gmx.com::Tyr6iRYL7BeQFnij:1C2vf\
| openssl sha1
(stdin)= 00000031ed7c1d3e78b1723e6a70a9948be408f2
```

Die Option `-n` ist nötig, um die Ausgabe eines Zeilenumbruchs zu verhindern. SHA-1 ist die in Hashcash verwendete Hashfunktion – ein Vorgänger von SHA-256. In der Hexadezimaldarstellung des

Hashwerts sehen wir sechs Nullen, entsprechend mindestens 24 Nullen in der Binärdarstellung.

Es ist also sehr einfach, die Lösung eines Hashcash-Puzzles zu überprüfen: Es ist lediglich ein Hash zu berechnen und die Anzahl der Null-Bits zu zählen.

Andererseits kann die Schwierigkeit, ein Hashcash-Puzzle zu lösen, über einen sehr weiten Bereich variiert werden. Die durchschnittliche Rechenarbeit, die zur Lösung eines Puzzles nötig ist, steigt exponentiell mit der Anzahl geforderter Null-Bits des Puzzles. Um das zu verdeutlichen, nutzen wir die Speedtest-Funktion von Hashcash, die die nötige Rechenzeit für eine bestimmte Schwierigkeit abschätzt:

```
$ hashcash -s -b 20
time estimate: 0 seconds (152   milli-seconds)

$ hashcash -s -b 30
time estimate: 159 seconds (3 minutes)

$ hashcash -s -b 40
time estimate: 158914 seconds (1.8 days)

$ hashcash -s -b 50
time estimate: 162727743 seconds (5.2   years)

$ hashcash -s -b 60
time estimate: 166633208344 seconds (5284 years)
```

Zum Vergleich: Das Bitcoin-Netzwerk löst derzeitig durchschnittlich aller zehn Minuten ein derartiges Puzzle mit einer Schwierigkeit von mehr als 70 Bits (Stand Anfang 2017). Wir werden später noch darauf eingehen, warum es das tut.

Mit kryptografischen Hashfunktionen, digitalen Signaturen, Linked Timestamping und Proof-of-Work haben wir jetzt das nötige kryptografische Rüstzeug, um uns im nächsten Kapitel der Blockchain zuzuwenden.