Datensicherung, ja. Aber wie?

Juni 2020

In die Cloud?

In die Cloud?

Wohl nicht unbedingt eine gute Idee. Fallstudie:

»Apple bewahrt iCloud-Backups nur sechs Monate auf«.

»iPhone- und iPad-Backups werden 180 Tage nach der letzten Datensicherung still aus iCloud gelöscht – auch bei zahlenden Abonnenten.«

In: Heise online (12. Juni 2020).

https://www.heise.de/news/Apple-bewahrt-iCloud-Backups-nur-sechs-Monate-auf-4782483.html

Ein Backup-System lässt sich einteilen in eine niedere und eine höhere Ebene.

Ein Backup-System lässt sich einteilen in eine niedere und eine höhere Ebene.

Niedere Ebene – Umgang mit den eigentlichen Daten:

- Verpackung, ggf. Datenkompression
- Schutz vor Verlust durch Redundanz (Duplikate)
- Schutz der Integrität
- Wahrung der Vertraulichkeit

Ein Backup-System lässt sich einteilen in eine niedere und eine höhere Ebene.

Niedere Ebene – Umgang mit den eigentlichen Daten:

- Verpackung, ggf. Datenkompression
- Schutz vor Verlust durch Redundanz (Duplikate)
- Schutz der Integrität
- Wahrung der Vertraulichkeit

Höhere Ebene – höhere logische Prozesse:

- Sicherungsarten, Deduplikation
- Versionsverwaltung
- Integrität der Verwaltung

Wir wollen uns hier auf die niedere Ebene beschränken.

Wir wollen uns hier auf die niedere Ebene beschränken.

Klassisches Anliegen: private Langzeitarchivierung.

Auftakt — Poor Man’s Backup

Naiver Ansatz: Schreibe Verzeichnis auf einen Datenträger.

Naiver Ansatz: Schreibe Verzeichnis auf einen Datenträger.

Probleme:

- Fremdes Dateisystem soll Metadaten nicht anfassen.
- Einige Dateisysteme unterstützen keine langen Dateinamen bzw. kein Unicode.
- Bestimmte Programme erfordern die Verpackung zu einem Paket.

Naive Lösung: Verpacke Verzeichnis als »`Paket.tar.xz`«.

Naive Lösung: Verpacke Verzeichnis als »`Paket.tar.xz`«.

Problem:

- Schon ein beschädigtes Bit kann dazu führen, dass sich das Archiv nicht mehr entpacken lässt.

Verpacke ohne Kompression als »`Paket.tar`«.

Verpacke ohne Kompression als »`Paket.tar`«.

Problem:

- Beschädigung fällt nicht auf.

Berechne kryptografischen Hashwert des Pakets.

Berechne kryptografischen Hashwert des Pakets.

Problem:

- Erlaubt keine Rückschlüsse auf Ort und Umfang der Beschädigungen.

Verpacke als »Paket.zip«.

Das ZIP-Format schützt vor Datenverlust, indem es

- für jede Datei einen CRC-32-Prüfwert speichert,
- das Inhaltsverzeichnis zweimal enthält.

Verpacke als »Paket.zip«.

Das ZIP-Format schützt vor Datenverlust, indem es

- für jede Datei einen CRC-32-Prüfwert speichert,
- das Inhaltsverzeichnis zweimal enthält.

Probleme:

- Prüfwert nur pro Datei, daher ungenau.
- Leider kein Prüfwert der komprimierten Daten.
- Eventuell Komplikationen, falls der Teil mit dem Inhaltsverzeichnis verloren geht.

Ansatz: Das Tar-Format benutzen und die Hashwerte aller Dateien *vor* dem Verpacken berechnen lassen und anbeifügen.

Ansatz: Das Tar-Format benutzen und die Hashwerte aller Dateien *vor* dem Verpacken berechnen lassen und anbeifügen.

Verbleibendes Problem:

- Lokalisierung von Beschädigungen nicht genau genug. Betrachte z. B. sehr große Dateien.

Kurze Pause

Was für Beschädigungen können eigentlich auftreten?

Was für Beschädigungen können eigentlich auftreten?

Das sind:

- Uniformer Bit-Rot: Verlust einzelner Bits oder Bytes, gleichverteilt über den gesamten Datenträger. Thermisches Rauschen, allgegenwärtige Ladungslecks, etc.
- Burst-Fehler: Verlust mehrerer Bits oder Bytes hintereinander.
- Wegfall ganzer Blöcke.
- Zerstörung des gesamten Datenträgers, z. B. durch Wegfall zentraler Steuerungseinheiten.
- Verschiebung: Daten an falscher Stelle, z. B. durch Synchronisationsfehler.

Betrachten wir zunächst den uniformen Bit-Verlust. Angenommen, ein Byte ist nach 10 Jahren mit einer sehr hohen Wahrscheinlichkeit p, sagen wir $p = 0.999999$, noch intakt, d. h. das Risiko für Beschädigung beträgt 1 zu 1 Mio.

Betrachten wir zunächst den uniformen Bit-Verlust. Angenommen, ein Byte ist nach 10 Jahren mit einer sehr hohen Wahrscheinlichkeit p, sagen wir $p = 0.999999$, noch intakt, d. h. das Risiko für Beschädigung beträgt 1 zu 1 Mio.

Bei 1 MB Daten hat man $N = 10^6$ Bytes und nur noch eine Wahrscheinlichkeit $p' = p^N \approx 0.37$ für die Intaktheit aller Daten. Das ist schon ein Risiko von ca. 2/3.

Betrachten wir zunächst den uniformen Bit-Verlust. Angenommen, ein Byte ist nach 10 Jahren mit einer sehr hohen Wahrscheinlichkeit p, sagen wir $p = 0.999999$, noch intakt, d. h. das Risiko für Beschädigung beträgt 1 zu 1 Mio.

Bei 1 MB Daten hat man $N = 10^6$ Bytes und nur noch eine Wahrscheinlichkeit $p' = p^N \approx 0.37$ für die Intaktheit aller Daten. Das ist schon ein Risiko von ca. 2/3.

Man beachte: Komprimierte Bilder können z. B. schon bei einem einzigen Bitfehler unlesbar werden.

Angenommen wir haben nun zwei Duplikate einer Datei, beide sind jedoch beschädigt.

Angenommen wir haben nun zwei Duplikate einer Datei, beide sind jedoch beschädigt.

Die Hashwerte nützen uns zunächst nichts.

Angenommen wir haben nun zwei Duplikate einer Datei, beide sind jedoch beschädigt.

Die Hashwerte nützen uns zunächst nichts.

Mit einer gewissen Wahrscheinlichkeit kann man zumindest alle Beschädigungen lokalisieren, – nämlich dann wenn die Daten an den Stellen nicht mehr übereinstimmen. Allerdings gibt es selbst dann irgendwann eine kombinatorische Explosion an Möglichkeiten für die Fehlerkorrektur.

Dieser Problematik kann man leicht ausweichen. Wir fügen jedem Datenblock einen Paritätswert hinzu. Eignen tut sich dafür ein CRC-32-Prüftwert (32 Bit lang) oder ein kryptografischer Hashwert (128 bis 256 Bit lang).

Dieser Problematik kann man leicht ausweichen. Wir fügen jedem Datenblock einen Paritätswert hinzu. Eignen tut sich dafür ein CRC-32-Prüftwert (32 Bit lang) oder ein kryptografischer Hashwert (128 bis 256 Bit lang).

Pro 256 Byte ein CRC-32-Prüfwert vergrößert den Speicherbedarf lediglich um $4/256 \approx 1.6\,\%$.

Dieser Problematik kann man leicht ausweichen. Wir fügen jedem Datenblock einen Paritätswert hinzu. Eignen tut sich dafür ein CRC-32-Prüftwert (32 Bit lang) oder ein kryptografischer Hashwert (128 bis 256 Bit lang).

Pro 256 Byte ein CRC-32-Prüfwert vergrößert den Speicherbedarf lediglich um $4/256 \approx 1.6\,\%$.

Ermöglicht bei geringen Beschädigungen sogar eine Fehlerkorrektur ohne ein Duplikat.

Man beachte: CRC-32 erkennt einen größeren Fehler nur mit einer Wahrscheinlichkeit von höchstens $1 - \frac{1}{2^{32}}$.

Man beachte: CRC-32 erkennt einen größeren Fehler nur mit einer Wahrscheinlichkeit von höchstens $1 - \frac{1}{2^{32}}$.

$\rightarrow$ Fasse eine feste Zahl an Blöcken zu einem Bündel zusammen und berechne für dieses zusätzlich einen kryptografischen Hashwert. Bei einer idealen Hashfunktion wird ein Fehler dann mit einer Wahrscheinlichkeit $p \approx 1 - \frac{1}{2^{256}}$ erkannt. Man darf nun $p = 1$ setzen.

Die Integrität ist nun mehr oder weniger geschützt, die Vertraulichkeit jedoch nicht. Verteilt man Datenträger mit Duplikaten auf unterschiedliche Orte, nimmt die Wahrscheinlichkeit für einen Diebstahl zu.

Die Integrität ist nun mehr oder weniger geschützt, die Vertraulichkeit jedoch nicht. Verteilt man Datenträger mit Duplikaten auf unterschiedliche Orte, nimmt die Wahrscheinlichkeit für einen Diebstahl zu.

Schlimmer: Man bemerkt nicht, wenn jemand in Besitz der Daten gelangt ist.

Die Integrität ist nun mehr oder weniger geschützt, die Vertraulichkeit jedoch nicht. Verteilt man Datenträger mit Duplikaten auf unterschiedliche Orte, nimmt die Wahrscheinlichkeit für einen Diebstahl zu.

Schlimmer: Man bemerkt nicht, wenn jemand in Besitz der Daten gelangt ist.

→ Daten vor dem Speichern verschlüsseln.

Die Integrität ist nun mehr oder weniger geschützt, die Vertraulichkeit jedoch nicht. Verteilt man Datenträger mit Duplikaten auf unterschiedliche Orte, nimmt die Wahrscheinlichkeit für einen Diebstahl zu.

Schlimmer: Man bemerkt nicht, wenn jemand in Besitz der Daten gelangt ist.

→ Daten vor dem Speichern verschlüsseln.

Problem:

- Blockchiffre verträgt sich in einigen Betriebsmodi nicht mit Fehlerkorrektur.

Lösung:

- Benutze eine Stromchiffre,
- oder Blockchiffre im Counter-Mode.

Lösung:

- Benutze eine Stromchiffre,
- oder Blockchiffre im Counter-Mode.

→ Schlüsselstrom wird mit den Klardaten bitweise XOR-verknüpft. Folglich führen fehlerhafte Bits im Chiffrat lediglich zu entsprechenden fehlerhaften Bits in den Klardaten. Verträgt sich mit der Fehlerkorrektur. Verträgt sich auch mit der Korrektur von Verschiebungsfehlern: Schlüsselstrom z. B. einfach solange verschieben, bis der CRC-32-Prüfwert eines Datenblocks stimmt.

Vorsicht: Die Nonce darf nicht abhanden kommen.

Vorsicht: Die Nonce darf nicht abhanden kommen.

Idee: Einmalschlüssel anstelle von Nonce benutzen.

Vorsicht: Die Nonce darf nicht abhanden kommen.

Idee: Einmalschlüssel anstelle von Nonce benutzen.

Leider heikel. Idee: Einmalschlüssel bei der Verschlüsselung von einem kryptografischen Zufallszahlengenerator erzeugen lassen.

Welche Chiffre konkret?

Z. B. die Stromchiffre ChaCha20 von Daniel Bernstein. Ein ziemlich minimalistisches Verfahren (der eigentliche Algorithmus beträgt weniger als 100 Zeilen Rust oder Python).

Ende.