Fortgeschrittene Techniken der Kryptographie

Jürgen Fuß

Episode 11: Merkle Trees und One-Time Signatures

HAGENBERG | LINZ | STEYR | WELS

FH OBERÖSTERREICH
UNIVERSITY OF APPLIED SCIENCES UPPER AUSTRIA

Merkle Trees

UNIVERSITY OF APPLIED SCIENCES UPPER AUSTRIA

Merkle Trees

Die Idee von Merkle Trees ist, für eine sehr große Anzahl an Datensätzen einen Hashwert zu berechnen, so dass damit die Integrität jedes einzelnen Datensatzes mit wenig Aufwand überprüft werden kann.
Verwendet werden dazu **binäre Bäume**, d. h. jeder Knoten im Baum, der kein Blatt ist, hat genau zwei Kinder.
Der Einfachheit halber gehen wir hier davon aus, dass die Anzahl der Datensätze eine Potenz von 2 ist. In diesem Fall lassen sich Merkle Trees besonders schön bauen.

Erstellen eines Merkle Tree

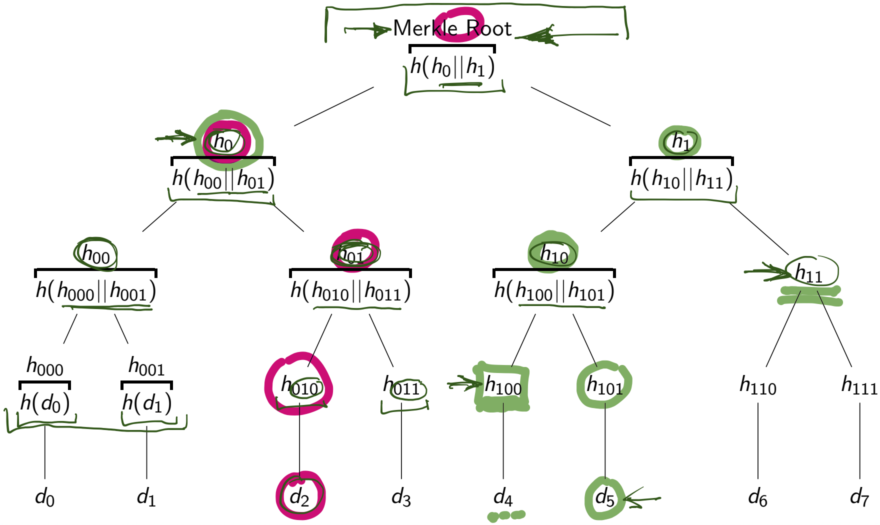
Um zu $N = 2^n$ Datensätzen $d_0, \ldots, d_{N-1}$ einen Merkle Tree zu erstellen, geht man wie folgt vor:

Schritt 1: Blätter des Merkle Tree berechnen: Zunächst wird jeder Datensatz d_i mit einer Hashfunktion h gehasht. Den Hashwert bezeichnen wir mit h_i (wobei wir für i die n-stellige Binärdarstellung verwenden). Diese Hashwerte sind die Blätter des Merkle Tree.

Schritt 2, 3, . . . : Innere Knoten des Merkle Tree berechnen: Von den Blättern hin zur Wurzel des Baums werden nun die Knoten des Merkle Tree berechnet. Der Wert an einem inneren Knoten ergibt sich dabei als der Hashwert über die Konkatenation der Werte der beiden Kinder des Knotens.

Letzter Schritt: Merkle Root berechnen: Der Wert an der Wurzel des Baums – **Merkle Root** genannt – ergibt sich genau so wie die Werte der inneren Knoten.

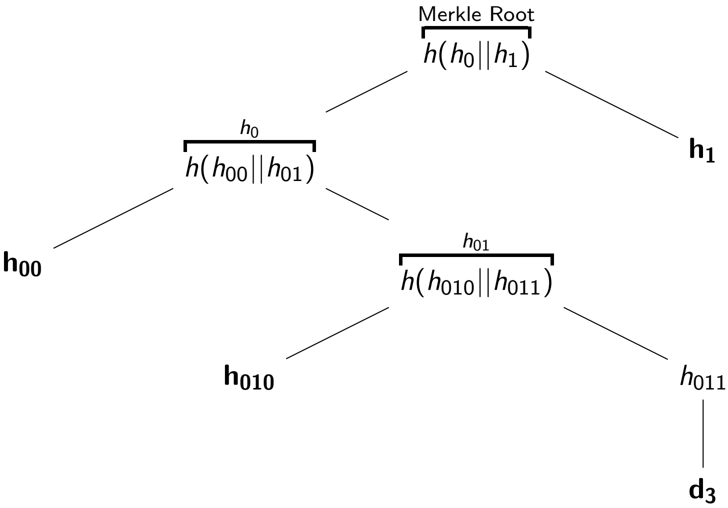
Ein Merkle Tree für 8 Datensätze



Verfizieren eines Datensatzes

Sind alle verarbeiteten Datensätze bekannt, lässt sich auf die selbe Art und Weise die Merkle Root wieder berechnen. Jede Änderung eines oder mehrerer Datensätze ändert den Wert der Merkle Root. Einfach einen Hashwert über alle Hashwerte zu berechnen hätte den selben Effekt und wäre viel einfacher.
Interessant wird die Angelegenheit aber, sobald ein einzelner Datensatz überprüft werden soll. In diesem Fall kann – anstatt alle Datensätze (oder deren Hashwerte) – zu verwenden, auch gerade so viel Information geliefert werden, wie nötig ist, um zur Merkle Root hochzuhashen. Sie bilden den sogenannten **Authentication Path**.
Es ist hier einfach zu erkennen, dass für $N = 2^n$ nicht alle 2^n Blätter des Baums benötigt werden, sondern nur n Hashwerte für den Authentication Path – einer auf jeder Ebene des Baums. Diese n Hashwerte zusammen mit dem Datensatz können also gegen die Merkle Root verglichen werden.

Verifikation des Datensatzes d_3 mit h_{010}, h_{00}, h_1 als Authentication Path



Anwendung: File Sharing (1)

Eine praktische Anwendung von Merkle Trees zur Integritätssicherung ist in Filesharing-Systemen zu finden.
Dort werden (große) Dateien in gleich große Stücke aufgeteilt, die dann als einzelne Teile auch aus verschiedenen Quellen geladen und dann wieder zusammengebaut werden können. Einzelne Teile können durch Fehler bei der Übertragung oder vorsätzlich durch Filesharer verändert werden.
Um derartige Veränderungen von einzelnen Teilen erkennen zu können, können Merkle Trees eingesetzt werden.

Anwendung: File Sharing (2)

Jeder Teil wird zu einem Blatt im Merkle Tree Die Merkle Root dient als Prüfsumme für die gesamte Datei, die zentral an einer vertrauenswürdigen Stelle heruntergeladen werden kann.
Mit einem Teil wird nun auch sein Authentication Path heruntergeladen und dessen Integrität kann damit über die Merkle Root auch sofort überprüft werden.
In Bittorent wurden ursprünglich in einem Torrent Hashwerte für alle Teile (Chunks) gespeichert. In Bittorrent Version 2 wird nur mehr die Merkle Root gespeichert und dafür jeweils der Authentication Path mit einem Teil mitgeliefert.

One-Time Signatures

Lamport-Signaturen – Schlüsselerzeugung

$$\begin{tikzcd} \boxed{s_0} \arrow[d, "h"'] & \boxed{s_1} \arrow[d, "h"'] \\ \boxed{p_0} & \boxed{p_1} \end{tikzcd}$$

- Wähle eine Hashfunktion h mit 256 Bit Outputlänge.
- Der **Private Key** Pr besteht aus zwei zufällig gewählten Bitfolgen
$$s_0, s_1$$
der Länge 256.
- Der dazugehörige **Public Key** besteht aus den Hashwerten
$$p_0 := h(s_0) \text{ und } p_1 := h(s_1).$$

Private Keys und Public Keys sind also je 512 Bit lang.

Lamport-Signaturen – Signieren

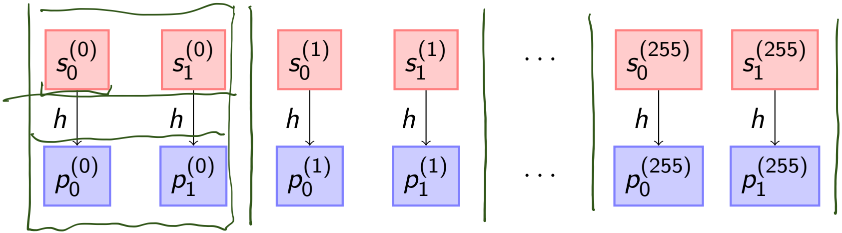
- Mit diesem Verfahren kann nur ein einzelnes Bit b signiert werden. Für $b = 0$ ist die Signatur s_0, für $b = 1$ ist die Signatur s_1, oder kürzer:

$$\mathrm{Sign}_{Pr}(b) = s_b.$$

Lamport-Signaturen – Verifizieren

- Um zu prüfen, ob die Signatur gültig ist, braucht nur ihr Hashwert berechnet zu werden.
- Für das Bit $b = 0$ sollte sich p_0, für das Bit $b = 1$ sollte sich p_1 ergeben.

Lamport-Signaturen – 256 Bits

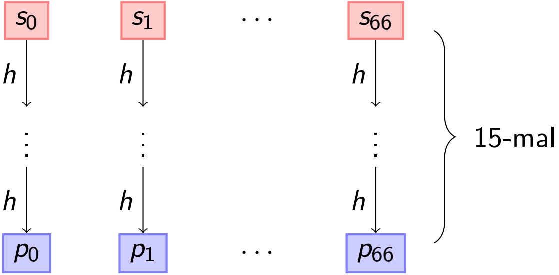


Private Keys und Public Keys werden in diesem Fall $2 \cdot 256 \cdot 256$ Bit, also 16 KiB groß. Die Signatur zu einer 256 Bit langen Nachricht $b_0, \ldots, b_{255}$ ist dann

$$\mathrm{Sign}_{Pr}(b_0, \ldots, b_{255}) = s_{b_0}^{(0)}, s_{b_1}^{(1)}, \ldots, s_{b_{255}}^{(255)}.$$

Signaturen sind $256 \cdot 256$ Bit, also 8 KiB groß.

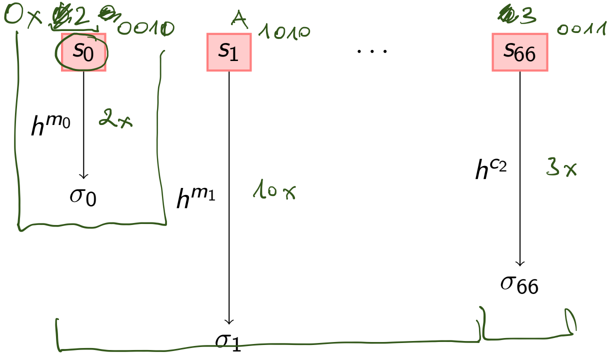
WOTS – Schlüsselerzeugung



Private Keys und Public Keys sind damit nur $67 \cdot 256$ Bit, also etwas über 2 KiB lang.

- Zum Signieren einer 256 Bit langen Nachricht m wird diese in 64 4-Bit-Stücke $m_0, \ldots, m_{63}$ zerlegt. Jeder dieser Werte wird als eine Zahl zwischen 0 und 15 interpretiert.
- Dann wird der Wert $c := 64 \cdot 15 - (\boxed{m_0 + m_1 + \cdots + m_{63}})$ berechnet. Diese Zahl liegt zwischen 0 und $64 \cdot 15$ und kann binär mit 12 Bits dargestellt werden. Diese zwölf Bits werden wieder in drei 4-Bit-Stücke c_0, c_1, c_2 zerlegt, die auch wieder als Zahlen zwischen 0 und 15 interpretiert werden.
- Insgesamt erhält man so 67 Werte zwischen 0 und 15.

WOTS – Signieren (2)



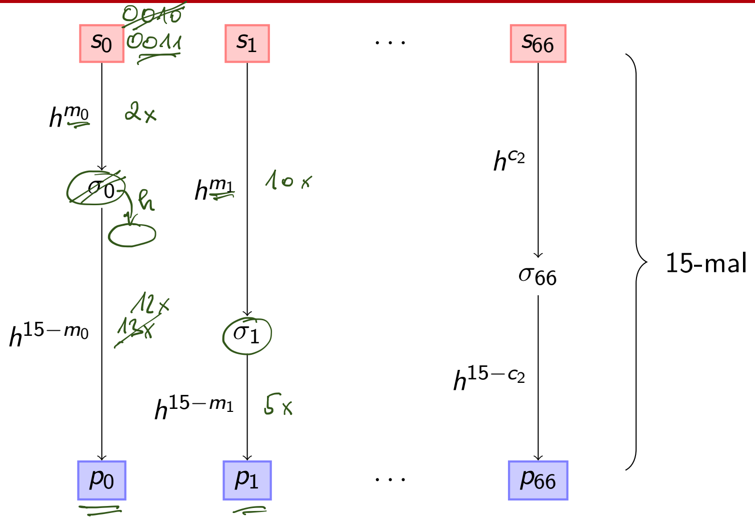
Um auszudrücken, dass ein Wert s k-mal hintereinander gehasht wird, schreiben wir hier

$$h^k(s) := \underbrace{h(h(\ldots h(s)))}_{k\text{-mal}}.$$

Die Signatur ist nun

$$\mathrm{Sign}_{Pr}(m) := (\sigma_0, \sigma_1, \ldots, \sigma_{66}).$$

Signaturen sind somit ebenfalls etwas über 2 KiB lang.



s_0
~~0010~~
0011
s_1
...
s_{66}
h^{m_0}
2x
σ_0
h
h^{m_1}
10x
h^{c_2}
σ_{66}
15-mal
h^{15-m_0}
12x
~~13x~~
σ_1
h^{15-m_1}
5x
h^{15-c_2}
p_0
p_1
...
p_{66}

WOTS – Verifizieren (2)

Zur Verifikation einer Signatur werden $m_0, \ldots, m_{63}$ und c_0, c_1, c_2 wie beim Signieren berechnet. Der erste Wert s_0 wurde beim Signieren m_0-mal gehasht und ergab den Wert σ_0 in der Signatur. Dieses σ_0 wird nun noch weitere $(15 - m_0)$-mal gehasht. Das Ergebnis wird mit p_0 verglichen. Für die übrigen Blöcke wird analog verfahren.

WOTS – Prüfbits

Die zusätzlichen Bits c_0, c_1, c_2 dienen hier als Prüfbits. Sieht man nämlich eine Signatur z. B. für eine Nachricht mit einem Block m_0, musste für diese Signatur s_0 genau m_0-mal gehasht werden.
Es ist dann ganz einfach, eine Signatur für eine Nachricht zu erstellen, wo m_0 um 1 größer ist. Dazu muss man den m_0-mal gehashten Wert aus der Signatur nur noch einmal hashen.
Der Wert c hilft gegen diesen Angriff. Wird nämlich einer der Werte m_i größer, dann auch deren Summe und damit wird c kleiner. Daher wird einer der Werte c_0, c_1, c_2 kleiner. Dort kann der passende Hashwert nicht mehr einfach berechnet werden.