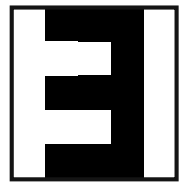
HUMBOLDT-UNIVERSITÄT ZU BERLIN

Implementierung eines MPFSS Algorithmus mit Cuckoo Hashing

Leonie Reichert · 1.11.2019

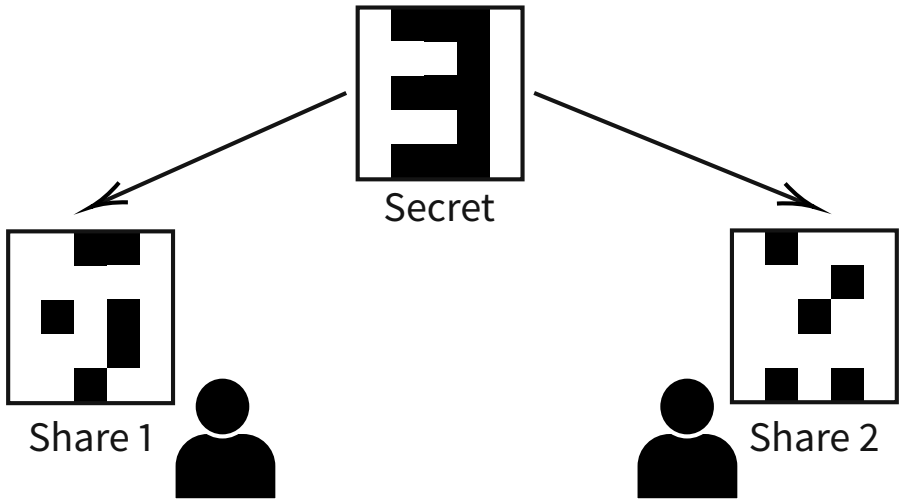
Was ist Multi-Point Function Secret Sharing?

Was ist Secret Sharing?

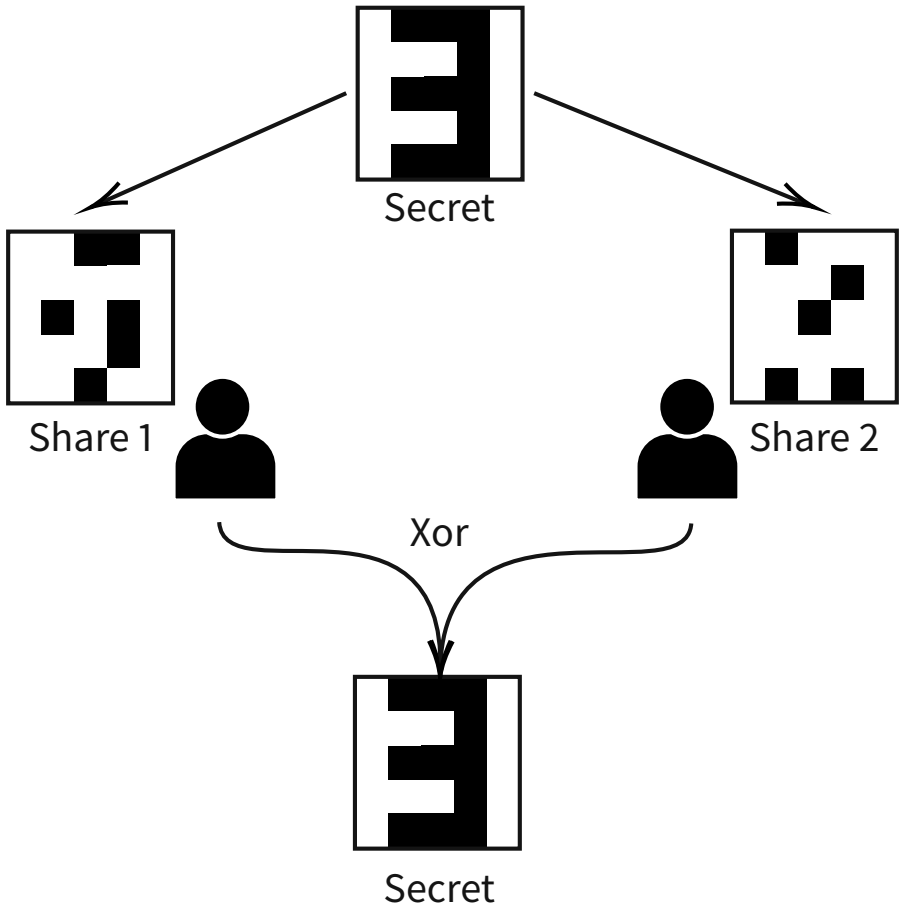


Secret

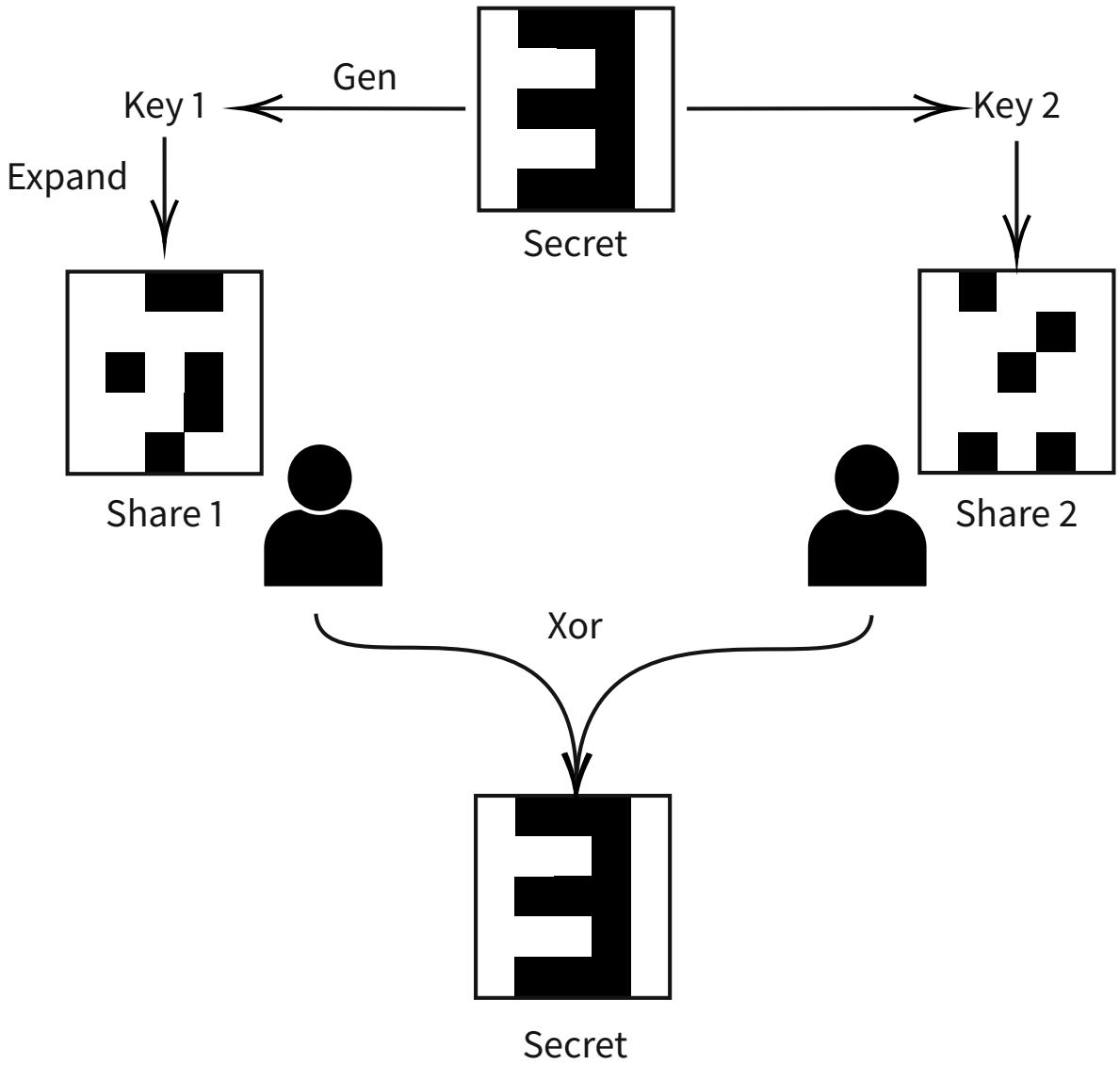
Was ist Secret Sharing?



Was ist Secret Sharing?



Was ist Secret Sharing?

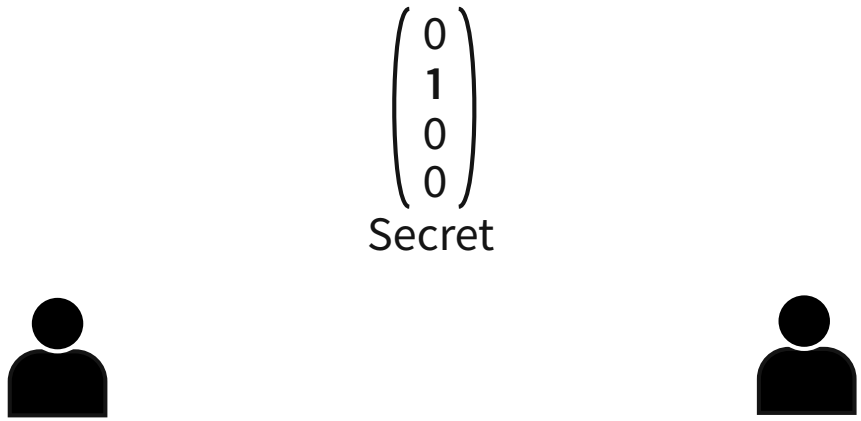


Was ist Function Secret Sharing?

- Grundidee: Function Secret Sharing (FSS)
  - Zerlege $f(x)$ so dass : $f(x) = f_0(x) + f_1(x)$
  - Teilfunktionen verschleiern $f(x)$
  - $f_0(b)$ und $f_1(b)$ für beliebiges b berechenbar
    - $\Rightarrow$ Keine zusätzliche Kommunikation
- Zentral oder verteilt berechenbar
- Einfachste Funktion: Point Function
  - Überall $f(x) = 0$, außer an einer Stelle

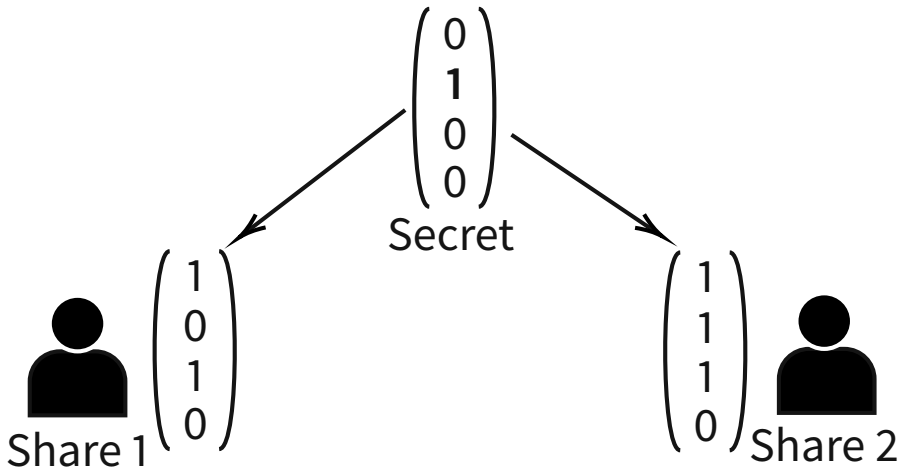
Distributed Point Funtion

- Distributed Point Function (DPF)
  - Implementierung von FSS für Point Functions



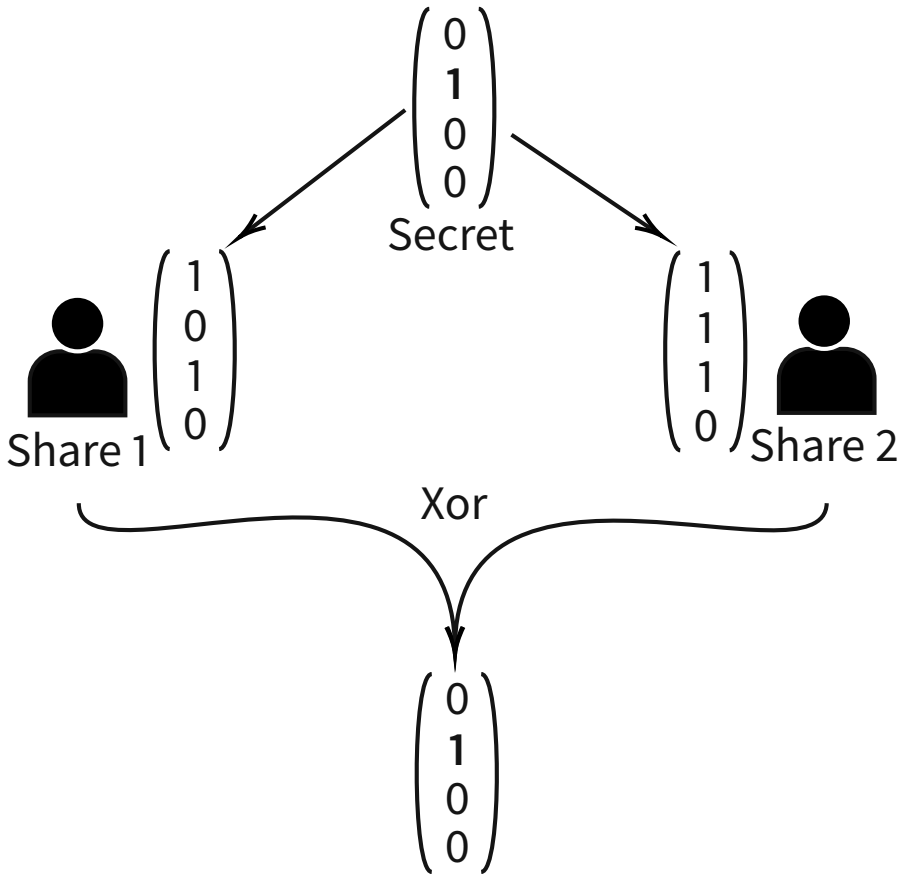
Distributed Point Funtion

- Distributed Point Function (DPF)
  - Implementierung von FSS für Point Functions



Distributed Point Funtion

- Distributed Point Function (DPF)
  - Implementierung von FSS für Point Functions

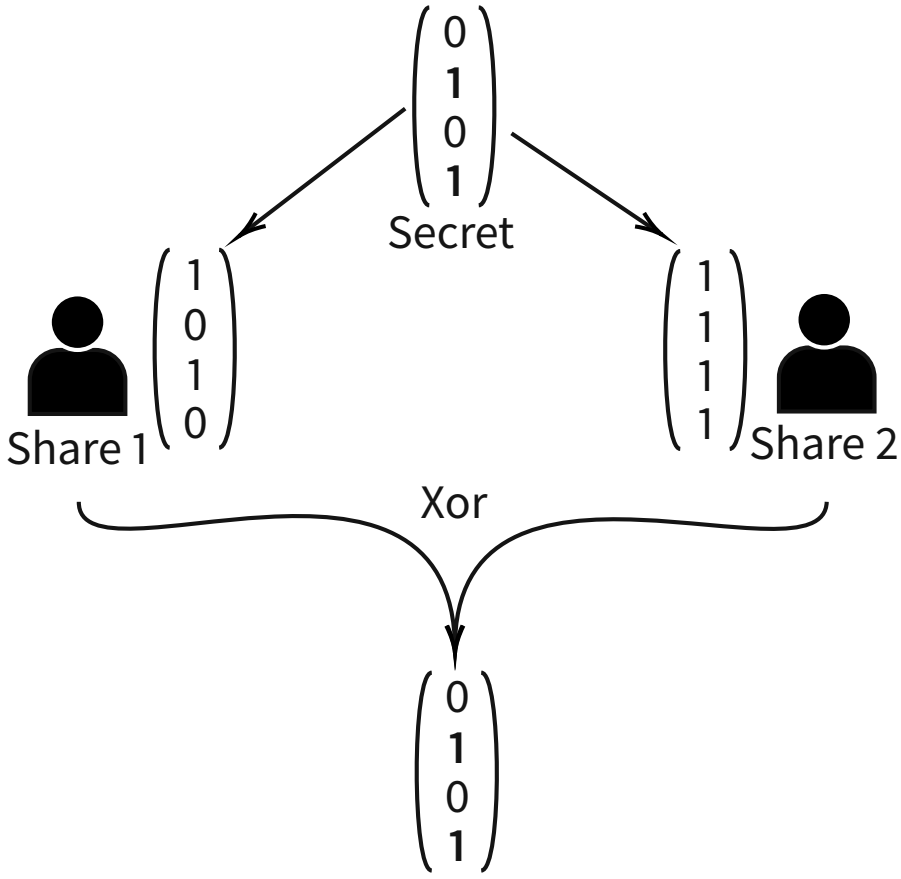


FSS von Multi-Point Functions

- $f(x) \neq 0$ an t Stellen ("Indices")
- Indices verborgen oder einer Partei bekannt

FSS von Multi-Point Functions

- $f(x) \neq 0$ an t Stellen ("Indices")
- Indices verborgen oder einer Partei bekannt



Anwendungen

- Beschleunigung von verschleierten Vektor Multiplikationen
  - VOLE: "Vector Oblivious Linear Function Evaluation" [1]
  - Eine Möglichkeit Matrixmultiplikationen umzusetzten
- Vektor-Matrix Produkt von dünnbesetzten Matritzen
  - MPFSS als Zwischenschritt der Berechnung
- Oblivious Random Access Memory [2]
  - Schneller Schreib- und Leseoperationen
- ...

[1] Boyle et al. : "Compressing Vector OLE"

[2] Doerner and Shelat: "Scaling ORAM for secure computation"

Existierende Implementierungen

- Implementierung DPF existiert [3]
  - In Obliv-C geschrieben

  $\Rightarrow$ Darauf aufbauen

[3] Doerner and Shelat: "Scaling ORAM for secure computation"

[4] Zahur and Evans: "Obliv-C: A Language for Extensible Data-Oblivious Computation"

Existierende Implementierungen

- Implementierung DPF existiert [3]
  - In Obliv-C geschrieben
  - $\Rightarrow$ Darauf aufbauen
- Obliv-C [4]
  - Framework für *Secure Multi-Party Computation*
  - Abstrahiert und Vereinfacht Kommunikation zwischen Parteien
  - Übersetzt C-Code in *Yao Garbled Circuits*

[3] Doerner and Shelat: "Scaling ORAM for secure computation"

[4] Zahur and Evans: "Obliv-C: A Language for Extensible Data-Oblivious Computation"

Naives MPFSS

Naives MPFSS

- Single-Point Function zu Multi-Point Function?
  $\Rightarrow$ Führe DPF t mal aus
- Jede Partei verxodert die entstehenden t Vektoren

Naives MPFSS

- Single-Point Function zu Multi-Point Function?
  $\Rightarrow$ Führe DPF t mal aus
- Jede Partei verxodert die entstehenden t Vektoren
- Problem: Jede DPF geht einmal über gesamtes Inputintervall
  $\Rightarrow$ Kosten: $\mathcal{O}(t \cdot n)$

Naives MPFSS

- Single-Point Function zu Multi-Point Function?
  $\Rightarrow$ Führe DPF t mal aus
- Jede Partei verxodert die entstehenden t Vektoren
- Problem: Jede DPF geht einmal über gesamtes Inputintervall
  $\Rightarrow$ Kosten: $\mathcal{O}(t \cdot n)$
- Vorteil: Verschleierte Indices möglich

MPFSS mit Cukoo Hashing

MPFSS mit Cuckoo Hashing

- Ziel: Laufzeit verbessern
- Idee: Verkleinere Größe der DPFs
  - $\Rightarrow$ Zerlegen des Inputdomäne in Buckets

MPFSS mit Cuckoo Hashing

- Ziel: Laufzeit verbessern
- Idee: Verkleinere Größe der DPFs

  $\Rightarrow$ Zerlegen des Inputdomäne in Buckets
- Zuordnung von gewählten Indices zu Buckets notwendig
  - Ein Index pro Bucket
  - Lösbar durch Hashing!

MPFSS mit Cuckoo Hashing

- Hashing für Zuordnung
- Eine Partei muss Indices kennen
  - Abschwächung!
  - "Known-indices MPFSS"
  - Ausreichend für meisten Anwendungen

Was ist Cuckoo Hashing?

- Mehrere Hashfunktionen $h_1, \ldots, h_w$
  - $\Rightarrow$ Müssen unabhängig voneinander sein
- Eine Tabelle

Was ist Cuckoo Hashing?

- Mehrere Hashfunktionen $h_1, \ldots, h_w$
  - $\Rightarrow$ Müssen unabhängig voneinander sein
- Eine Tabelle
- Bei Kollision
  - Element das zuerst da war wird entfernt und das neue Element eingefügt
  - Das entfernte Element wird mit neuer Hashfunktion wieder eingefügt
  - $\Rightarrow$ Cuckoo, zu dt. Kuckuck

Cuckoo Hashing

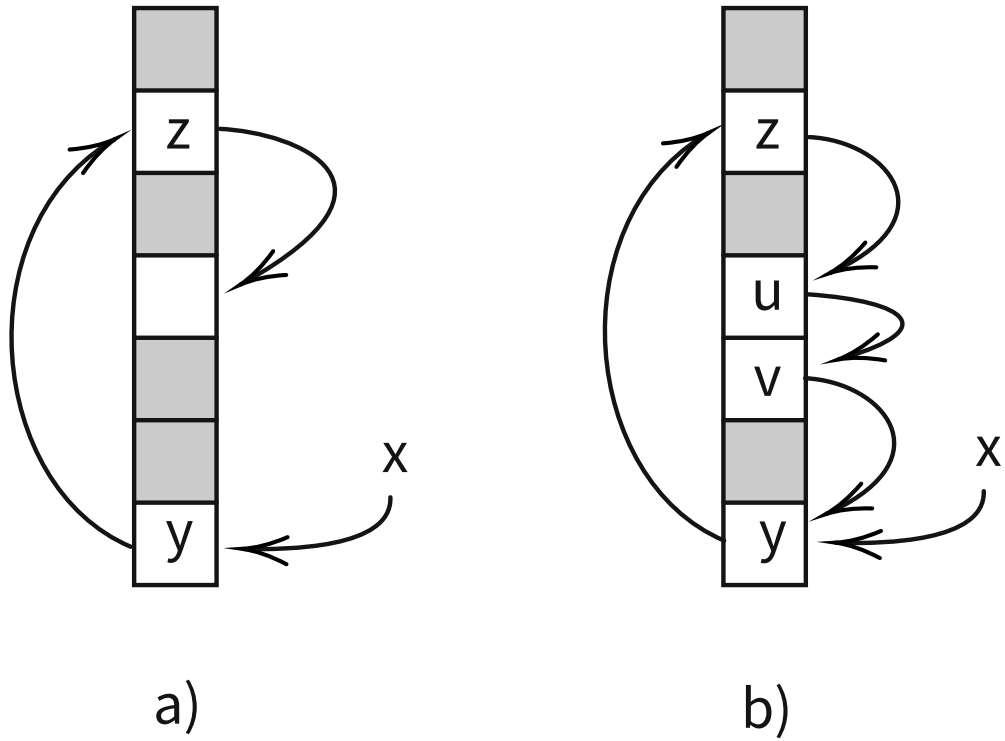


Figure: Graphik aus Pagh and Rodler: "Cuckoo hashing"

Cuckoo Hashing

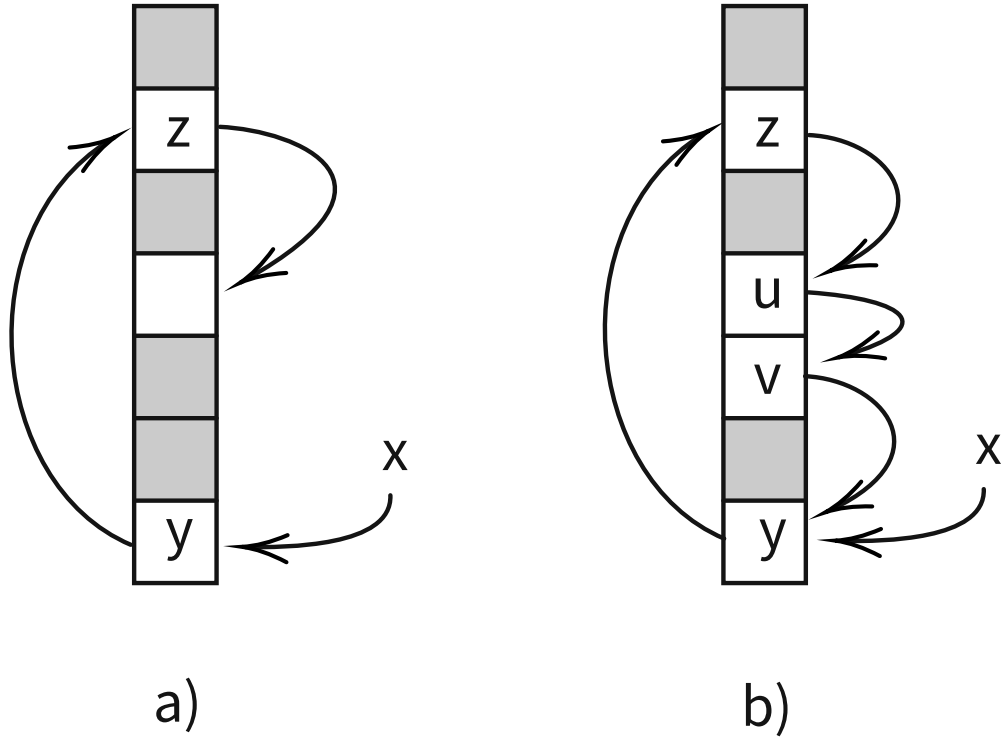


Figure: Graphik aus Pagh and Rodler: "Cuckoo hashing"

- Eine obere Grenze notwendig für die Zahl aufeinander folgender Entfernungen
  - $\Rightarrow$ Erreichen der Grenze wird hier als Fehler behandelt

Vorteile von Cuckoo Hashing

- Konstante Lookup Zeit
- Hohe Auslastung der Tabelle
- Elemente können in sublinearer Zeit eingefügt werden
- Schon häufiger für ähnliche Batching-Probleme verwendet [5]
  - Viele empirische Untersuchungen der Laufzeit
  - Parameterauswahl gut erforscht

[5] Demmler et al. : PIR-PSI: Scaling Private Contact Discovery

MPFSS Cuckoo: Schritte

1. Wähle n, t und w unabhängige Hashfunktionen.

MPFSS Cuckoo: Schritte

1. Wähle n, t und w unabhängige Hashfunktionen.
2. Hashe jedes Element aus Domäne $[0, n]$ je einmal mit jeder Funktion
   - Verwendung von gewöhnlichem Hashverfahren
   - m Buckets verschiedener Länge entstehen

MPFSS Cuckoo: Schritte

1. Wähle n, t und w unabhängige Hashfunktionen.
2. Hashe jedes Element aus Domäne $[0, n]$ je einmal mit jeder Funktion
   - Verwendung von gewöhnlichem Hashverfahren
   - m Buckets verschiedener Länge entstehen
3. Nehme selbe Hashfunktionen für Cuckoo Hashing und hashe die Indices in Tabelle der Größe m
   - Zuordnung von Indices zu Buckets

MPFSS Cuckoo: Schritte

1. Wähle n, t und w unabhängige Hashfunktionen.
2. Hashe jedes Element aus Domäne $[0, n]$ je einmal mit jeder Funktion
   - Verwendung von gewöhnlichem Hashverfahren
   - m Buckets verschiedener Länge entstehen
3. Nehme selbe Hashfunktionen für Cuckoo Hashing und hashe die Indices in Tabelle der Größe m
   - Zuordnung von Indices zu Buckets
4. Amplitudenwerte fixen

MPFSS Cuckoo: Schritte

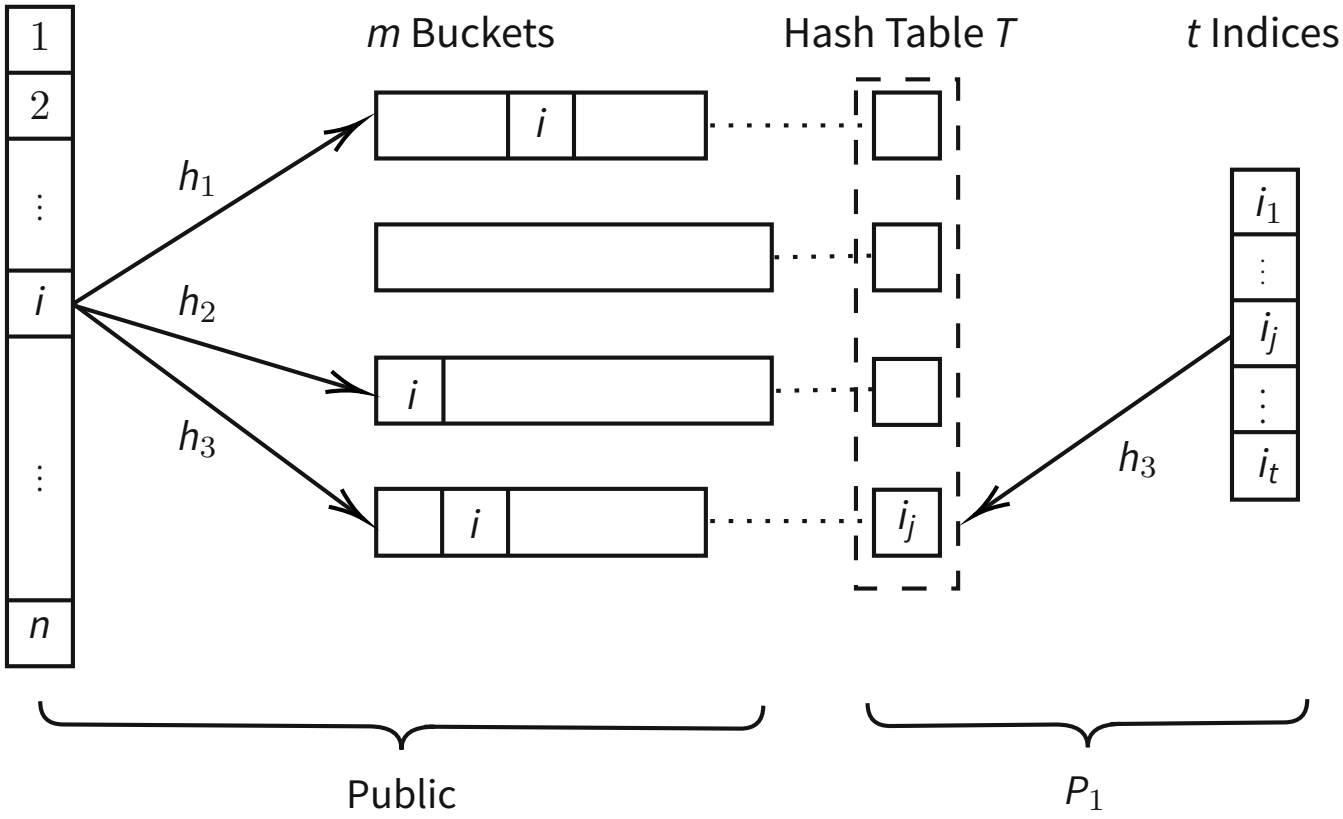
1. Wähle n, t und w unabhängige Hashfunktionen.
2. Hashe jedes Element aus Domäne $[0, n]$ je einmal mit jeder Funktion
   - Verwendung von gewöhnlichem Hashverfahren
   - m Buckets verschiedener Länge entstehen
3. Nehme selbe Hashfunktionen für Cuckoo Hashing und hashe die Indices in Tabelle der Größe m
   - Zuordnung von Indices zu Buckets
4. Amplitudenwerte fixen
5. Führe pro Bucket einmal DPF aus
   - Input für DPF deutlich kürzer

MPFSS Cuckoo: Schritte

1. Wähle n, t und w unabhängige Hashfunktionen.
2. **Hashe jedes Element aus Domäne $[0, n]$ je einmal mit jeder Funktion**

3. **Nehme selbe Hashfunktionen für Cuckoo Hashing und hashe die Indices in Tabelle der Größe m**
4. Amplitudenwerte fixen
5. Führe pro Bucket einmal DPF aus

MPFSS Cuckoo: Buckets und Indices



MPFSS Cuckoo: Schritte

1. Wähle n, t und w unabhängige Hashfunktionen.
2. Hashe jedes Element aus Domäne $[0, n]$ je einmal mit jeder Funktion
3. Nehme selbe Hashfunktionen für Cuckoo Hashing und hashe die Indices in Tabelle der Größe m

4. **Amplitudenwerte fixen**
5. Führe pro Bucket einmal DPF aus

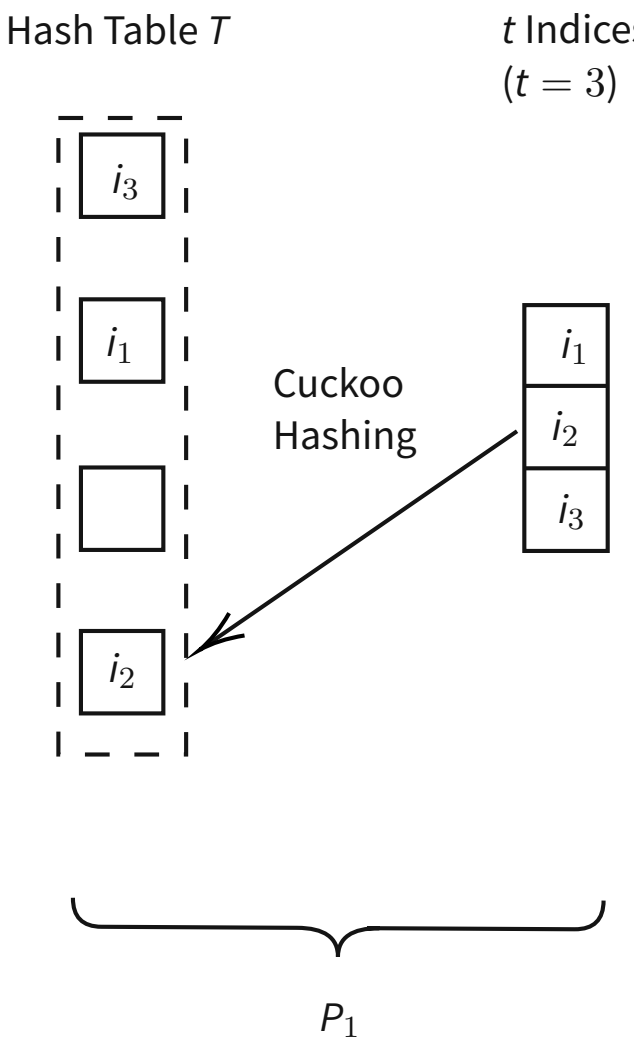
MPFSS Cuckoo: Amplituden Werte fixen

- Amplituden gehören zu Indices
- Amplituden sind verschleiert
  - Jede Partei hat einen Wert für den i-ten Index
  - Wahre Amplitude: XOR der beiden Werte
- Partei P_1 hat Cuckoo Hashing durchgeführt
  - Weiß, in welchem Bucket der i-te Index ist
- Partei P_2 darf diese Info nicht haben

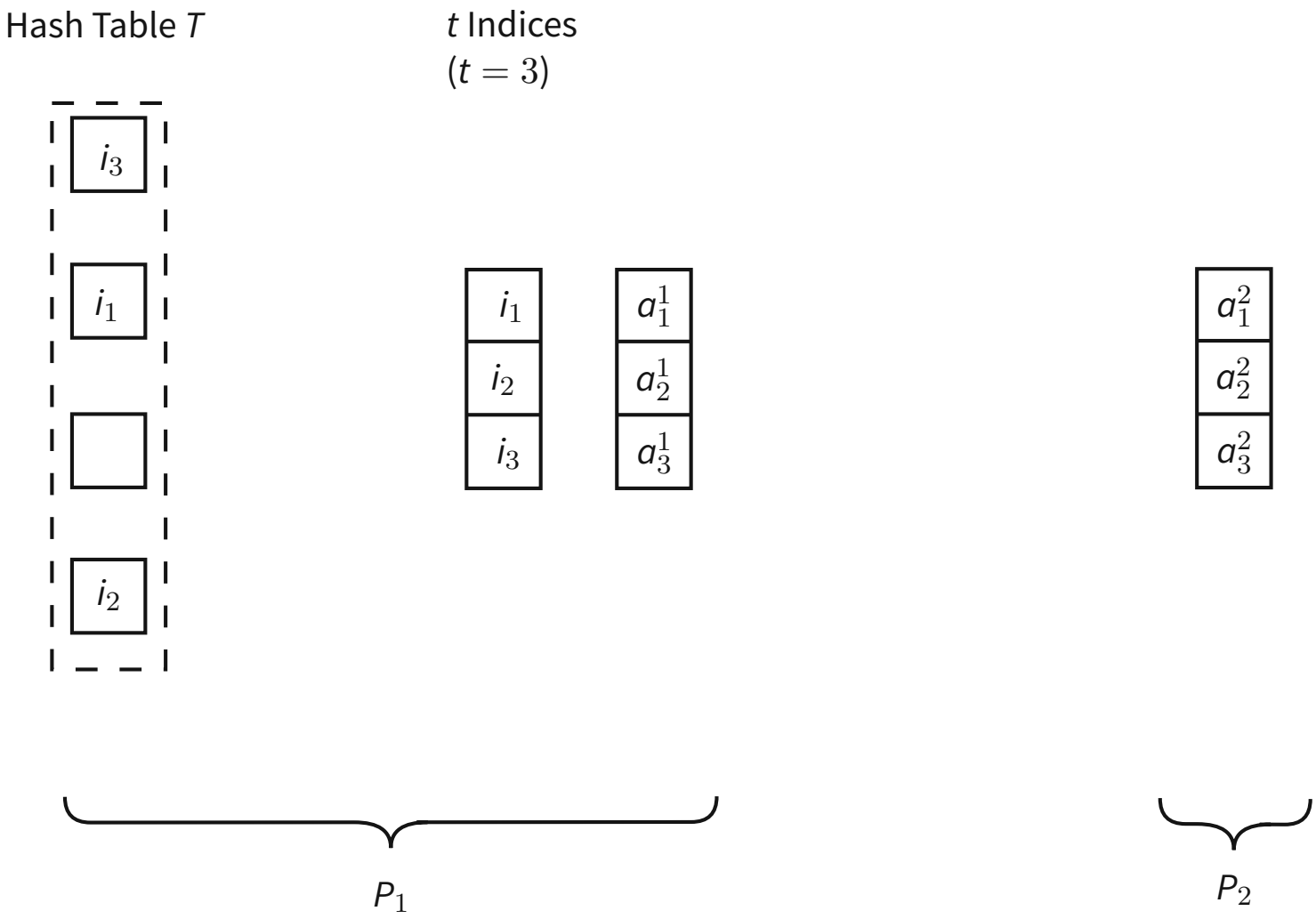
MPFSS Cuckoo: Amplituden Werte fixen

- Lösung: Sorting Networks
  - Effizientes Sortieren von Elementen
  - Implementierungen in Obliv-C dafür existieren
  - Ergebnis: Sortierte Liste von verschleierten Werten
  - D.h. Werte können nur durch Kooperation wiederhergestellt werden

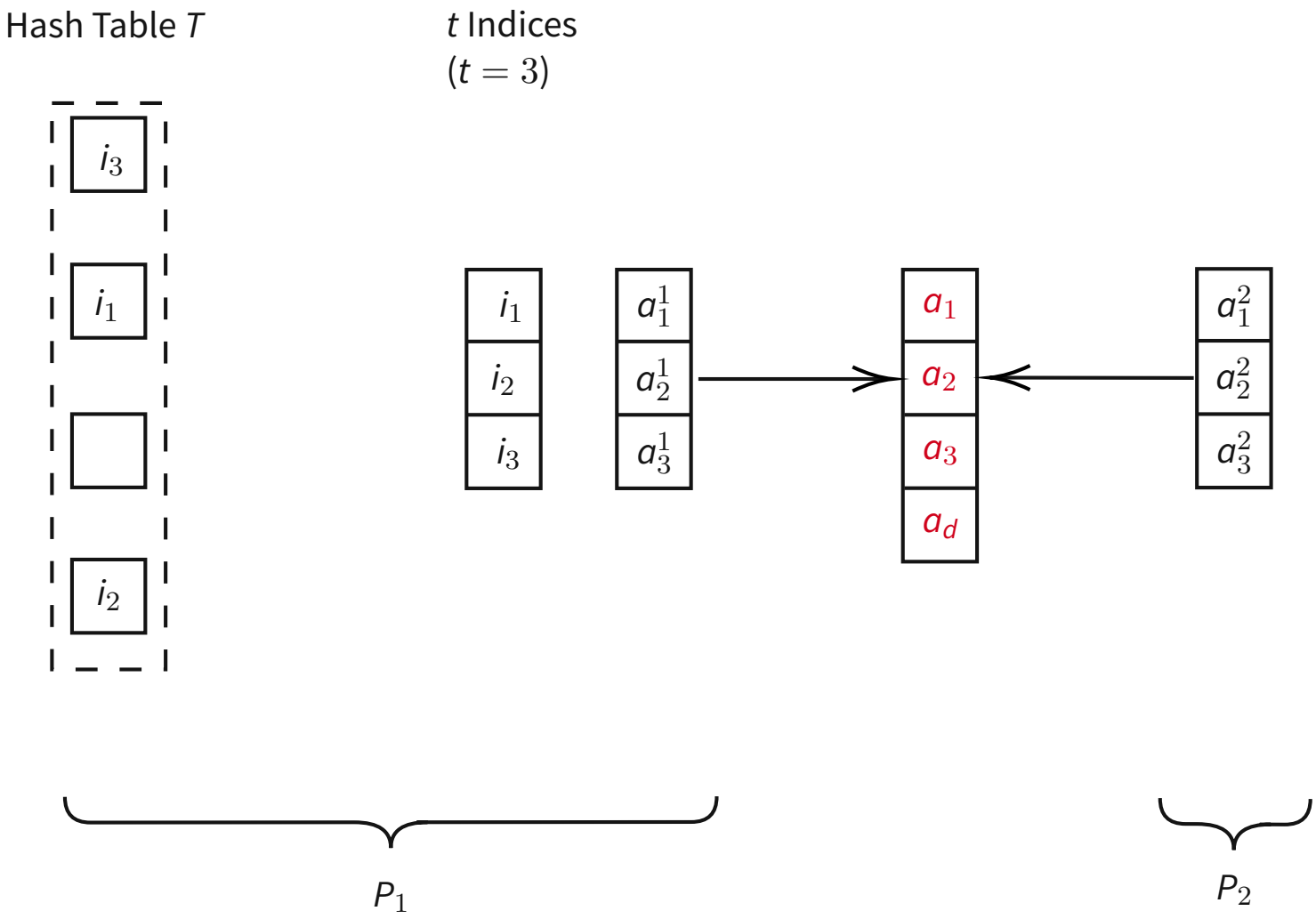
MPFSS Cuckoo: Amplituden Werte fixen



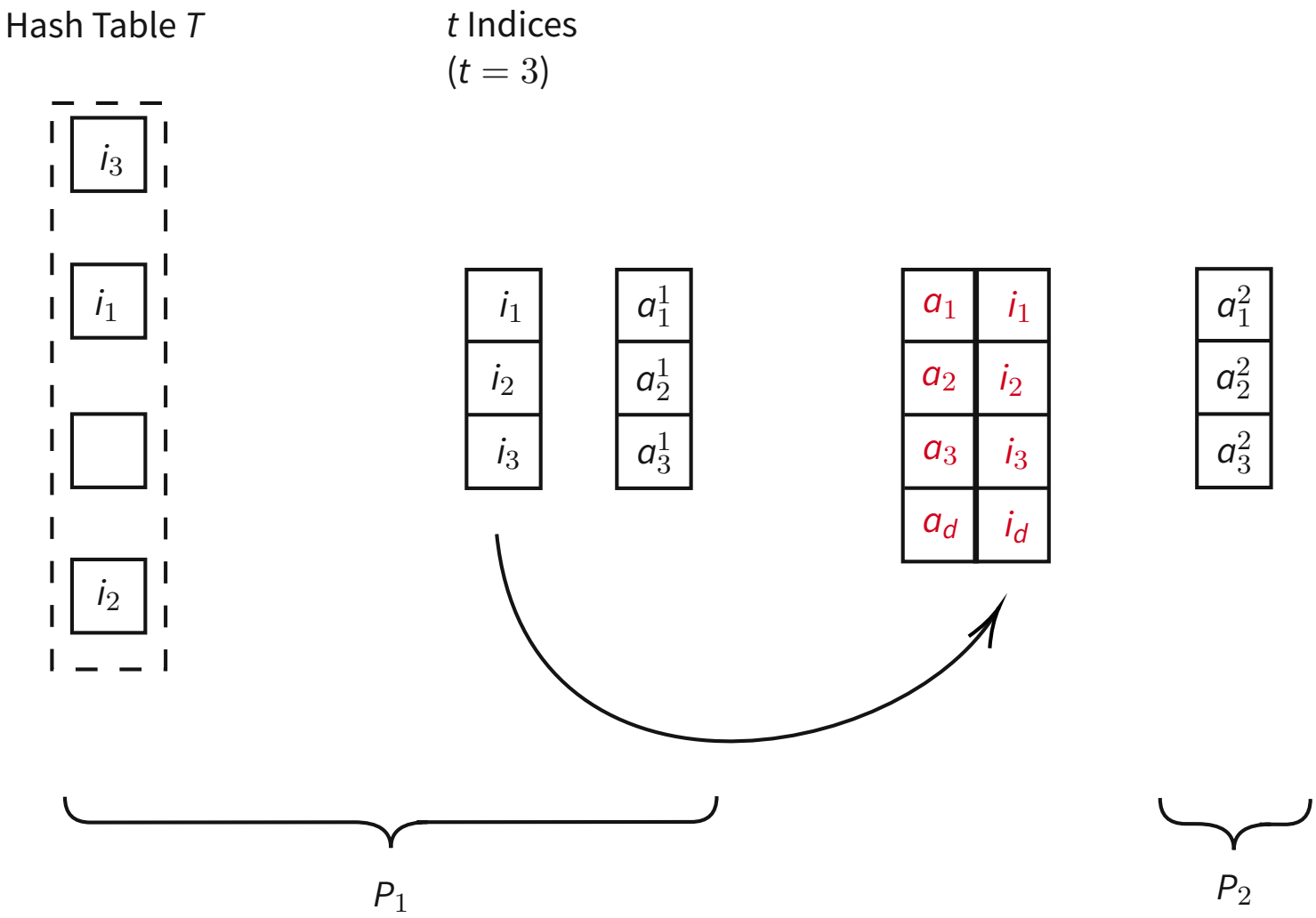
MPFSS Cuckoo: Amplituden Werte fixen



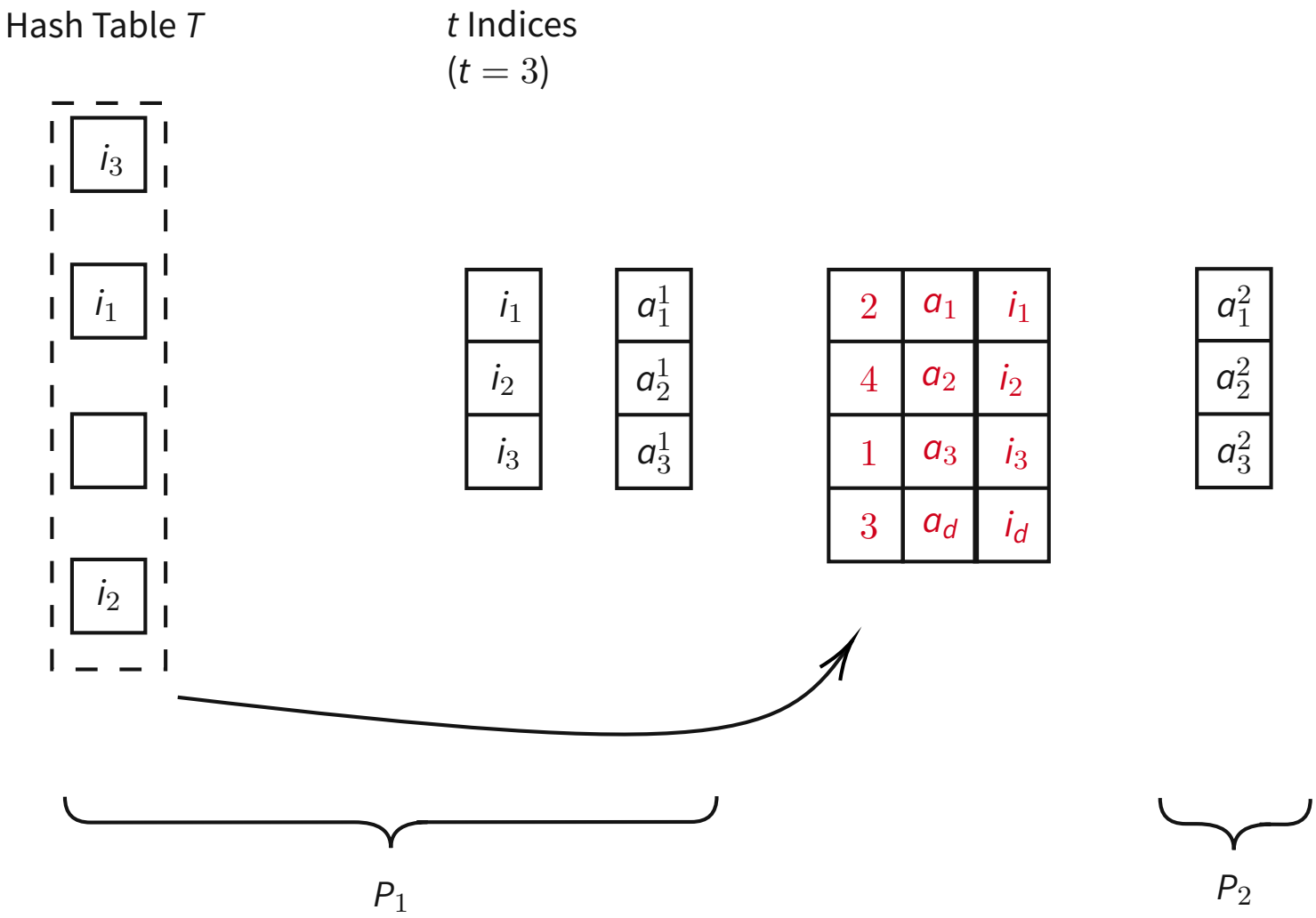
MPFSS Cuckoo: Amplituden Werte fixen



MPFSS Cuckoo: Amplituden Werte fixen



MPFSS Cuckoo: Amplituden Werte fixen



MPFSS Cuckoo: Amplituden Werte fixen

2	a_1	i_1
4	a_2	i_2
1	a_3	i_3
3	a_d	i_d

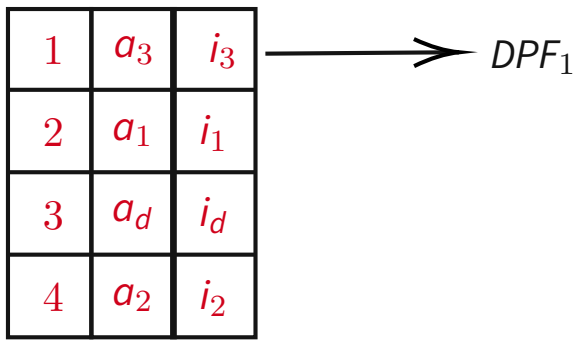
MPFSS Cuckoo: Amplituden Werte fixen

Sorted with Sorting Network

1	a_3	i_3
2	a_1	i_1
3	a_d	i_d
4	a_2	i_2

MPFSS Cuckoo: Amplituden Werte fixen

Sorted with Sorting Network



MPFSS Cuckoo: Schritte

1. Wähle n, t und w unabhängige Hashfunktionen.
2. Hashe jedes Element aus Domäne $[0, n]$ je einmal mit jeder Funktion
3. Nehme selbe Hashfunktionen für Cuckoo Hashing und hashe die Indices in Tabelle der Größe m
4. Amplitudenwerte fixen

5. **Führe pro Bucket einmal DPF aus**

MPFSS Cuckoo: DPF ausführen

- Pro Bucket einmal DPF ausführen
  - Position von Index t_i in Bucket i (verschleiert!)
  - Zugehörigen Amplitudenwert a_i (verschleiert!)
  - Größe von Bucket i

MPFSS Cuckoo: DPF ausführen

- Pro Bucket einmal DPF ausführen
  - Position von Index t_i in Bucket i (verschleiert!)
  - Zugehörigen Amplitudenwert a_i (verschleiert!)
  - Größe von Bucket i
- Erhalte m DPF Output Vektoren
- Kombination der Output Vektoren ergibt MPFSS Vektor mit Länge n

MPFSS Cuckoo: Optimierungen Cuckoo Hashing

- Wahrscheinlichkeit das Hashing fehlschlägt minimieren
  - $\Rightarrow$ Fehlschlagen leaked Informationen

MPFSS Cuckoo: Optimierungen Cuckoo Hashing

- Wahrscheinlichkeit das Hashing fehlschlägt minimieren
  - $\Rightarrow$ Fehlschlagen leaked Informationen
- Auslastung der Tabelle maximieren
  - $\Rightarrow$ Größe der Tabelle legt Anzahl an Buckets fest
  - $\Rightarrow$ Weniger Buckets sind besser

MPFSS Cuckoo: Optimierungen Cuckoo Hashing

- Aus empirische Untersuchungen zu Cuckoo Hashing
  - Drei Hashfunktionen am effizientesten

[6] Angel et al.: PIR with compressed queries and amortized query processing

MPFSS Cuckoo: Optimierungen Cuckoo Hashing

- Aus empirische Untersuchungen zu Cuckoo Hashing
  - Drei Hashfunktionen am effizientesten
  - Zufällige Hashfunktion als nächstes

[6] Angel et al.: PIR with compressed queries and amortized query processing

MPFSS Cuckoo: Optimierungen Cuckoo Hashing

- Aus empirische Untersuchungen zu Cuckoo Hashing
  - Drei Hashfunktionen am effizientesten
  - Zufällige Hashfunktion als nächstes
  - Keine zusätzliche Datenstruktur ("Stash")

[6] Angel et al.: PIR with compressed queries and amortized query processing

MPFSS Cuckoo: Optimierungen Cuckoo Hashing

- Aus empirische Untersuchungen zu Cuckoo Hashing
  - Drei Hashfunktionen am effizientesten
  - Zufällige Hashfunktion als nächstes
  - Keine zusätzliche Datenstruktur ("Stash")
  - Für Anzahl Buckets: Verwende Formel von Angel et al. [6]

[6] Angel et al.: PIR with compressed queries and amortized query processing

MPFSS Cuckoo: Optimierungen Cuckoo Hashing

- Aus empirische Untersuchungen zu Cuckoo Hashing
  - Drei Hashfunktionen am effizientesten
  - Zufällige Hashfunktion als nächstes
  - Keine zusätzliche Datenstruktur ("Stash")
  - Für Anzahl Buckets: Verwende Formel von Angel et al. [6]
  - Dadurch Fehlerwahrscheinlichkeit $p = 2^{-40}$

[6] Angel et al.: PIR with compressed queries and amortized query processing

MPFSS Cuckoo: Optimierungen Cuckoo Hashing

- Original Paper zu Cuckoo Hashing
  - Hashfunktionen aus $(log(1), log(n))$-universellen Hashfamilie

[7] Abseil: www.abseil.io/about

MPFSS Cuckoo: Optimierungen Cuckoo Hashing

- Original Paper zu Cuckoo Hashing
  - Hashfunktionen aus $(log(1), log(n))$-universellen Hashfamilie
- "Truely random" Hashfunktionen funktionieren auch

[7] Abseil: www.abseil.io/about

MPFSS Cuckoo: Optimierungen Cuckoo Hashing

- Original Paper zu Cuckoo Hashing
  - Hashfunktionen aus $(log(1), log(n))$-universellen Hashfamilie
- "Truely random" Hashfunktionen funktionieren auch
- Implementierung verwendet Hashfunktion von Abseil [7]
  - Feste zufälligen Wert um zwischen h_1, h_2 und h_3 zu unterscheiden
  - Hashe also Kombination aus Schlüssel und dem Zufallswert

[7] Abseil: www.abseil.io/about

Vergleich der Algorithmen

Vergleich

- MPFSS Naive
  - DPF ausführen: $\mathcal{O}(n \cdot t)$
  - MPFSS Vektor erstellen

Vergleich

- MPFSS Naive
  - DPF ausführen: $\mathcal{O}(n \cdot t)$
  - MPFSS Vektor erstellen
- MPFSS Cuckoo
  - Buckets erstellen mit gewöhnlichem Hashing

    $\Rightarrow$ Vernachlässigbar

Vergleich

- MPFSS Naive
  - DPF ausführen: $\mathcal{O}(n \cdot t)$
  - MPFSS Vektor erstellen
- MPFSS Cuckoo
  - Buckets erstellen mit gewöhnlichem Hashing
    $\Rightarrow$ Vernachlässigbar
  - Zuordnung finden mittels Cuckoo Hashing

Vergleich

- MPFSS Naive
  - DPF ausführen: $\mathcal{O}(n \cdot t)$
  - MPFSS Vektor erstellen
- MPFSS Cuckoo
  - Buckets erstellen mit gewöhnlichem Hashing

    $\Rightarrow$ Vernachlässigbar
  - Zuordnung finden mittels Cuckoo Hashing
  - Amplitudenwerte sortieren mit Sorting Networks

Vergleich

- MPFSS Naive
  - DPF ausführen: $\mathcal{O}(n \cdot t)$
  - MPFSS Vektor erstellen
- MPFSS Cuckoo
  - Buckets erstellen mit gewöhnlichem Hashing

    $\Rightarrow$ Vernachlässigbar
  - Zuordnung finden mittels Cuckoo Hashing
  - Amplitudenwerte sortieren mit Sorting Networks
  - m DPFs ausführen

Vergleich

- MPFSS Naive
  - DPF ausführen: $\mathcal{O}(n \cdot t)$
  - MPFSS Vektor erstellen
- MPFSS Cuckoo
  - Buckets erstellen mit gewöhnlichem Hashing

    $\Rightarrow$ Vernachlässigbar
  - Zuordnung finden mittels Cuckoo Hashing
  - Amplitudenwerte sortieren mit Sorting Networks
  - m DPFs ausführen
  - MPFSS Vektor erstellen

Vergleich

- MPFSS Naive
  - DPF ausführen: $\mathcal{O}(n \cdot t)$
  - MPFSS Vektor erstellen
- MPFSS Cuckoo
  - Buckets erstellen mit gewöhnlichem Hashing
    - $\Rightarrow$ Vernachlässigbar
  - Zuordnung finden mittels Cuckoo Hashing
  - Amplitudenwerte sortieren mit Sorting Networks
  - m DPFs ausführen
  - MPFSS Vektor erstellen

$\Rightarrow$ Komplexer Term für Laufzeit, daher Experimente notwendig

Experimente

Experimente

- Messungen zwischen Microsoft Azure Servern
- Gezeigte Plots: im LAN gemessen
- Code wurde parallelisiert
- Erstellung von Buckets nicht berücksichtigt
- Fehlerbalken sind Standartabweichungen

Experimente: Verschiedene Anzahl Indices

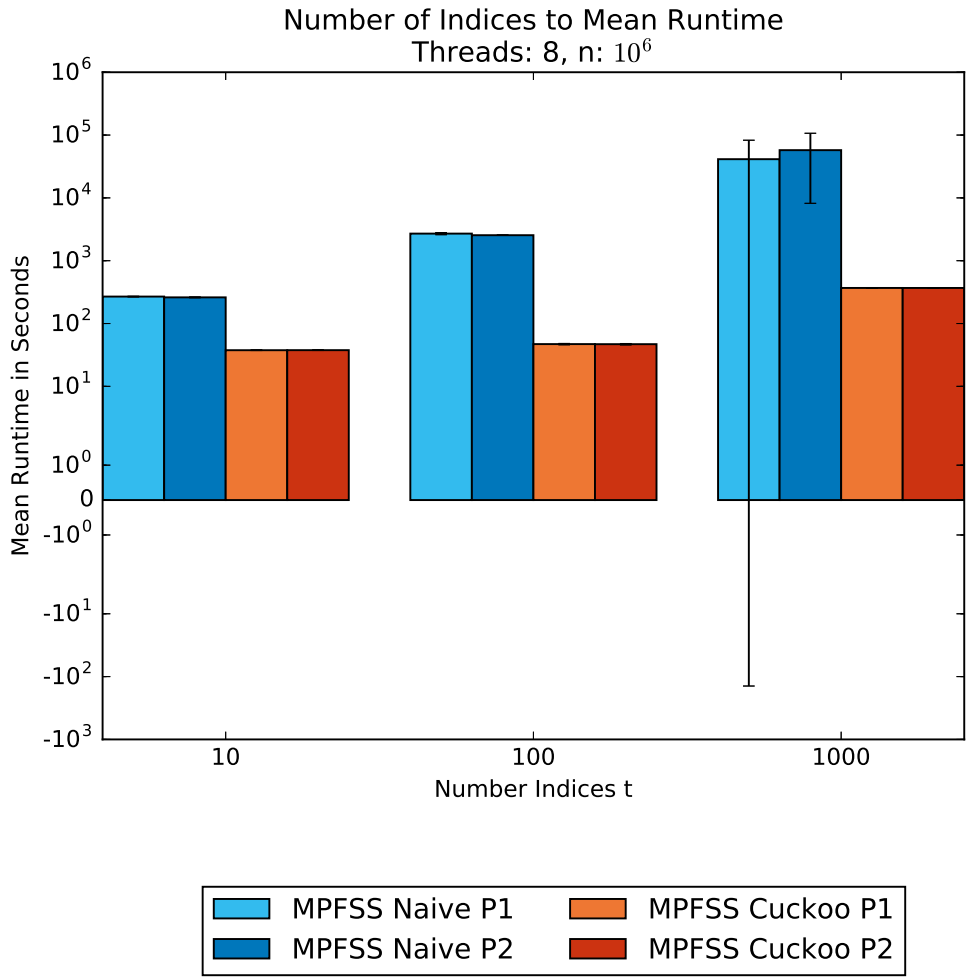


Figure: Jeder Balken 6 - 14 Messungen.

Experimente: Verschiedene Anzahl Indices

- Unterschiede bei P_1 und P_2 für MPFSS Naive
- Geringe Varianz bei MPFSS Cuckoo
- MPFSS Cuckoo deutlich Schneller

Experimente: Variable n und t

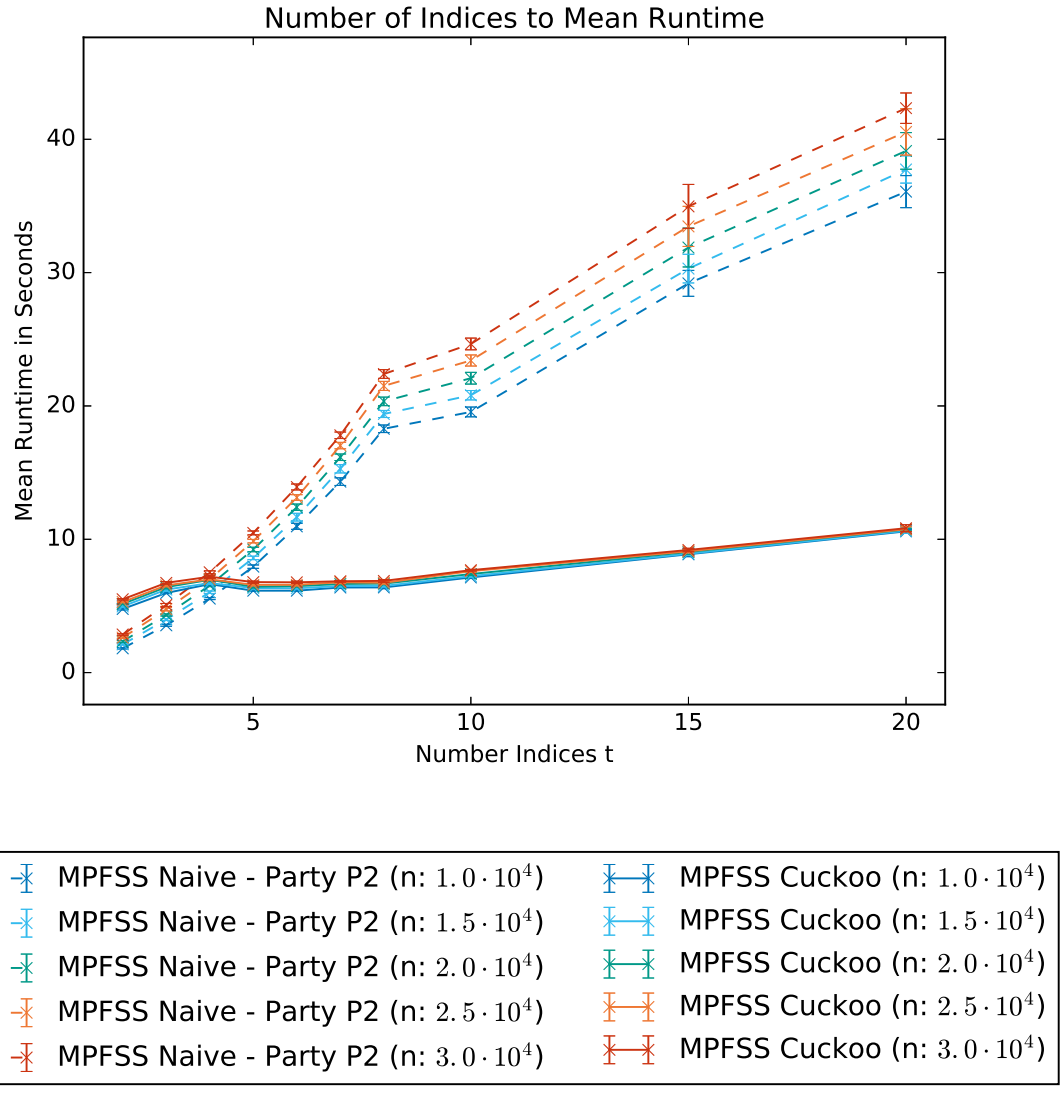


Figure: Jeder Datenpunkt besteht aus 14 Messungen. Gemessen mit 8 Threads.

Experimente: Variable n und t

- Für kleine Anzahl Indices
  - MPFSS Naive kurzzeitig besser
- Für größere n und t
  - MPFSS Cuckoo schneller

Experimente: Multi-Threading

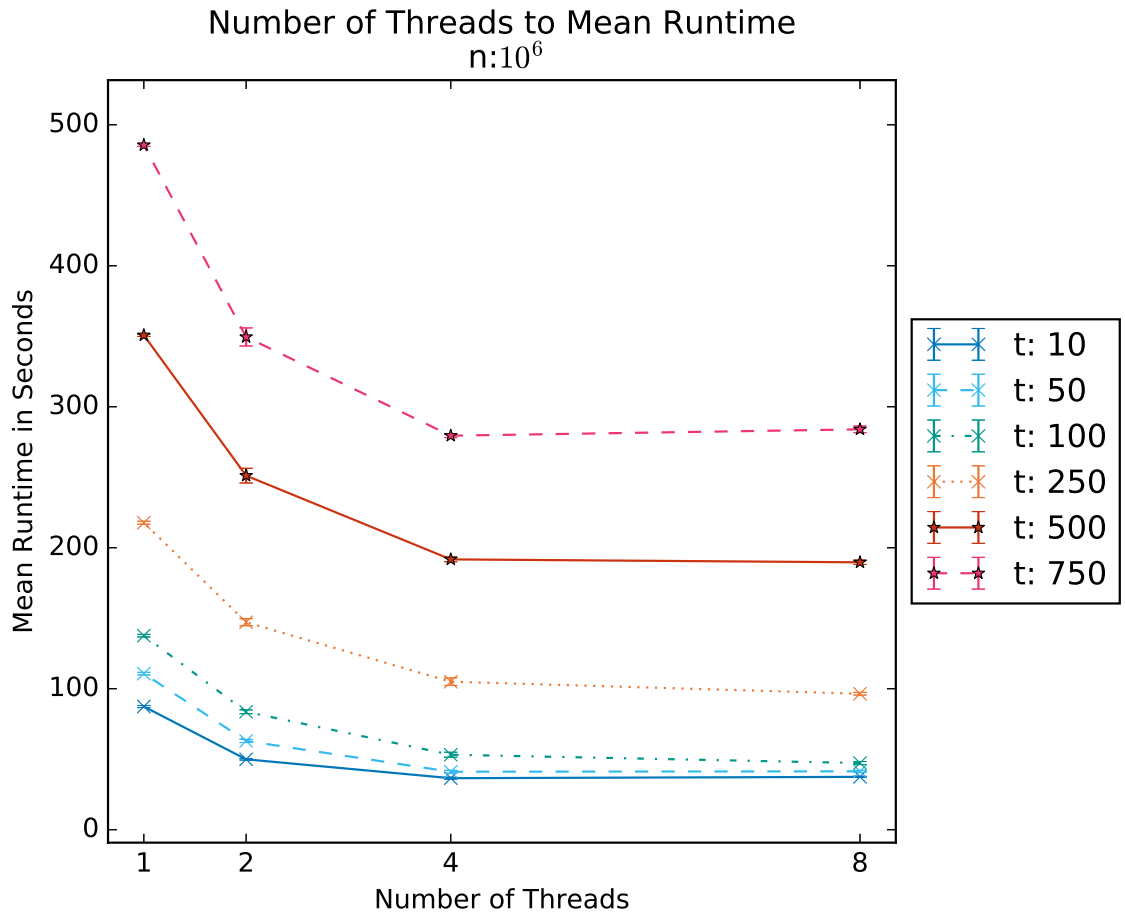


Figure: Jeder Datenpunkt besteht aus 26 - 30 Messungen.

Experimente: Sorting Networks

t	Mittelwert	Std	Mittelwert gesamtes Protokol
30	0,0561	0,005	39,5
300	1,26	0,11	115,0
500	2,49	0,18	189,6
1000	6,18	0,48	368,3

Table: Messungen in Sekunden. Gesamtes Protokoll ausgeführt auf 8 Threads. Sortierung nicht parallelisierbar.

Ergebnis

- MPFSS Cuckoo ist schneller als MPFSS Naive in fast allen Experimenten
- Nur für kleine t und n ist MPFSS Naive schneller
- Messungen übers Internet kommen zum gleichen Ergebnis
- Vier Threads reichen aus

Verbesserungen

- Permutation Networks anstelle von Sorting Networks
- Verwendung von Olivious Memory:
  - Indices können verschleiert bleiben

Vielen Dank für die Aufmerksamkeit.
Fragen?

Quellen

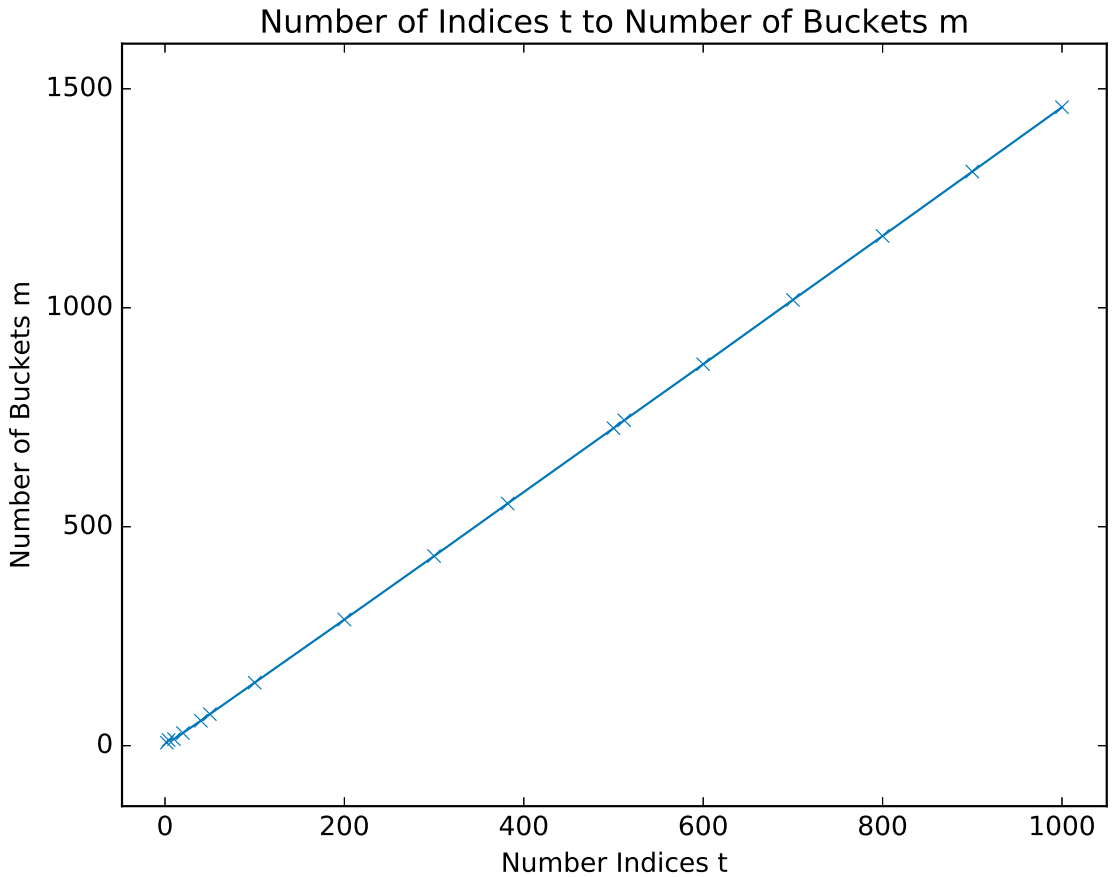
1. Boyle, Elette, et al. "Compressing vector OLE." Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2018.
2. Doerner and Shelat. "Scaling ORAM for secure computation" Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security : 523535. ACM, 2017.
3. Zahur and Evans. "Obliv-C: A Language for Extensible Data-Oblivious Computation" IACR Cryptology ePrint Archive. 2015.
4. Pagh, Rasmus, and Flemming Friche Rodler. "Cuckoo hashing." Journal of Algorithms 51.2 (2004): 122-144
5. Demmler, Daniel, et al. "PIR-PSI: Scaling private contact discovery." Proceedings on Privacy Enhancing Technologies 2018.4 (2018): 159-178
6. Angel, Sebastian, et al. "PIR with compressed queries and amortized query processing." 2018 IEEE Symposium on Security and Privacy (SP). IEEE, 2018.

Backup Slides

MPFSS Cuckoo: Formal für Buckets von Angel et al.

- Empirische Funktion für Größe der Hash Tabelle, abhängig von Anzahl eingefügter Elemente
- Hash table Expansion: $e = \lambda/a_n - b_n/a_n$
- Anzahl Buckets: $m = e \cdot t$
- Wenn $t \leq 512$:
  - $a_n = 123.5, b_n = -130 - log_2(t)$
- Wenn $4 < t < 512$:
  - $a_n = 123.5 \cdot CDF_{normal}(x = t, \mu = 6.3, \sigma = 2.3)$
  - $b_n = -130 \cdot CDF_{normal}(x = t, \mu = 6.45, \sigma = 2.18) - log_2(t)$
  - CDF_{normal}: Kumulative Verteilungsfunktion über eine Normalverteilung
- Wenn $t < 4$: Gleiche Werte wie für $t = 4$

MPFSS Cuckoo: Formal für Buckets von Angel et al.



MPFSS Cuckoo: Theoretische Betrachtung

- Kosten Kommunikation: $\mathcal{O}\big(m\lambda \cdot log(l)\big)$.
- Seedlänge für eine DPF mit Größe N: $\mathcal{O}(\lambda \cdot log(N))$
- DPF wird m mal ausgeführt
- Maximale Bucketlänge: $l = nk/m + \mathcal{O}\big(\sqrt{nk \cdot log(m)/m}\big)$, wenn $n > \mathcal{O}(mlog(m))$
- Sorting Network mit Batcher MergeSort Laufzeit: $\mathcal{O}(mlog(m)^2)$
- Eingesetzt ergibt das ...

$$\mathcal{O}\Big(mlog(m)^2 + m\lambda \cdot log\big(nk/m + \sqrt{nk \cdot log(m)/m}\big)\Big)$$

Experimente: Messung übers Internet

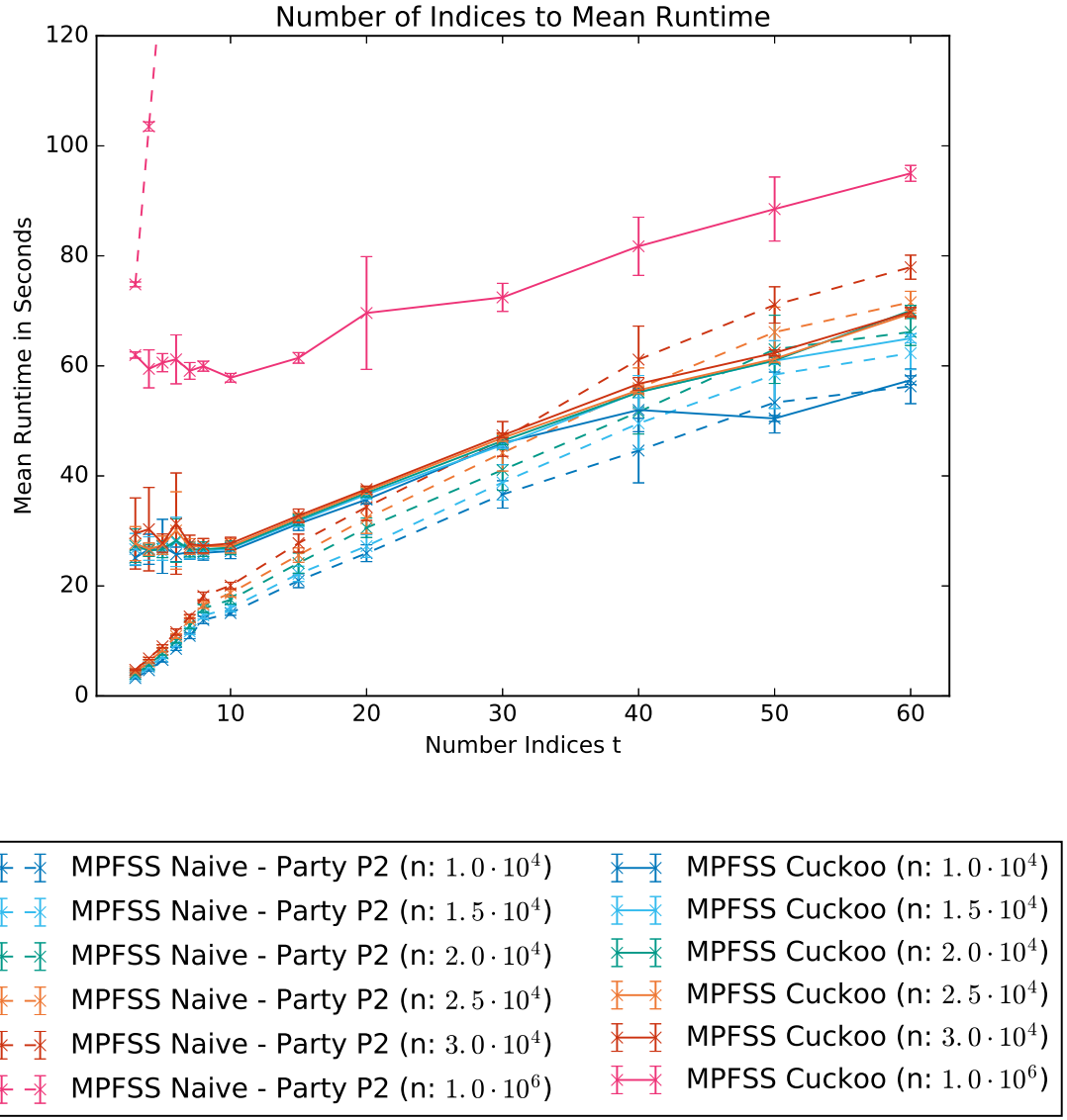


Figure: Jeder Datenpunkt 10 Messungen.

Experimente: Auslastung *n* zu *t*

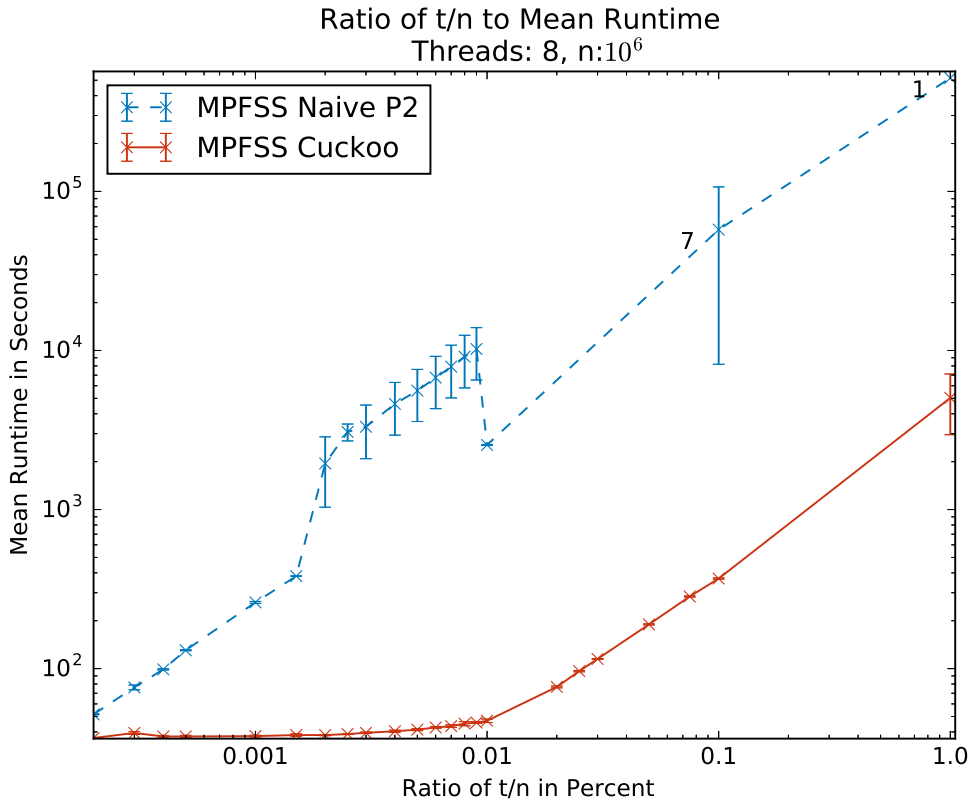


Figure: Jeder Datenpunkt 10 Messungen.

Experimente: Auslastung n zu t

- Untersucht ob wie eine höhere Auslastung der Domäne die Laufzeit beeinflusst
- Relevant für manche Anwendungen, z.B. VOLE