Karlsruhe Institute of Technology

KASTEL

Modellierung und Analyse der kryptografischen Sicherheit von Restic

Master's Thesis of

Fritz Marvin Hund

At the KIT Department of Informatics
KASTEL – Institute of Information Security and Dependability

First examiner:	Prof. Dr. Jörn Müller-Quade
Second examiner:	Prof. Dr. Thorsten Strufe
First advisor:	Dr. Jeremias Mechler
Second advisor:	M.Sc. Saskia Bayreuther

01. Dezember 2025 – 01. Juni 2026

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Zusammenfassung

In dieser Arbeit wird die kryptografische Sicherheit von Restic untersucht. Dazu werden sich die drei Hauptfunktionalitäten von Restic zum Erstellen von Backups, Wiederherstellen von Backups und Verwalten der Backupdaten herausgesucht. Für jede dieser Funktionen werden zwei Modellierungen vorgestellt. Die erste Modellierung legt ihren Schwerpunkt auf Realismus, bei der die Funktionen so nah wie möglich an der Realität modelliert werden. Diese Modellierung kann in Zukunft für weitere Betrachtungen von Restic herangezogen werden, da sie sehr allgemein gehalten ist. Die zweite Modellierung für jede Funktion ist abstrakter. Dabei werden alle drei Funktionen aus der Sicht des in dieser Arbeit betrachteten Angriffsmodells betrachtet. Der Fokus liegt bei dieser Modellierung auf dem Nachrichtenaustausch zwischen den zwei Parteien, in die Restic für diese Modellierung zerlegt wird.
Es wird die informationstheoretische Sicherheit von Restics kryptografischen Primitiven untersucht und akzeptable konkrete Sicherheitsschranken für diese Primitive bestimmt. Daraufhin werden zwei Sicherheitsspiele für Restic vorgestellt. Ein Sicherheitsspiel für die Vertraulichkeit von Restic, das an eine IND$ Sicherheit angelehnt ist. Das zweite Sicherheitsspiel für die Integrität von Restic ähnelt einem EUF-CMA Sicherheitsspiel. Für beide Sicherheitsspiele wird eine allgemein abstrakter Reduktionsbeweis geführt, in den Stück für Stück Restics konkrete kryptografische Primitive eingesetzt werden mit den konkret bestimmten Sicherheitsschranken. Damit wird ein Beweis für Restics Vertraulichkeit und Integrität geführt, solange die Anzahl der Datenblöcke, die mit dem gleichen Masterkey eines Repositorys verschlüsselt werden, niemals $2^{47,7}$ übersteigt.

Contents

1. Einleitung

Restic [8] ist eine Backup-Software zur schnellen und effizienten Datensicherung und Datenwiederherstellung, die hauptsächlich die Programmiersprache GO verwendet. Restic ist eine Software, die eingesetzt wird, um Dateien eines Benutzers zu verschlüsseln, als Backup zu speichern, wiederherzustellen und die verschlüsselten Dateien zu verwalten. Diese Dateien des Benutzers werden Benutzerdaten genannt.
Dabei können mit Restic Backups sowohl lokal gespeichert werden, als auch auf externen Computersystemen. Restic kann auf den drei gängigen Betriebssystemen, Windows, Linux und macOS, betrieben werden und wird als Open-Source-Software auf einem GitHub Depot entwickelt [8]. Restic besitzt mehrere Kommandozeilenbefehle mit denen verschiedene Funktionalitäten von Restic verwendet werden können. Diese Kommandozeilenbefehle werden Restic-Befehle genannt. Die Bedingungen von Restic kann ausschließlich über die Ausführung dieser Restic-Befehle erfolgen. Alle Backups, die mit Restic erstellt werden, werden in einem sogenannten Repository gespeichert. Restic-Befehle können außerdem benutzt werden, um dieses Repository zu verwalten. In dieser Masterarbeit werden drei Restic-Befehle genauer betrachtet. Diese sind der *backup* Befehl zur Erstellung eines neuen Backups, der *restore* Befehl zum Wiederherstellen eines früheren Backups und der *prune* Befehl zur Verwaltung des Repositorys.
Backup-Programme dienen dazu, dem Verlust von Daten entgegenzuwirken und vergangene Systemzustände wiederherzustellen. Daher finden sie oft Verwendung in Systemen mit sensiblen Daten, die für Angreifer von Interesse sein können oder bei Verlust zu Schäden führen. Darum ist es sehr wichtig, dass Backup-Programme keine zusätzliche Sicherheitsschwachstelle im System darstellen. Einem Angreifer sollte es prinzipiell also nicht möglich sein, durch Zugriff auf die verschlüsselten Backupdaten Informationen über die ursprünglichen Daten zu erhalten. Des Weiteren sollte es einem Angreifer auch nicht möglich sein unbemerkt die Backupdaten zu verändern. Restic verschlüsselt alle Backupdaten und authentifizierter alle Backupdaten mit einem Nachrichtenauthenfizierungscode (MAC). Das reicht als Intuition für die Umsetzung der genannten Sicherheitseigenschaften aus, stellt jedoch kein Sicherheitsbeweis dar. Daher ist ein Sicherheitsbeweis für die Vertraulichkeit und Integrität eines Backup-Programms unverzichtbar.
Dies gilt selbstverständlich auch für Restic, welchem ein solcher Sicherheitsbeweis bisher fehlt oder nur zu Teilen vorhanden ist [10]. Wie [4] zeigt, wird Restic unter anderem am CERN Institut eingesetzt, um Backups von Benutzerdaten zu erstellen. Damit findet Restic auch Verwendung an einer internationalen Forschungseinrichtung und Restics Sicherheit ist damit von allgemeinem Interesse.
Es existiert bereits ein Paper [10], das die Sicherheit des Chunking-Verfahrens (siehe Kapitel 2.3.2.6) von Restic überprüft. Allerdings wurde die Backup-Erstellung von Restic als

Ganzes noch nicht offiziell mit einem Sicherheitsmodell verifiziert. Genau dort setzt diese Masterarbeit an und versucht, diese Lücke zu schließen.

2. Grundlagen

Dieses Kapitel enthält Notationen und Definitionen, die in der restlichen Masterarbeit verwendet werden. Dabei werden zunächst verwendete Notationen und allgemeine Begriffe erklärt. Danach wird auf die betrachtete Kryptografie von Restic eingegangen und der Ablauf der kryptografischen Primitive von Restic definiert. Der größte Teil dieses Kapitel geht auf die Grundlagen von Restic und dessen Datenstrukturen ein. Zum Schluss wird das Angreifermodell definiert und diskutiert, dass in dieser Masterarbeit betrachtet wird.

2.1. Notation & Allgemeine Begriffe

Dieses Kapitel definiert die Notationen und Begriffe, die zur Beschreibung Restics als kryptografisches Protokoll und dessen Sicherheitsbeweis, verwendet werden. Dazu werden Notation in allgemeine Notationen für Listen und Datenstrukturen und Notationen speziell für Bytestrings getrennt. Außerdem werden alle von dieser Masterarbeit verwendeten Datentypen und allgemeinen Begriffe definiert.

2.1.1. Allgemeine Notationen

Die Tabelle 2.1 listet Notationen auf, die von dieser Masterarbeit verwendet werden. Notationen, die speziell für Bytestrings verwendet werden, sind in Tabelle 2.2 beschrieben.

Es wird unter anderem bei Listen mit fortlaufendem Index die Notation $\langle X_1, \ldots, X_{max} \rangle$ verwendet. Dabei stellt X_{max} das letzte Element der Liste dar und ist damit das Element mit dem größten Index in dieser Liste. *max* stellt hierbei keine Variable dar, sondern ist lediglich eine Notation für den maximalen Index innerhalb dieser Liste. Für eine zweite Liste $\langle Y_1, \ldots, Y_{max} \rangle$ kann der Index von Y_{max} eine andere Zahl sein als der Index von X_{max}. Diese Notation wird verwendet, wenn viele Listen betrachtet werden, deren Länge zu diesem Zeitpunkt nicht inhaltlich relevant ist.

2.1.2. Bytestring

Wann immer in dieser Masterarbeit ein Bytestring oder eine Datenstruktur in Byte-Darstellung erwähnt wird, ist eine beliebig lange Folge von aufeinanderfolgenden Bytes gemeint. Diese Byte-Folgen werden immer von links nach rechts gelesen. Bei einer Folge von drei Bytes 0xFFAA00 wird 0xFF als das erste Byte bezeichnet und 0x00 als das letzte Byte dieser Folge.

Notation	Bedeutung
$A\|\|B$	Ordnungserhaltende Konkatenation zweier Listen A und B
$A := B$	A wird der deterministische Wert von B zugewiesen
$A \xleftarrow{\$} B$	A wird ein nicht-deterministischer Wert von B zugewiesen
$\langle X_1, \ldots, X_{max} \rangle$	Eine Liste mit fester Ordnung (Vektor). X_{max} ist das letzte Element dieser fortlaufenden Liste mit maximalem Index.
$\|A\|$	Ist A eine Liste oder Menge, ist $\|A\|$ die Anzahl ihrer Elemente. Die Anzahl der Elemente wird auch Größe genannt.
$\tilde{D}$	Eine Datenstruktur von Restic, die durch Restic neu erstellt wurde und von S_{Restic} an S_{Backup} zum Speichern gesendet wird.
D	Eine Datenstruktur von Restic, die nicht neu erstellt wurde, sondern eine Datenstruktur, die auf S_{Backup} gespeichert ist und die von S_{Backup} an S_{Restic} gesendet wird.
nil	Ein Platzhalter für einen nicht genauer definierten Wert. Dieser Platzhalter wird hauptsächlich im Pseudocode verwendet.

Table 2.1.: Von dieser Masterarbeit verwendeten allgemeinen Notationen

Notation	Bedeutung
$A\|\|B$	Eine Konkatenation der beiden Bytestrings A und B
$\|A\|$	Die Länge eines Bytestrings in Bits
$A \oplus B$	Entspricht dem Wert A XOR B und ist damit ein bitweises exklusives Oder
$N[a:]$	Der Bytestring N ohne die ersten a Bytes
$N[:b]$	Der Bytestring N ohne die letzten b Bytes
$N[a:b]$	Der Bytestring N ohne die ersten a und ohne die letzten b Bytes

Table 2.2.: Von dieser Masterarbeit verwendete Notationen für Bytestrings

In Tabelle 2.2 befinden sich alle von dieser Masterarbeit verwendeten Notation von Operationen für Bytestrings.

2.1.3. Datentypen

Um im Pseudocode Variablen besser beschreiben zu können und um die Datenstrukturen von Restic analysieren zu können werden in dieser Masterarbeit Datentypen eingeführt. Die verwendeten Datentypen, sowie deren Definition, befindet sich in Tabelle 2.3.

2.1.4. Backup

Alle erstellten und gespeicherten Daten von einem Aufruf des *backup* Befehls werden als ein Backup bezeichnet. Restic speichert diese Backups in einem Repository (siehe Kapitel

Name	Definition
Int	Eine Ganzzahl ohne fester Länge (Im Code meistens 64 Bit)
$Bool$	Ein Wahrheitswert, der entweder *true* oder *false* ist
$String$	Eine UTF-8 codierte Zeichenkette beliebiger Länge
$Bytestring$	Ein Bytestring beliebiger Länge
$Byte[n]$	Ein Bytestring der Länge n
$Hexstring[n]$	Ein hexadezimal kodierter Bytestring in Form einer Zeichenkette der Länge n. Zwei Zeichen entsprechen 1 Byte des Bytestrings.
$Timestamp$	Ein Zeitstempel mit Datum und Uhrzeit
$List_T$	Eine ordnungserhaltende Liste von Datentypen/Datenstrukturen vom Typ T

Table 2.3.: In dieser Masterarbeit verwendete Datentypen

2.3.1.2). Das heißt ein Repository kann durch mehrere *backup* Befehlsaufrufe mehrere Backups enthalten.

2.1.5. Daten, verschlüsselte Daten und Nachrichten

Als Daten werden in dieser Masterarbeit beliebige Datenstrukturen bezeichnet, die in einen zusammenhängenden Bytestring umgewandelt wurden. Das kann sowohl die Byte-Darstellung eines JSON-Strings sein, als auch Benutzerdaten, von denen ein Backup erstellt werden soll. Als Benutzerdaten wird sämtlicher Inhalt von Dateien bezeichnet, von denen ein Backup erstellt wird.
Mit Backupdaten werden alle verschlüsselten und authentifizierten Datenstrukturen bezeichnet, die beim Backup der Benutzerdaten von Restic erstellt werden.
Als verschlüsselte Daten wird die Zusammensetzung aus Initialisierungsvektor, dem Chiffrat der unverschlüsselten Daten und dem Nachrichtenauthenfizierungscode bezeichnet (siehe Kapitel 2.3.2.3).
Nachrichten werden zwischen verschiedenen Parteien ausgetauscht und können verschlüsselte Daten, unverschlüsselte Daten oder eine Liste davon sein.

2.1.6. Verzeichnissystem

Ein Verzeichnissystem ist eine Menge von Verzeichnissen und Dateien, sowie einer hierarchischen Ordnung, die definiert, welches Verzeichnis oder Datei in welchem anderen Verzeichnis liegt. Außerdem müssen alle Verzeichnisse und Dateien von einem einzigen Verzeichnis (Root-Verzeichnis) aus erreichbar sind. Ein Verzeichnissystem lässt sich mathematisch als gerichteter Graph modellieren. Durch die hierarchische Ordnung sind keine Zyklen erlaubt und durch das Root-Verzeichnis ist jedes andere Verzeichnis oder Datei von diesem Root-Verzeichnis aus erreichbar. Damit ist der Graph eines Verzeichinssystems ein

Baum und wird in Kapitel 2.3.1.9 definiert.
Restics *backup* Befehl und *restore* wird immer auf einem Verzeichnissystem ausgeführt, das sich auf dem System befindet, auf dem Restic ausgeführt wird. Dabei wird durch Target-Pfade (siehe Kapitel 2.3.1.7) und Filter bestimmt von welchen Verzeichnissen und Dateien ein Backup erstellt wird und welche nicht verarbeitet werden und damit ignoriert werden. Diese Masterarbeit unterscheidet zwischen zwei Verzeichnissystemen.
Das reale Verzeichnissystem ist das komplette Verzeichnissystem des Systems auf dem Restic ausgeführt wird (S_{Restic}). Dieses Verzeichnissystem beinhaltet insbesondere Verzeichnisse und Dateien, die nicht von jedem Restic-Befehl verarbeitet werden. Das Root-Verzeichnis des realen Verzeichnissystems ist das Root-Verzeichnis des Betriebssystems von S_{Restic}. Bei einem UNIX-System wäre der Pfad des Root-Verzeichnisses *"/"*.
Das virtuelle Verzeichnissystem ist ein Unterverzeichnissystem des realen Verzeichnissystems. Das heißt, es besitzt die gleiche hierarchische Ordnung, aber nicht alle Verzeichnisse und Dateien des realen Verzeichnissystems. Mathematisch gesehen ist das virtuelle Verzeichnissystem ein Subgraph des realen Verzeichnissystems. Jedem durchgeführten Backup und damit auch jedem *backup* Befehl, der ausgeführt wird, kann eindeutig ein virtuelles Verzeichnissystem zugeordnet werden. Das virtuelle Verzeichnissystem enthält nur Verzeichnisse und Dateien, die durch die Target-Pfade und Filter für das Backup ausgewählt werden. Der *backup* Befehl wählt für jedes ausgewählte Verzeichnis oder Datei auch die Verzeichnisse aus, die in dem Pfad von einem der Target-Pfade bis zu dem ausgewählten Verzeichnis oder Datei enthalten sind. In Restic werden die Target-Pfade bis zu dem Root-Verzeichnis des Systems zurückverfolgt, auf dem Restic ausgeführt wird. Alle durchquerten Verzeichnisse auf dem Pfad von einem Target-Pfad zum Root-Verzeichnis, werden ebenfalls durch Restic zu einem Backup hinzugefügt. Das konstruierte virtuelle Verzeichnissystem beinhaltet ebenfalls jedes Verzeichnis, das in dem Pfad vom Root-Verzeichnis bis zu den jeweiligen Target-Pfaden enthalten ist. Damit enthält ein virtuelles Verzeichnissystem immer das Root-Verzeichnis, alle für das Backup ausgewählten Verzeichnisse und Dateien und alle Verzeichnisse, um die Target-Pfade vom Root-Verzeichnis aus zu erreichen. Ein virtuelles Verzeichnissystem enthält keine Verzeichnisse oder Dateien, die nicht Teil des realen Verzeichnissystems zum Zeitpunkt des Backups waren. Durch diese Konstruktion bildet ein virtuelles Verzeichnissystem ein zusammenhängendes Verzeichnissystem, beginnend beim Root-Verzeichnis des realen Verzeichnissystems. Dadurch kann wie später in Kapitel 2.3.1.9 gezeigt, ein Verzeichnisbaum für ein virtuelles Verzeichnissystem konstruiert werden, der ebenfalls eindeutig für das ausgeführte Backup ist.
Mit dieser Konstruktion kann jeder Ausführung eines *backup* Befehls und damit jedem erstellten Backup ein eindeutiges virtuelles Verzeichnissystem zugeordnet werden. Für die restliche Masterarbeit wird das virtuelle Verzeichnissystem eines Backups nur noch als Verzeichnissystem (VS) für dieses Backup bezeichnet.
Ein virtuelles Verzeichnissystem findet auch für die abstrakte Modellierung des *restore* Befehls Anwendung. Für den *restore* Befehl gibt es keine Filter-Funktionen und nur einen Target-Pfad. Daher entspricht das virtuelle Verzeichnissystem eines *restore* Befehls dem Verzeichnissystem, das der Target-Pfad beschreibt, sowie dem Root-Verzeichnis des Betriebssystems und allen Verzeichnissen, auf dem Pfad vom Root-Verzeichnis bis zu dem Verzeichnis des Target-Pfads. Damit ist auch das Root-Verzeichnis des virtuellen Verzeichnissystems für den *restore* Befehl das Root-Verzeichnis des Betriebssystems.

2.2. Kryptografie

In diesem Kapitel werden die Begriffe Vertraulichkeit und Integrität für diese Masterarbeit definiert. Dabei werden kryptografischen Primitive erklärt, die von Restic eingesetzt werden, um diese Sicherheitseigenschaften zu gewährleisten. Außerdem werden Sicherheitsanforderungen vorgestellt, die in dieser Masterarbeit Verwendung finden.

2.2.1. Vertraulichkeit

Der NIST Standard 1800-26A beschreibt Vertraulichkeit als:

"Confidentiality - preserving authorized restrictions on information access and disclosure, including means for protecting personal privacy and proprietary information"

Im Falle dieser Masterarbeit ist eine autorisierte Person eine Person, die die Schlüssel der eingesetzten Verschlüsselungsverfahren besitzt. Eine nicht autorisierte Person, die das Chiffrat eines Klartextes sieht, sollte nicht in der Lage sein irgendeine Information über den Klartext des Chiffrats berechnen zu können. In der Praxis ist dies nicht immer umsetzbar, da Informationen über die Länge eines Chiffrats oft Rückschlüsse auf die Länge des Klartextes zulassen. Daher werden im Laufe der Masterarbeit diverse Informationen herausgearbeitet, die nicht geheim gehalten werden können und von der Vertraulichkeit ausgeschlossen werden müssen. Diese Informationen werden als Leakage bezeichnet.
Ein kryptografisches Verfahren, um Vertraulichkeit zu gewährleisten ist ein Verschlüsselungsalgorithmus. In diesem Kapitel wird der AES256-CTR Verschlüsselungsalgorithmus vorgestellt, der zum aktuellen Zeitpunkt auch von Restic mit einer Blockgröße von $n = 16$ Byte verwendet wird.
Eine Standard-Sicherheitsanforderung ist Indistinguishability under Chosen Plaintext Attacks (IND-CPA). Das Sicherheitsspiel für IND-CPA wird ebenfalls in diesem Kapitel vorgestellt. Außerdem wird die Sicherheitsanforderung IND$-CPA vorgestellt, die im Laufe dieser Masterarbeit verwendet wird, um die Vertraulichkeit bezüglich IND$-CPA von Restic zu beweisen.

2.2.1.1. AES256-CTR

AES256-CTR bezeichnet einen symmetrischen Verschlüsselungsalgorithmus, der einen Bytestring beliebiger Länge verschlüsseln kann und die Blockchiffre AES256 verwendet. Für die Blockchiffre AES256 wird ein 256 Bit langer Schlüssel benötigt. CTR steht für den Counter-Mode und bezeichnet den Betriebsmodus, mit dem die Blockchiffre verwendet wird.
Für AES256-CTR wird der zu verschlüsselnde Bytestring in m Datenblöcke D_i einer festen Länge n unterteilt. Es wird eine Zahl der Länge n gewählt. Diese Zahl wird Initialisierungsvektor (IV) genannt. AES256-CTR iteriert deterministisch über die Datenblöcke

des Bytestrings in aufsteigender Reihenfolge. Dabei wird für jeden Datenblock der IV um 1 erhöht und mit AES256 und dem Schlüssel k zu einem Bytestring $AES256_k(IV+i)$ der Länge n verschlüsselt. Das XOR-Ergebnis dieses Bytestrings $AES256_k(IV+i)$ und dem aktuellen Datenblock D_i, ergibt den nächsten Block des Chiffrats. Dieser Block hat ebenfalls die Länge n. In Gleichung 2.1 ist der Aufbau des erzeugten Chiffrats mit AES256-CTR beschrieben.

$$AES256_k(IV+0) \oplus D_0 || AES256_k(IV+1) \oplus D_1 || ... || AES256_k(IV+m-1) \oplus D_{m-1} \quad (2.1)$$

Die Entschlüsselung erfolgt analog zur Verschlüsselung, nur dass statt D_i der i-te Block des Chiffrats benutzt wird. Zur Entschlüsselung wird der gleiche IV wie zur Verschlüsselung benutzt und damit kürzt sich durch das XOR der Term $AES256_k(IV+i)$ raus und man erhält den Klartext-Datenblock $AES256_k(IV+i) \oplus AES256_k(IV+i) \oplus D_i = D_i$.

2.2.1.2. IND-CPA

Für viele Verschlüsselungsverfahren ist es der Standard eine IND-CPA Sicherheitsgarantie zu fordern. Um die IND-CPA Sicherheit zu beweisen wird ein Sicherheitsspiel mit zwei Parteien verwendet. Eine Partei ist der Angreifer $\mathcal{A}$ und die andere ist der Challenger, der dem Angreifer ein Verschlüsselungsorakel zur Verfügung stellt. Zum Start des Sicherheitsspiels generiert der Challenger einen Schlüssel für das zu analysierende Verschlüsselungsverfahren. Das Sicherheitsspiel wird in zwei Phasen unterteilt. Die erste Phase ist die Orakel-Phase und die zweite Phase heißt Challenge-Phase.
In der Orakel-Phase kann der Angreifer $\mathcal{A}$ sich beliebige Bytestrings erzeugen und vom Orakel mit dem Schlüssel des Challengers verschlüsseln lassen. Der Angreifer erhält abgesehen von seinen gewählten Klartexten und den dazu erzeugten Chiffraten keine weiteren Informationen. Dieses Vorgehen kann der Angreifer beliebig oft wiederholen und geht dann in die Challenge-Phase über.
In der Challenge-Phase wählt der Angreifer zwei gleich lange Bytestrings und übergibt diese dem Orakel. Der Angreifer darf keinen Bytestring wählen, den er sich in der Orakel-Phase bereits verschlüsseln ließ. Das Orakel wählt zufällig einen der beiden Bytestrings aus, verschlüsselt diesen und übergibt das Chiffrat an $\mathcal{A}$. Der Angreifer gibt eine Vermutung ab, zu welchem der beiden Klartexte das Chiffrat gehört. Ein Angreifer gewinnt das Sicherheitsspiel, wenn er das Chiffrat dem richtigen Klartext zuordnen konnte.
Die Erfolgswahrscheinlichkeit des Angreifers $\mathcal{A}$ wird bezeichnet mit $Adv^{IND-CPA}(\mathcal{A}) := |P[\mathcal{A}] - \frac{1}{2}|$. Die Erfolgswahrscheinlichkeit des Angreifers ist die Wahrscheinlichkeit, mit der der Angreifer das Sicherheitsspiel gewinnt $P[\mathcal{A}]$ minus $\frac{1}{2}$. Damit ist die Erfolgswahrscheinlichkeit des Angreifers 0, wenn er beim Zuordnen des Chiffrats raten würde.

2.2.1.3. IND$-CPA

Diese Masterarbeit verwendet hauptsächlich die IND$-CPA (IND$) Sicherheitsgarantie. Bei dem Sicherheitsspiel für IND$ muss der Angreifer echte Chiffrate von zufälligen Bytestrings

unterscheiden können, um das Spiel zu gewinnen.
Es gibt einen Angreifer $\mathcal{A}$ und einen Challenger, der $\mathcal{A}$ eine Verschlüsselungsorakel zur Verfügung stellt. Der Challenger generiert zu Beginn des Spiels einen Schlüssel für das Verschlüsselungsverfahren des Sicherheitsspiels. Außerdem wählt der Challenger zufällig gleichverteilt ein $b \xleftarrow{\$} 0, 1$, das entscheidet, ob $\mathcal{A}$ echte Chiffrate von dem Orakel erstellt bekommt, oder ob das Orakel dem Angreifer zufällige Bytestrings anstelle von Chiffraten präsentiert. $\mathcal{A}$ kann beliebige Bytestrings (Klartext) erzeugen und diese an das Verschlüsselungsorakel übergeben. Gilt $b = 0$, dann verschlüsselt das Orakel den Klartext des Angreifers mit dem Schlüssel des Challengers und gibt das Chiffrat an den Angreifer zurück. Gilt $b = 1$, dann generiert das Orakel einen zufälligen Bytestring der dieselbe Länge, wie das echte Chiffrat hat und gibt diesen zufälligen Bytestring an den Angreifer zurück. Dieses Vorgehen kann der Angreifer beliebig oft wiederholen und muss dann eine Vermutung für den Wert von b aufstellen. Der Angreifer wählt seine Vermutung $z \in \{0, 1\}$ und gibt diese an den Challenger weiter. Entspricht die Vermutung z dem Wert des zu Beginn gewählten b, dann gewinnt $\mathcal{A}$ das Spiel.
Die Erfolgswahrscheinlichkeit eines Angreifers $\mathcal{A}$ wird bezeichnet mit $Adv^{IND\$}(\mathcal{A}) := |P[z = 0|b = 0] - P[z = 0|b = 1]|$.

2.2.2. Integrität

Der NIST Standard 1800-26A beschreibt Integrität als:

"Integrity – guarding against improper information modification or destruction and ensuring information non-repudiation and authenticity"

Das Manipulieren oder Vernichten von Daten kann in bestimmten Angriffsszenarien nicht verhindert werden, aber es gibt Mechanismen, mit denen dies erkannt werden kann und entsprechend reagiert werden kann. Im Falle dieser Masterarbeit bedeutet Integrität, dass es einer nicht autorisierten Person nicht möglich ist Daten unbemerkt zu verändern oder zu löschen.
Ein kryptografisches Verfahren, um Integrität zu gewährleisten ist ein Verfahren zum Generieren von Nachrichtenauthenfizierungscode (MAC). Ein MAC ist ein Wert, der über einem Bytestring berechnet wird und sich bei jeder Veränderung des Bytestrings ebenfalls verändert. Damit ist ein MAC ein Fingerabdruck eines Bytestrings und kann dazu benutzt werden, zu überprüfen, ob der Bytestring verändert wurde. In diesem Kapitel wird der Poly1305-AES128 Algorithmus vorgestellt, der auch von Restic verwendet wird.
Die Standard-Sicherheitsanforderung für Integrität ist Existential Unforgeability under Chosen Message Attack (EUF-CMA). Das Sicherheitsspiel für EUF-CMA wird ebenfalls in diesem Kapitel vorgestellt.

2.2.3. Poly1305-AES128

Poly1305-AES128 bezeichnet ein Verfahren zum Berechnen eines Nachrichtenauthenfizierungscodes (MAC) für einen Bytestring beliebiger Länge. Dazu wird die Blockchiffre AES128 verwendet. Für Poly1305-AES128 werden zwei Schlüssel verwendet. Ein 128 Bit langer Schlüssel für die Blockchiffre AES128 und ein 128 Bit langer Schlüssel für Poly1305 selbst. Für den Poly1305-Schlüssel gelten Einschränkungen, die bestimmte Bits des Schlüssels festlegen. Dadurch besitzt der Schlüsselraum für den Poly1305-Schlüssel nur eine Größe von 2^{106}, obwohl der Schlüssel aus 128 Bit besteht. Eine detaillierte Beschreibung für die Funktion Poly1305 wird in [2] vorgestellt.
Für Poly1305-AES128 wird eine Zahl N (Nonce) mit einer Größe von 16 Byte zufällig gewählt. Diese Zahl wird mit AES128 und einem Schlüssel k_{AES128} verschlüsselt und ergibt ebenfalls einen 16 Byte großen Bytestring $AES128_{k_{AES128}}(N)$. Der Bytestring, für den ein MAC erstellt werden soll, wird in 16 Byte große Datenblöcke unterteilt. Aus jedem Datenblock wird ein Polynom berechnet, wobei die einzelnen Bytes eines Datenblocks, die Koeffizienten des Polynoms darstellen. Die Polynome aller Datenblöcke des zu authentifizierenden Bytestrings D werden mit dem Schlüssel k_{Poly} und der Funktion $Poly$ zu einem 130 Bit langen Bytestring $Poly_{k_{Poly}}(D)$ verrechnet. Wie in Gleichung 2.2 zu sehen ist, ergibt sich der MAC MAC_D für einen Bytestring D aus der Addition von $Poly_{k_{Poly}}(D)$ und einer mit AES128 verschlüsselten zufälligen Zahl N.

$$MAC_D = (Poly_{k_{Poly}}(D) + AES128_{k_{AES128}}(N)) \bmod 2^{128} \tag{2.2}$$

Um einen MAC für einen Bytestring D zu verifizieren, wird der MAC mit Poly1305-AES128 erneut über dem Bytestring D mit der gleichen Zahl N berechnet und mit dem zu verifizierenden MAC verglichen. Unterscheiden sich die beiden MACs, wurde entweder der zu verifizierende MAC oder der Bytestring verändert.

2.2.3.1. EUF-CMA

Für viele Nachrichtenauthenfizierungscodes ist es der Standard eine EUF-CMA Sicherheitsgarantie zu fordern. Um die EUF-CMA Sicherheit zu beweisen wird ein Sicherheitsspiel mit zwei Parteien verwendet. Die eine Partei ist der Angreifer $\mathcal{A}$ und die andere ist der Challenger, der dem Angreifer ein Orakel für die Berechnung von MACs zur Verfügung stellt. Zum Start des Sicherheitsspiels generiert der Challenger einen Schlüssel für das zu analysierende MAC-Verfahren. Das Sicherheitsspiel wird in zwei Phasen unterteilt. Die erste Phase ist die Orakel-Phase und die zweite Phase heißt Challenge-Phase.
In der Orakel-Phase kann der Angreifer $\mathcal{A}$ beliebige Bytestrings erzeugen und dem Orakel übergeben. Das Orakel berechnet für den Bytestring des Angreifers mit dem Schlüssel des Challengers einen MAC und gibt diesen an den Angreifer zurück. Der Angreifer erhält abgesehen von seinen gewählten Klartexten und den dazu erzeugten MACs keine weiteren Informationen. Dieses Vorgehen kann der Angreifer beliebig oft wiederholen und geht dann in die Challenge-Phase über.

In der Challenge-Phase wählt der Angreifer einen beliebigen Bytestring D und erzeugt selbstständig einen MAC M' dazu und übergibt beides an den Challenger. Der Angreifer darf keinen Bytestring wählen, zu dem in der Orakel-Phase von dem Orakel bereits ein MAC berechnet wurde. M' wird auch Fälschung genannt. Der Challenger berechnet für D den MAC MAC_D und vergleicht diesen MAC mit M'. Ein Angreifer gewinnt das Sicherheitsspiel, wenn gilt $MAC_D = M'$.
Die Erfolgswahrscheinlichkeit des Angreifers $\mathcal{A}$ wird bezeichnet mit $Adv^{EUF-CMA}(\mathcal{A}) := P[M' = MAC_D]$. Die Erfolgswahrscheinlichkeit des Angreifers ist die Wahrscheinlichkeit, mit der der Angreifer erfolgreich einen MAC zu einem von ihm gewählten Bytestring fälschen konnte.

Multi-Challenge EUF-CMA (MC-EUF-CMA):
MC-EUF-CMA ist eine abgewandelte Version von EUF-CMA, bei der der Angreifer in der Challenge-Phase beliebig viele Bytestrings (Nachrichten) erzeugen darf, der er noch nicht in der Orakel-Phase verwendet hatte. Zu jedem dieser Bytestrings erzeugt der Angreifer selbst einen MAC und schickt die Nachrichten-MAC-Paare an den Challenger. Der Challenger berechnet für jeden Bytestring des Angreifers den echten MAC. Stimmt für mindestens ein Nachrichten-MAC-Paar der MAC des Angreifers mit dem vom Challenger für den Bytestring berechneten MAC überein, gewinnt der Angreifer das Spiel.
Damit ist das EUF-CMA Sicherheitsspiel das gleiche Spiel, wie das MC-EUF-CMA Sicherheitsspiel, wobei der Angreifer in der Challenge-Phase nur ein Nachrichten-MAC-Paar an den Challenger übergibt. Somit gilt insbesondere, dass ein Angreifer, der das EUF-CMA Spiel gewinnt, ebenfalls das MC-EUF-CMA Sicherheitsspiel gewinnt. Für die Erfolgswahrscheinlichkeit $Adv^{MC-EUF-CMA}(\mathcal{A})$, dass ein Angreifer $\mathcal{A}$ das Sicherheitsspiel MC-EUF-CMA gewinnt, folgt somit die Ungleichung $Adv^{EUF-CMA}(\mathcal{A}) \leq Adv^{MC-EUF-CMA}(\mathcal{A})$.

2.2.3.2. MAC$

Dieses Kapitel definiert ein IND$-artiges Sicherheitsspiel für Authentifizierungsverfahren, die zu einer Nachricht N einen MAC T berechnen. Es gibt für das Sicherheitsspiel MAC$ ebenfalls einen Challenger, einen Angreifer $\mathcal{A}$ und ein Authentifizierungsorakel.
Es sei $MAC = (Gen, Auth, Vrf)$ ein MAC Verfahren und λ der Sicherheitsparameter. Dann ist das Sicherheitsspiel MAC$($\mathcal{A}$, λ) zwischen Challenger C und Angreifer $\mathcal{A}$ wie folgt definiert:

1. C zieht $k \xleftarrow{\$} Gen(1^\lambda)$ und $b \xleftarrow{\$} \{0, 1\}$.
   Der Challenger C initialisiert das Orakel $O_{k,b} := Init(1^\lambda, k, b)$ und stellt dieses $\mathcal{A}$ zur Verfügung.
2. Der Angreifer kann nun bis zu $q(\lambda)$ Orakelzugriffe durchführen, wobei $q(\lambda)$ ein Polynom in λ ist:
   a) $\mathcal{A}$ wählt eine beliebige Nachricht N und gibt diese an sein Orakel $O_{k,b}(N)$.

b) $O_{k,b}$ berechnet $M_0 := Auth(N, k)$ und zieht uniform ein $M_1 \xleftarrow{\$} \{0,1\}^{|M_0|}$ gleicher Länge.

c) $O_{k,b}$ gibt M_b an den Angreifer zurück.

3. Der Angreifer gibt eine Zahl $z \in \{0,1\}$ aus.

$\mathcal{A}$ gewinnt genau dann das MAC$ Spiel, wenn gilt $b = z$. Die Erfolgswahrscheinlichkeit eines Angreifers $\mathcal{A}$ ist $Adv^{MAC\$}(\mathcal{A}, \lambda) := |Pr[b = z] - \frac{1}{2}|$. Ein MAC-Verfahren ist MAC$-sicher, wenn für alle PPT-Angreifer $\mathcal{A}$ gilt $Adv^{MAC\$}(\mathcal{A}, \lambda) \leq negl(\lambda)$.

2.2.4. PRP/PRF Switching Lemma

Dieses Lemma wird in [9] für Blockchiffren vorgestellt. Dieses Lemma kann benutzt werden, um die PRF-Eigenschaft einer Blockchiffre durch die PRP-Eigenschaft der gleichen Blockchiffre bei einer Wahrscheinlichkeitsabschätzung zu ersetzen. $Adv_E^{PRF}(\mathcal{B})$ bezeichnet dabei die Wahrscheinlichkeit, dass ein Angreifer $\mathcal{B}$ die Blockchiffre E von einer echten Zufallsfunktion unterscheiden kann. Während $Adv_E^{PRP}(\mathcal{B})$ die Wahrscheinlichkeit bezeichnet, dass ein Angreifer $\mathcal{B}$ die Blockchiffre E von einer echten Zufallspermutation unterscheiden kann. Für das Switching Lemma verwendet [9] die Abschätzung $|Adv_E^{PRP}(\mathcal{B}) - Adv_E^{PRF}(\mathcal{B})| \leq \frac{\sigma^2}{2^{k+1}}$ für die Differenz der beiden Erfolgswahrscheinlichkeiten des Angreifer B bezüglich der PRP-Eigenschaft und der PRF-Eigenschaft der Blockchiffre E. σ bezeichnet die Anzahl der Datenblöcke, die mit dieser Blockchiffre verschlüsselt wurden und k ist die Größe eines Datenblocks in Bits. Daraus resultiert das PRP/PRF Switching Lemma 2.3.

$$Adv_E^{PRF}(\mathcal{B}) \leq Adv_E^{PRP}(\mathcal{B}) + \frac{\sigma^2}{2^{k+1}} \tag{2.3}$$

2.2.5. Kollision & Birthday-Boundary

Restic verwendet für die verwendeten kryptografischen Primitive zufällig generierte einmal verwendete Zahlen (Nonce). Außerdem werden von Restic erzeugte Datenstrukturen eindeutig über den Hashwert der Datenstrukturen identifiziert. Sowohl bei der Generierung von Nonce und der Berechnung von Hashwerten kann es zu Kollisionen kommen, da Restic nicht kontrolliert, welche Nonces bisher schon verwendet wurden. Kollisionen sind der Tatsache geschuldet, dass der Raum über dem eine Nonce gezogen wird oder ein Hashwert berechnet wird nicht unendlich groß sind. Nach endlich vielen Durchführungen oder Eingaben wird es früher oder später zu einer Kollision kommen. Für die Nonce Ziehung bedeutet Kollision, dass eine Nonce gezogen wird, die zu einem früheren Zeitpunkt schon einmal gezogen wurde. Bei Hashfunktionen bedeutet Kollision, dass für zwei unterschiedliche Eingaben der gleiche Hashwert berechnet wird.
Birthday-Boundary bezeichnet eine Grenze für die Anzahl an Durchführungen, ab der die Wahrscheinlichkeit für eine Kollision nicht mehr vernachlässigbar ist. So ist für das

gleichverteilte Ziehen von n Bit Nonces die Birthday-Boundary $2^{\frac{n}{2}}$. Gibt es N verschiedene Nonces, ist die Birthday-Boundary nach $\sqrt{N}$ Nonce Ziehungen erreicht. Damit ergibt sich für das gleichverteilte Ziehen von n Bit Nonces eine Wahrscheinlichkeit für mindestens eine Kollision von $\frac{\sigma^2}{2^{n+1}}$, wobei σ die Anzahl der gezogenen Nonces ist. Ist die Birthday-Boundary überschritten beträgt die Kollisionswahrscheinlichkeit bei jeder darauf folgenden Nonceziehung oder Hashberechnung mehr als 50%.
In vielen kryptografischen Verfahren führt eine Kollision zu einem Verlust der Sicherheit, wodurch eine akzeptable Kollisionswahrscheinlichkeit weit unter 50% liegt. Im Laufe der Masterarbeit wird eine Obergrenze für die Kollisionswahrscheinlichkeit festgelegt und Restics kryptografische Primitive anhand dieser Grenze analysiert.

2.2.6. Granularität

Bei der Betrachtung von Restics Sicherheit wird in dieser Masterarbeit zwischen zwei Ebenen unterschieden. Sicherheit auf Primitiv-Ebene bezeichnet die Sicherheitsgarantien der von Restic verwendeten kryptografischen Primitive und wird in Kapitel 4 betrachtet. Sicherheit auf Repository-Ebene bezeichnet Sicherheitsgarantien, die Restic als kryptografisches Protokoll bietet unter der Betrachtung eines Repositorys und darauf ausgeführten Restic-Befehlen. Die Betrachtung von Restics Sicherheit auf Repository-Ebene findet spielbasiert in Kapitel 5 statt.

2.3. Restic

Restic [8] ist eine Backup-Software zur schnellen und effizienten Datensicherung und Datenwiederherstellung. Dieses Kapitel beinhaltet alle Informationen über Restic, die im Laufe dieser Masterarbeit verwendet werden. Es werden Begriffe und Konzepte erklärt, die im Kontext von Restic relevant sind. Es werden die von Restic eingesetzten Primitive zur Datenverarbeitung und Kryptografie vorgestellt. Außerdem werden die Datenstrukturen erklärt, die von Restic verwendet werden, um ein schnelles und effizientes Verarbeiten von Daten zu gewährleisten.

2.3.1. Begriffe & Konzepte von Restic

In diesem Kapitel werden alle Begriffe und Konzepte von Restic vorgestellt, die im Laufe dieser Masterarbeit verwendet werden, um Restic als kryptografisches Protokoll zu modellieren.

2.3.1.1. Restic-Befehle

In dieser Masterarbeit wird oft von Restic-Befehlen gesprochen, wenn es um die Ausführung bestimmter Funktionalitäten von Restic geht. Die Software Restic stellt Funktionen zur Erstellung und Wiederherstellung von Backups zur Verfügung, sowie Funktionen zum Verwalten der erzeugten Backup-Daten. Drei Restic-Befehle, die diese Funktionen umsetzen, werden in dieser Masterarbeit genauer betrachtet. Jeder dieser Restic-Befehle besitzt einen gleichnamigen Kommandozeilenbefehl mit dem Restic angesprochen werden kann. Daher wird die Ausführung einer Funktion, die Restic anbietet, als die Ausführung eines Restic-Befehls bezeichnet.
Bei den drei betrachteten Befehlen handelt es sich um den *backup* Befehl, zum Erstellen eines Backups, den *restore* Befehl zum Wiederherstellen eines Backups und den *prune* Befehl zum Verwalten von erzeugten Backup-Daten. Restic bietet noch weitere Befehle an, die aber nicht intensiver von dieser Masterarbeit betrachtet werden, da sie nur kleiner Funktionalitäten umsetzen oder ihre Funktionalitäten bereits von den drei betrachteten Befehlen abgedeckt werden.

2.3.1.2. Repository

Bei der Verwendung von Restic wird ein Speicherort angelegt, an dem alle verschlüsselten Benutzerdaten, von denen ein Backup erstellt wurde, gespeichert werden. Dieser Speicherort wird Repository genannt. Ein Repository enthält weitere Restic-Datenstrukturen (siehe Kapitel 2.3.4), die zur Verarbeitung der Backupdaten und zur Bereitstellung der Funktionalitäten von Restic benötigt werden. Ein Repository kann mehrere Backups enthalten, sowie Backups von unterschiedlichen Verzeichnispfaden.
Zu Beginn eines Restic-Befehls muss der Benutzer angeben, welches Repository für diesen Restic-Befehl verwendet wird.

2.3.1.3. Masterkey

Jedes Repository besitzt einen Masterkey, der aus einem AES256 Schlüssel, und einem Poly1305 Schlüssel zusammen mit einem AES128 Schlüssel besteht. Mit diesem Masterkey können sämtliche Daten des Repositorys verschlüsselt, entschlüsselt und authentifiziert werden. Jede Key-Datenstruktur 2.9 eines Repositorys enthält den verschlüsselten und authentifizierten Masterkey des Repositorys als Eintrag *data*. Der Masterkey bleibt über die gesamte Lebensdauer eines Repositorys gleich und wird niemals geändert.
Die Masterkey Datenstruktur ist keine direkt von Restic verwendete Datenstruktur und wird daher auch nicht weiter modelliert. Sie besteht jedoch aus einem 32 Bit Key mk_{AES256} für AES256 und einem 32 Bit Key für Poly1305, der aus zwei konkatenierten 16 Bit Keys mk_{Poly} und mk_{AES128} besteht.

2.3.1.4. Mehrbenutzermodus

Restic bietet die Möglichkeit, dass mehrere verschiedene Benutzer auf dasselbe Repository zugreifen können und dabei jeder Benutzer sein eigenes Passwort verwenden kann. Damit ein Mehrbenutzermodus funktioniert, speichert Restic den verschlüsselten und authentifizierten Masterkey für jeden Benutzer in einer eigenen Datei (siehe Kapitel 2.3.4.3). Aus einem Benutzerpasswort wird ein Userkey erzeugt, mit dem der Masterkey verschlüsselt und authentifiziert wird. Damit können nun verschiedene Benutzer mit verschiedenen Passwörtern auf dasselbe Repository zugreifen.

2.3.1.5. Parteien

Diese Masterarbeit betrachtet die zu analysierenden Restic-Befehle als Zweiparteienprotokolle. Die erste Partei wird als Resticsystem S_{Restic} bezeichnet. S_{Restic} stellt einen Computer dar, auf dem das Programm Restic [8] installiert ist und von einem Benutzer bedient wird. Wann immer diese Masterarbeit von Restic spricht, ist damit das Programm Restic [8] auf dem Resticsystem gemeint. Die zweite Partei wird als Backupsystem S_{Backup} bezeichnet. S_{Backup} stellt ein System dar, auf dem das von Restic erstellte Repository gespeichert wird und für S_{Backup} zu jeder Zeit einsehbar ist. S_{Backup} kann das gleiche System wie S_{Restic} sein, ein anderer Computer oder ein anderer Server. S_{Backup} kann ein System sein, auf dem der Restic-Rest-Server [7] installiert ist, aber auch ein System auf dem kein spezielles Restic-Backend installiert sein muss. Das Backupsystem stellt nur ein System dar, mit dem das Resticsystem kommunizieren können muss. Alle Systeme, die die Voraussetzungen eines Backupsystems erfüllen, befinden sich in dem Kapitel *Backend* in der Datei README.md des Restic-GitHub-Repos [8] aufgelistet.

2.3.1.6. Parteienkommunikation

Für die Modellierung von Restic als kryptografisches Protokoll wird die Kommunikation zwischen S_{Restic} und S_{Backup} als instantan angesehen. Es werden daher auch keine Netzwerkprotokolle in die Modellierung miteinbezogen. Der Fokus liegt auf den von Restic erzeugten Daten, deren Struktur und dem Datenaustausch zwischen S_{Restic} und S_{Backup}. Ein Datenaustausch zwischen den zwei Systemen wird immer von S_{Restic} aus gestartet.
Für den Pseudocode wird die Kommunikation zwischen S_{Restic} und S_{Backup} in zwei Stufen unterteilt. `send(N)` bezeichnet das Senden einer Nachricht oder Anfrage mit Inhalt N von S_{Restic} an S_{Backup}. Die Antwort von S_{Backup} an S_{Restic} für den letzten `send(N)` Aufruf wird als Rückgabewert von `receive()` dargestellt. `delete(N)` ist eine spezielle Version von `send(N)`, die keine Antwort erwartet, sondern S_{Backup} auffordert eine Datenstruktur aus dem Repository zu löschen.
Es werden zwei Nachrichtenarten für N betrachtet. `send(Type)` fordert kategorisch alle Datenstrukturen eines Repositorys an, die vom Typ $Type$ sind. $Type$ bezeichnet hierbei einen Typ von Restic-Datenstrukturen (siehe Kapitel 2.3.4).
`send(ID, Type)` fordet speziell eine Datenstruktur vom Typ $Type$ an, die die Restic-ID ID

hat. Sobald Restic die Antwort empfängt wird überprüft, ob die geschickte Datenstruktur wirklich der angeforderten Restic-ID entspricht.
Wenn etwas mit `send(N)` gesendet wurde und `receive()` nichts empfängt, dann kommt es bei Restic immer zu einem Fehler (error) und der Restic-Befehl beendet seine Ausführung.

2.3.1.7. Target-Pfade

Beim Erstellen eines Backups werden Restic Dateipfade oder Verzeichnispfade auf S_{Restic} übergeben, von denen ein Backup erstellt werden soll. Diese Pfade werden Target-Pfade genannt.

2.3.1.8. Parent-Snapshot

Jedes Mal, wenn ein Backup von Restic erstellt und gespeichert wird, wird passend dazu eine Snapshot-Datenstruktur (siehe Tabelle 2.12) auf dem Repository gespeichert. Bei der Durchführung eines Backups, lädt Restic einen bereits im Repository existierenden Snapshot, dessen Target-Pfade alle Target-Pfade des aktuellen *backup* Befehls enthalten. Dieser Snapshot wird Parent-Snapshot für die Ausführung dieses *backup* Befehls genannt. Gibt es keinen Snapshot, dessen Target-Pfade alle Target-Pfade des aktuellen *backup* Befehls enthalten, existiert für diesen Restic-Befehle kein Parent-Snapshot. Der *backup* Befehl kann aber auch ohne Parent-Snapshot ausgeführt werden, wohingenen der *restore* Befehl ohne Parent-Snapshot seine Befehlsausführung mit einem Fehler beendet.

2.3.1.9. Verzeichnisbaum

Beim Erstellen eines Backups werden Target-Pfade an Restic übergeben. Das Backup wird daraufhin von allen Verzeichnissen und Dateien erstellt, für die ein Präfix ihres Pfades in Form eines Target-Pfads übergeben wurde. Um diese Struktur konsistent abzubilden und zu speichern, legt Restic bei Backups und Restores einen Verzeichnisbaum an. Dazu werden die Target-Pfade bis auf das Root-Verzeichnis des Betriebssystems zurückverfolgt. Genau an diesem Root-Verzeichnis beginnt ein Verzeichnisbaum. So beginnt ein Verzeichnisbaum eines UNIX-Systems bei dem Pfad *"/"*. Ein Verzeichnisbaum bildet die Struktur eines Verzeichnissystems ab und es existiert für jedes Verzeichnissystem genau ein Verzeichnisbaum und umgekehrt.
Diese Masterarbeit betrachtet einen Verzeichnisbaum als informationstheoretischen gerichteten Graphen $T := (V, E)$, der aus einer Knotenmenge V und Kantenmenge $E \subseteq V \times V$ besteht. Für einen Verzeichnisbaum T gilt, T ist zusammenhängend und es gilt $\forall v \in V : |\{(v_i, v_j) \in E : v_j = v\}| \leq 1$. Außerdem ist jeder Knoten aus V von einem festen Knoten $v_{root} \in V$ über Kanten aus E erreichbar. Dadurch gilt auch $\forall (v_i, v_j) \in E : v_j \neq v_{root}$. Dieser Knoten v_{root} wird Wurzel oder Wurzelknoten des Baums T genannt. Alle Knoten $v \in V$, für die gilt $\exists v' \in V : (v, v') \in E$, werden innere Knoten von T genannt. Für eine Kante $(v, v') \in E$ wird v' als Kindknoten von v bezeichnet und v wird für v' als Elternknoten bezeichnet. Alle

Knoten $v \in V$, für die gilt $\forall(v_i, v_j) \in E : v_j \neq v$, werden Blätter oder Blattknoten von T genannt.
Die inneren Knoten eines Verzeichnisbaums repräsentieren einzelne Verzeichnisse und die Blätter eines Verzeichnisbaums repräsentieren entweder leere Verzeichnisse oder Dateien. In dieser Masterarbeit werden Knoten auch als Datei oder Verzeichnis bezeichnet, abhängig davon was diese Knoten in dem zugehörigen Verzeichnissystem repräsentieren. Eine Kante $(v, v') \in E$ bedeutet, dass das Verzeichnis/Datei v' sich direkt in dem Verzeichnis v des zugehörigen Verzeichnissystems befindet. Der Wurzelknoten v_{root} eines Verzeichnisbaums repräsentiert immer das Root-Verzeichnis des Verzeichnissystems. Bei einem Verzeichnissystem mit $m \in \mathbb{N}$ Verzeichnissen und $n \in \mathbb{N}$ Dateien lässt sich ohne Beschränkung der Allgemeinheit (o.B.d.A.) V schreiben als $V := \{v_1, \ldots, v_m, v_{m+1}, \ldots, v_{m+n}\}$, wobei gilt $v_1, \ldots, v_m$ repräsentieren Verzeichnisse und $v_{m+1}, \ldots, v_{m+n}$ repräsentieren Dateien. Des Weiteren gilt o.B.d.A. $v_1 := v_{root}$. Sofern nicht anders angegeben, benutzt diese Masterarbeit diese Definition von V und v_1.
Ein Verzeichnisbaum zu dem virtuellen Verzeichnissystem eines Backups ist ein Subgraph des Verzeichnisbaums des realen Verzeichnissystems. Der Verzeichnisbaum eines virtuellen Verzeichnissystems ist ebenfalls ein Baum, da das Verzeichnissystem zusammenhängend ist und durch die hierarchische Ordnung keine Zyklen zulässt. Außerdem ist in einem virtuellen Verzeichnissystem immer das Root-Verzeichnis des Betriebssystems enthalten und damit existiert auch ein Root-Knoten in dem Verzeichnisbaum. Durch die Konstruktion des virtuellen Verzeichnissystems entspricht der Verzeichnisbaum dieses Verzeichnissystems dem Verzeichnisbaum, der von Restic für ein Backup konstruiert wird. Diese Eigenschaft wird in der Modellierung der Sicherheitsspiele ausgenutzt, um die Restic-Befehle als abstrakte Protokolle zu modellieren.
In Restic wird ein Verzeichnisbaum durch die Tree-Blob Datenstrukturen eindeutig abgebildet und gespeichert. Wie das geschieht, wird in Kapitel 2.3.4.8 erklärt. Die Dateiinhalte werden in mehrere Data-Blob Datenstrukturen zerlegt und gespeichert. Der Zusammenhang zwischen Knoten eines Verzeichnisbaums und Data-Blobs wird in Kapitel 2.3.4.7 erklärt.

2.3.2. Verfahren zur Datenverarbeitung

Dieses Kapitel stellt die von Restic verwendeten Primitive zur Datenverarbeitung vor. Restic verwendet nicht nur Verschlüsselungsverfahren und Authentifizierungsverfahren, sondern auch ein Komprimierungsverfahren, um Speicherplatz zu sparen, einen Hash-Algorithmus zum eindeutigen Identifizieren von Datenstrukturen und ein Chunking-Verfahren für die Deduplikation. Um Restic so exakt wie möglich als kryptografisches Protokoll modellieren zu können und die Sicherheit von Restic zu beweisen, müssen diese Primitive vorher definiert sein.

2.3.2.1. Komprimierung

Viele Daten, die das Resticsystem erzeugt oder verarbeitet, werden zuerst mit einem deterministischen ZSTD-Algorithmus [6] komprimiert oder dekomprimiert. Es handelt sich bei

Restics level (intern)	Restic Synonyme (extern)	Meta Äquivalent
1	fastest	zstd level 1
2	auto	zstd level 3
3	better	zstd level 7
4	max	zstd level 11

Table 2.4.: Ungefähre Entsprechung der Restic ZSTD-Komprimierungslevel zu denen von Metas offizieller Implementierung

dem von Restic verwendeten ZSTD-Algorithmus um einen Algorithmus [6] aus der GO-Standardbibliothek. Bei dem von Restic verwendete Komprimierungsalgorithmus können bis zu vier Komprimierungslevel eingestellt werden. Diese Komprimierungslevel unterscheiden sich jedoch von den Leveln der Meta-ZSTD-Implementierung [5]. Eine Gegenüberstellung der von Restic verwendeten internen und externen Bezeichnung der Komprimierungslevel, sowie den Meta-Pendants befindet sich in Tabelle 2.4.

Tabelle 2.4 zeigt, welche internen Komprimierungslevel welchen externen Restic Synonymen entsprechen. Die internen Restic Komprimierungslevel bezeichnen dabei den Wert im Quelltext von Restic. Die externen Restic Synonyme können als Argumente für die *compression* Option von Restics Kommandozeilenbefehle verwendet werden, um das Komprimierungslevel von Restic zu konfigurieren. Für die Komprimierung eines Bytestrings B mit dem Komprimierungslevel m wird die folgende Notation verwendet: $Comp^{m}_{zstd}(B) = B_{comp}$. Der komprimierte Bytestring von B wird als B_{comp} bezeichnet.

2.3.2.2. Verschlüsselung und Entschlüsselung

Restic verwendet zum Verschlüsseln von Daten eine AES256 Blockchiffre im Counter-Mode (CTR) Betriebssystem (siehe Kapitel 2.2.1.1). Für die gesamte Lebensdauer eines mit Restic erstellten Repositorys wird der gleiche 32 Byte lange Schlüssel mk_{AES256} verwendet, um Daten aus dem Repository zu verschlüsseln und zu entschlüsseln. Der Initialisierungsvektor (IV) für die Verschlüsselung besitzt eine Länge von 16 Byte und wird als Nonce zufällig aus einem Raum von 2^{128} Möglichkeiten gezogen. Ebenso beträgt die Blockgröße der verschlüsselten Blöcke eines Bytestrings auch 16 Byte. Für das Verschlüsselung eines Bytestrings B und mit dem Initialisierungsvektor IV wird folgende Notation verwendet:
$Enc^{IV}_{AES256}(B, mk_{AES256}) = Ciphertext^{IV}_{B}$
Da, wie oben erwähnt, immer der gleiche Schlüssel für ein Repository benutzt wird, wird der Schlüssel nicht immer explizit in der Notation verwendet:
$Enc^{IV}_{AES256}(B) = Enc^{IV}_{AES256}(B, mk_{AES256})$
Restic verwendet kein Padding im letzten Datenblock, für Bytestrings, deren Größe kein Vielfaches von 16 Byte ist. Stattdessen wird der letzte Block eines Chiffrats auf die Länge des zugehörigen Datenblocks gekürzt. Damit ergibt sich für einen Datenblock D_i der Länge $m < 16$ Byte der Chiffratblock $C_i = (D_i \oplus AES256_{mk_{AES256}}(IV + i))[:16 - m]$ ohne die letzten $16 - m$ Bytes. Daraus folgt auch, dass in Restic ein Chiffrat immer genauso lang ist wie der Bytestring, zu dem das Chiffrat gehört.

Beim AES256-CTR Verfahren entspricht der Verschlüsselungsalgorithmus dem Entschlüsselungsalgorithmus mit der Ausnahme, dass der IV nicht neu gewählt wird (siehe Kapitel 2.2.1.1). Dennoch wird für eine bessere Lesbarkeit diese Notation für den Entschlüsselungsalgorithmus verwendet:
$Dec^{IV}_{AES256}(Ciphertext^{IV}_B) = Dec^{IV}_{AES256}(Ciphertext^{IV}_B, mk_{AES256}) = B$

2.3.2.3. Authentifizierung

Zur Authentifizierung von Daten verwendet Restic Poly1305 zusammen mit einer AES128 Blockchiffre, um Nachrichtenauthenfizierungscodes (MACs) für die Daten zu berechnen (siehe Kapitel 2.2.3). Für die gesamte Lebensdauer eines mit Restic erstellten Repositorys wird der gleiche 16 Byte lange AES128 Schlüssel mk_{AES128} und der gleiche 16 Byte lange Poly1305 Schlüssel mk_{Poly} verwendet. Restic authentifiziert nur Daten, die von Restic vorher auch verschlüsselt wurden. Daher verwendet Restic als IV für Poly1305-AES128 den gleichen 16 Byte langen IV, der auch für die Verschlüsselung der Daten mit AES256-CTR verwendet wurde. Für die Berechnung eines MACs $MAC_{Ciphertext^{IV}_B}$ für einen verschlüsselten Bytestring $Ciphertext^{IV}_B$ mit dem Initialisierungsvektor IV wird die folgende Notation verwendet:
$Auth^{IV}_{Poly1305}(Ciphertext^{IV}_B, mk_{Poly}, mk_{AES128}) = MAC^{IV}_{Ciphertext^{IV}_B}$
Der von Restic berechnete MAC ist immer 16 Byte groß. Da, wie oben erwähnt, immer die gleichen Poly1305 und AES128 Schlüssel für ein Repository benutzt werden, werden die Schlüssel nicht immer explizit in der Notation verwendet:
$Auth^{IV}_{Poly1305}(Ciphertext^{IV}_B) = Auth^{IV}_{Poly1305}(Ciphertext^{IV}_B, mk_{Poly}, mk_{AES128})$
Die Verschlüsselung, Authentifizierung und Zusammensetzung des Chiffrats ist in Funktion 16 modelliert. Sofern nicht anders angegeben, verwendet die Funktion 16 den Masterkey (mk_{AES256}, mk_{Poly}, mk_{AES128}) des Repositorys als Schlüssel.

2.3.2.4. Verfikation

Sehr oft wird in dieser Masterarbeit davon gesprochen, dass Restic verschlüsselte Daten oder von S_{Backup} empfangene Nachrichten verifiziert. Restic zerlegt die verschlüsselten Daten in einen Initialisierungsvektor IV, ein Chiffrat $Ciphertext$ und einen Nachrichtenauthenfizierungscode MAC: $IV \parallel Ciphertext \parallel MAC$. IV bezeichnet dabei die ersten 16 Byte der verschlüsselten Daten und MAC die letzten 16 Byte der verschlüsselten Daten. Restic berechnet daraufhin den MAC für $Ciphertext$ mit IV und kann die verschlüsselten Daten erfolgreich verifizieren, wenn gilt $Auth^{IV}_{Poly1305}(Ciphertext) = MAC$.
Die Verifikation und Entschlüsselung verschlüsselter Daten findet immer zusammen statt und wird in Funktion 17 modelliert. Sofern nicht anders angegeben, verwendet Funktion 17 den Masterkey (mk_{AES256}, mk_{Poly}, mk_{AES128}) des Repositorys als Schlüssel.

2.3.2.5. Hash-Algorithmus

Jeder Restic-Datenstruktur wird eine eindeutige ID zugeordnet, mit der man diese Datenstruktur eindeutig referenzieren kann (siehe Kapitel 2.3.3.2). Zur Berechnung der ID einer Datenstruktur wird der SHA256 Hash-Wert von der Datenstruktur berechnet.
Wenn S_{Restic} eine Datenstruktur von S_{Backup} mit der ID dieser Datenstruktur anfordert, berechnet Restic den SHA256 Hash-Wert der empfangenen Daten und vergleicht diesen mit der ID der angeforderten Datenstruktur.

2.3.2.6. Deduplikation & Chunking

Eins der Hauptziele von Restic ist es, die Größe des Repository so klein wie möglich zu halten. Dabei ist die Fähigkeit zur Deduplikation besonders wichtig. Deduplikation bedeutet, dass bei einem Mehrfachvorkommen von identischen Daten, diese Daten nur ein Mal im Repository gespeichert werden und die restlichen Male referenziert werden, anstatt die Daten erneut zu speichern. Backup-Programme, die nicht über diese Fähigkeit verfügen, führen oft zu Backups mit einer höheren Speichergröße und längeren Zeiten zur Backup-Erstellung. In dem Fall von Restic ist mit Deduplikation gemeint, dass bereits im Repository existierende Data-Blobs oder Tree-Blobs (siehe Kapitel 2.3.4) nicht erneut gespeichert werden, sondern lediglich referenziert werden.
Um Deduplikation bei Benutzerdaten effizient umsetzen zu können, wird eine Mechanik namens Chunking verwendet, um größere Daten in kleinere Blöcke (Chunks) zu unterteilen. Dadurch steigt die Chance auf mehrfach vorkommende Chunks, wodurch die benötigte Speichergröße reduziert wird. Bei der Änderung oder dem Hinzufügen neuer Daten müssen nur veränderte Chunks zum Repository hinzugefügt werden und beispielsweise nicht eine ganze Datei. Restic benutzt ebenfalls ein Chunkingverfahren, um Deduplikation umzusetzen.
Das Chunkingverfahren von Restic erzeugt Chunks mit einer Größe zwischen 512 KiByte und 8 MiByte. In [10] wird unter anderem auch das Chunkingverfahren von Restic untersucht und dessen Sicherheit analysiert. [10] kommt zu dem Schluss, dass das Chunkingverfahren von Restic sicherheitsrelevante Schwachstellen aufweist. Außerdem stellt es ein alternatives Chunkingverfahren vor, das diese Schwachstellen beheben würde. Die Sicherheit von Restics Chunkingverfahren wird unter anderem auch in [1] aufgegriffen. Für diese Masterarbeit liefert [1] allerdings keine neuen Erkenntnisse, die nicht auch durch [10] gewonnen wurden.

2.3.3. Repräsentation von Restics Datenstrukturen

Dieses Kapitel beschreibt in welchem Format Restics Datenstrukturen gespeichert werden und wie die Datenstrukturen in einem Repository repräsentiert sind. Es wird außerdem erklärt wie Restic Datenstrukturen referenziert (Restic-ID) und welche Speichergrößenbeschränkungen die einzelnen Datenstruktur-Typen besitzen.

Datenstruktur D	ID ist SHA256 Hash von ...
Config & Pack-Header	-
Index & Lock & Snapshot	`encrypt(comp(json(`D`)))`
Key	`json(`D`)`
Tree-Blob	`json(`D`)`
Data-Blob	D
Pack	D

Table 2.5.: Wie berechnet Restic die ID einer Datenstruktur

2.3.3.1. JSON-Format

Einige Datenstrukturen von Restic werden an S_{Backup} gesendet, damit sie im Repository gespeichert werden. Dazu werden die meisten von Restics Datenstrukturen zuerst zu einem JSON-String formatiert bevor sie verschlüsselt werden. Dieser JSON-String enthält keine Leerzeichen und keine Zeilenumbrüche und enthält nur reine Variablennamen mit ihren Werten im JSON-Format. Alle Datenstrukturen, die Restic in einem Repository speichert, bis auf Data-Blobs und Packs, werden vorher in einen JSON-String formatiert.
Diese Masterarbeit verwendet die Notation `json(D)`, um anzuzeigen, dass eine Datenstruktur D zu einem JSON-String konvertiert wurde.

2.3.3.2. Restic-IDs

Alle von Restics Datenstrukturen bis auf die Config des Repositorys besitzen eine eindeutige 32 Byte lange ID. Die ID einer Datenstruktur wird mit dem Hash-Algorithmus SHA256 berechnet. Die Tabelle 2.5 gibt an, wie die ID einer Restic-Datenstruktur berechnet wird. Für viele Datenstrukturen wird die Datenstruktur zunächst in einen JSON-String umgewandelt, mit ZSTD komprimiert und dann mit Funktion 16 verschlüsselt, bevor der Hash-Wert berechnet wird. Die Komprimierung in Tabelle 2.5 ist aus leserlichen Gründen hier mit `comp(...)` dargestellt. Die ID eines Data-Blobs oder Tree-Blob wird immer über deren Klartext berechnet, da dies die Performance von Restic bei vielen Vergleichen von IDs dieser Datenstrukturen verbessert.
Die ID wird immer durch den Klartext oder das Chiffrat berechnet, aber niemals mit den IVs oder MACs. Im Pseudocode wird die Funktion `id()` benutzt.

2.3.3.3. Repräsentation der Datenstrukturen im Repository

Jede Datenstruktur, die in einem Repository gespeichert wird, außer Tree-Blobs oder Data-Blobs, wird in einer eigenen Datei gespeichert. Der Dateiname dieser Datei entspricht der ID der Datenstruktur. Nur der Name der Config-Datei C des Repositorys ist *config* und keine ID. Jedes Repository ist ein eigenes Verzeichnis auf dem S_{Backup}. In dem Repository-Verzeichnis befindet sich die Config-Datei und die folgenden Unterverzeichnisse: *data*, *index*, *keys*, *locks* und *snapshots*. In diesen Unterverzeichnissen werden die Dateien der

Typ	Länge (min)	Länge (max)
Config	120 Byte	150 Byte
Index	200 Byte	12 MiByte
Lock	100 Byte	130 Byte + $n_{Lock} \approx$ 230 Byte
Snapshot	300 Byte	1 KiByte + $n_{Snapshot} \leq$ 4 GiByte
Data-Blob	512 KiByte	8 MiByte
Tree-Blob	300 Byte	4 GiByte
Node	300 Byte	400 Byte + n_{Node}
Pack-Header	37 Byte	16 MiByte
Blob-Header	37 Byte	41 Byte

Table 2.6.: Größe der einzelnen unkomprimierten Restic-Datenstrukturen

zugehörigen Datenstrukturen gespeichert. In dem *data* Unterverzeichnis werden die Pack-Dateien $P_1, \ldots, P_{max}$ gespeichert. In dem *index* Unterverzeichnis werden die Index-Dateien $I_1, \ldots, I_{max}$ gespeichert. Das *keys* Unterverzeichnis speichert die Key-Dateien $K_1, \ldots, K_{max}$ des Repositorys und das *locks* Unterverzeichnis speichert alle Lock-Dateien $L_1, \ldots, L_{max}$. In dem *snapshot* Unterverzeichnis werden die Snapshot-Dateien $SS_1, \ldots, SS_{max}$ gespeichert. Alle Daten, die einem Repository durch die Ausführung hinzugefügt werden, werden mit einer Tilde von den Daten unterschieden, die bereits in dem Repository gespeichert sind. So bezeichnet beispielsweise $\tilde{K}_{new}$ eine neue Key-Datei, die dem Repository durch einen Befehl hinzugefügt wird. Daten werden nur für den Befehl, durch den sie hinzugefügt werden mit einer Tilde markiert und gelten nach Ausführung des Befehls als fester Teil des Repositorys. So ist für nachfolgende Befehle dieselbe Key-Datei Teil der Liste aller Key-Dateien $K_1, \ldots, K_{max}$ des Repositorys.

2.3.3.4. Datenstrukturgrößen

Die Tabelle 2.6 gibt eine untere und obere Schranke für die Größe aller unkomprimierten Restic-Datenstrukturen (siehe Kapitel 2.3.4) an, bevor sie verschlüsselt und authentifiziert werden. Dabei wird die Größe aller Datenstrukturen bis auf die Data-Blobs in deren JSON-Format (siehe Kapitel 2.3.3.1) betrachtet.
Eien Pack-Datenstruktur kann maximal 128 MiByte an verschlüsselten und authentifizierten Blob-Daten speichern. Die Pack-Datenstruktur wird allerdings nicht als Ganzes in Tabelle 2.6 betrachtet. Stattdessen werden die einzelnen Blobs betrachtet und der Pack-Header, sowie die darin enthaltenen Blob-Header der Pack-Datenstruktur. Der Blob-Header ist für unkomprimierte Blobs 37 Byte groß und für komprimierte Blobs 41 Byte.
Ein Tree-Blob kann die größte unkomprimierte Datenstruktur bilden. Erreicht die Größe eines Tree-Blob mindestens 4GiByte, kann Restic diesen Tree-Blob nicht korrekt verarbeiten und stoppt. Die Größe eines Tree-Blob wird durch die enthaltenen Nodes festgelegt. Daher enthält Tabelle 2.6 ebenfalls Abschätzungen für die Größe von Nodes in einem Tree-Blob.

Die Datenstrukturen Lock, Snapshot und Node haben keine konkrete maximale Größenbeschränkung. Die Größe dieser Datenstrukturen wird durch die Variablen n_{Lock}, $n_{Snapshot}$ und n_{Node} bes-

timmt. n_{Lock} repräsentiert die Größe der beiden Felder *hostname* und *username* der Lock-Datenstruktur (siehe Tabelle 2.11). Sowohl *hostname* als auch *username* werden von dem Resticsystem und dem Benutzer bestimmt, der einen Restic-Befehl aufruft. $n_{Snapshot}$ repräsentiert die Größe der beiden Felder *paths* und *excludes* der Snapshot-Datenstruktur (siehe Tabelle 2.12). Die Größe der beiden Felder *paths* und *excludes* wird durch den Benutzer, der den Restic-Befehl aufruft bestimmt.
Sowohl für n_{Lock} als auch $n_{Snapshot}$ kann eine Abschätzung getroffen werden, die allerdings mit Vorsicht behandelt werden sollte aus den oben genannten Gründen. Bei n_{Lock} handelt es sich nur um zwei Strings, die das Resticsystem und den Benutzer dieses System repräsentieren. Unter realistischen Umständen sollte jeder der beiden Strings nicht länger als 50 Zeichen groß sein. Damit gilt in diese Masterarbeit $n_{Lock} := 100Byte$. Für $n_{Snapshot}$ ist es nicht trivial eine gute Abschätzung zu finden. Die meisten Pfade in *paths* gehören zu ganzen Verzeichnissen mit einer deutlich größeren Menge als 1 MiByte an Benutzerdaten. Damit sollten die folgenden Abschätzungen eine realistische Obergrenze für $n_{Snapshot}$ darstellen. Wird ein Backup mit mehr als 100 GiByte von Benutzerdaten erstellt, geht diese Masterarbeit von einem Eintrag in *paths* und in *excludes* pro 1 MiByte an Benutzerdaten aus. Wird ein Backup mit weniger als 100 GiByte von Benutzerdaten erstellt, geht diese Masterarbeit von einem Eintrag in *paths* und in *excludes* pro 64 KiByte an Benutzerdaten aus. Die Länge eines Pfades kann mit 256 Zeichen pro Pfad nach oben Abgeschätzt werden. Damit ergibt sich eine Größe von 256 Byte pro Eintrag in *paths* und *excludes* und die folgenden Abschätzungen für $n_{Snapshot}$. Bei einem Backup mit mehr als 100 GiByte von Benutzerdaten entspricht $n_{Snapshot} := 2^{-11} \cdot size(Data_{User})$, wobei $size(Data_{User})$ die Größe der Benutzerdaten ist. Bei einem Backup mit weniger als 100 GiByte von Benutzerdaten entspricht $n_{Snapshot} := 2^{-7} \cdot size(Data_{User})$. Mit diesen Formeln ist die Größe einer Snapshot-Datenstruktur unbegrenzt für eine steigene Größe der Benutzerdaten. Die Formeln für $n_{Snapshot}$ stellen jedoch eine starke Überabschätzung da, die für eine unbegrenzte Größe von Benutzerdaten nicht realistisch skaliert. Daher wird der maximale Wert von 1 KiByte $+ n_{Snapshot}$ auf 4 GiByte gesetzt, der durch die Formel ungefähr bei einer Menge von 8 TiByte Benutzerdaten eintreten würde.
n_{Node} repräsentiert die Größe des *content* Feldes der Node-Datenstruktur (siehe Tabelle 2.14). Es ergibt sich die Gleichung $n_{Node} = 67 \text{ Byte} \cdot Data - Blobs_{Node}$, wobei $Data - Blobs_{Node}$ die Anzahl er Einträge im *content* Feld der Node-Datenstruktur ist.

2.3.4. Restics Datenstrukturen im Detail

Dieses Kapitel beschreibt den Aufbau und die Bedeutung der Restic-Datenstrukturen, die innerhalb eines Repositorys gespeichert werden. Ein Restic-Repository, das Backups speichert, besteht nicht nur aus den verschlüsselten und authentifizierten Benutzerdaten. Ein Repository enthält weitere Daten, die von Restic für eine schnellere und effizientere Verarbeitung der verschlüsselten Benutzerdaten verwendet werden. Die von Restic hinzugefügten Datenstrukturen befinden sich alle in einem JSON-Format, bevor sie eventuell komprimiert, verschlüsselt und authentifiziert werden.
Die hier vorgestellten Datenstrukturen entsprechen nicht eins zu eins den gleichnamigen Datenstrukturen im Quellcode von Restic. Die Datenstrukturen in diesem Kapitel werden

Name	Typ	Beschreibung
id	*Hexstring*[64]	ID der Datenstruktur (Dateiname)
size	*Int*	Größe der Datenstruktur/Datei in Byte

Table 2.7.: Inhalt der Metadaten von Restic-Datenstrukturen

Name	Typ	Beschreibung
version	*Int*	Restic Version (In dieser Masterarbeit: 2)
id	*Hexstring*[64]	ID des Repositorys
chunker_polynomial	*Hexstring*[14]	Irreduzibles Polynom von Grad 53 für das Chunking-Verfahren

Table 2.8.: Inhalt der Config-Datenstruktur

nur mit den Inhalten modelliert, die auch am Ende an S_{Backup} geschickt werden und dort gespeichert werden. Die Reihenfolge der Inhalte der Datenstrukturen entspricht der Reihenfolge, in der die Daten beispielsweise in JSON-Formate konvertiert werden oder der Reihenfolge, in der die Daten in Dateien auf S_{Backup} vorliegen.

2.3.4.1. Metadaten

Metadaten sind in Restic zwar keine eigene Datenstruktur, aber es kommt öfters vor, dass S_{Restic} Metadaten bestimmter Restic-Datenstrukturen bei S_{Backup} anfordert. Der Inhalt der Metadaten, die S_{Backup} zurückschickt, ist in Tabelle 2.7 beschrieben.

2.3.4.2. Config

Ein Repository enthält genau eine Config-Datei. Die Config-Datenstruktur wird niemals von Restic komprimiert bevor sie verschlüsselt wird. Die Config eines Repositorys enthält Informationen über das Repository allgemein.
Der Inhalt der Config-Datenstruktur ist in Tabelle 2.8 beschrieben.

2.3.4.3. Keys

Jede Key-Datenstruktur ist einem Benutzerpasswort und damit einem Benutzer zuzuordnen. Restic speichert in der Key-Datenstruktur Informationen, um aus dem zugehörigen Benutzerpasswort einen Userkey mit der Schlüsselableitungsfunktion *scrypt* abzuleiten. Dieser Userkey besteht aus einem AES256 Schlüssel und einen Poly1305-AES128 Schlüssel. Außerdem ist der mit diesen Schlüsseln verschlüsselte und authentifizierte Masterkey in der Key-Datenstruktur gespeichert. Der entschlüsselte Masterkey einer Key-Datenstruktur ist ein JSON-String, der die drei oben erwähnten Schlüssel enthält. Die Key-Datenstruktur selbst wird nicht von Restic komprimiert und wird auch nicht verschlüsselt oder authentifiziert.

Name	Typ	Beschreibung
hostname	*String*	Ein vom Benutzer übergebener Hostname oder der Hostname des Betriebssystems auf dem Restic läuft.
username	*String*	Ein vom Benutzer übergebener Benutzername oder der Username des aktuellen Benutzers des Betriebssystems auf dem Restic läuft.
kdf	*String*	Name des Verfahrens, um Parameter n, r und p aus dem Benutzerpasswort zu extrahieren. Dieser Wert ist zum aktuellen Zeitpunkt immer "scrypt".
n	*Int*	Parameter für die Schlüsselableitungsfunktion *scrypt*.
r	*Int*	Parameter für die Schlüsselableitungsfunktion *scrypt*.
p	*Int*	Parameter für die Schlüsselableitungsfunktion *scrypt*.
time	*Timestamp*	Zeitpunkt, zu dem dieser Key hinzugefügt wurde.
data	*Bytestring*	Der verschlüsselte und authentifizierte Masterkey.
salt	*Bytestring*	Salt für die Schlüsselableitungsfunktion *scrypt*.

Table 2.9.: Inhalt der Key-Datenstruktur

Name	Typ	Beschreibung
k_{AES256}	$Bytestring[32]$	Der AES256 Schlüssel, mit dem alle Daten des Repositorys verschlüsselt sind.
$k_{Poly1305}$	$Bytestring[16]$	Der Poly1305 Schlüssel, der zusammen mit k_{AES128} zur Berechnung und Verifikation alle MACs dieses Repositorys benötigt wird.
k_{AES128}	$Bytestring[16]$	Der AES128 Schlüssel, der für das Poly1305-AES128 Verfahren benötigt wird.

Table 2.10.: Inhalt der Masterkey-Datenstruktur

Jedes Repository enthält mindestens eine Key-Datei. Der Inhalt dieser Key-Datei ist der unverschlüsselte JSON-String der Key-Datenstruktur. S_{Backup} kann die Key-Dateien aller Repositorys jederzeit einsehen.
Der Inhalt der Key-Datenstruktur ist in Tabelle 2.9 beschrieben.

2.3.4.4. Masterkey

Die Masterkey-Datenstruktur ist eine interne Datenstruktur von Restic, die nur auf S_{Restic} existiert. Der Masterkey setzt sich aus dem einen Schlüssel für AES256-CTR und den zwei weiteren Schlüsseln für Poly1305-AES128 zusammen. Der Inhalt der Masterkey-Datenstruktur ist in Tabelle 2.10 beschrieben

Name	Typ	Beschreibung
time	*Timestamp*	Zeitpunkt, zu dem das Lock erstellt wurde.
exclusive	*Bool*	Gibt an, ob es sich um ein exklusives Lock handelt. Bei einem exklusiven Lock, darf kein weiteres Lock aktiv sein.
hostname	*String*	Siehe Tabelle 2.9.
username	*String*	Siehe Tabelle 2.9.
pid	*Int*	Prozess-ID des Prozesses, der das Lock auf S_{Restic} erstellt.
uid	*Int*	ID, des Benutzers auf dem Betriebssystem (0 bei Windows).
gid	*Int*	ID, der Gruppe des Benutzers auf dem Betriebssystem (0 bei Windows).

Table 2.11.: Inhalt der Lock-Datenstruktur

2.3.4.5. Locks

Beim Zugriff auf ein Repository erstellt S_{Restic} eine Lock-Datenstruktur und schickt diese komprimiert, verschlüsselt und authentifiziert an S_{Backup}. Locks werden benutzt, um beim Zugriff auf ein Repository Race-Conditions zu vermeiden. Am Ende eines Zugriffs von Restic auf ein Repository von S_{Backup}, schickt S_{Restic} in der Regel eine Anfrage zum Löschen der erzeugten Lock-Dateien an S_{Backup}.
Der Inhalt der Lock-Datenstruktur ist in Tabelle 2.11 beschrieben.

2.3.4.6. Snapshots

Jedes Mal, wenn ein neues Backup erstellt wird, wird ebenfalls eine Snapshot-Datenstruktur angelegt, die komprimiert, verschlüsselt und authentifiziert an S_{Backup} geschickt wird. Diese Snapshot-Datenstruktur enthält allgemeine Informationen über das erstellte Backup. Der Snapshot enthält unter anderem alle Target-Pfade des Backups und die ID des Root-Tree-Blobs dieses Backups. Target-Pfade sind die Dateipfade und Verzeichnispfade, von denen ein Backup erstellt wurde (siehe Kapitel 2.3.1.7).
Der genaue Inhalt der Snapshot-Datenstruktur ist in Tabelle 2.12 beschrieben.

2.3.4.7. Data-Blob

Der Inhalt von Dateien (Benutzerdaten) von denen Restic ein Backup erstellt, wird mit einem Chunking-Verfahren (siehe Kapitel 2.3.2.6) in einzelne Chunks zerteilt. Diese Chunks sind zusammenhängende Bytestrings und in der Regel besitzen sie eine Länge zwischen 512 KiByte und 8 MiByte. Lediglich der letzte Chunk einer Datei kann kleiner als 512 KiByte sein, da Restic für sein Chunking-Verfahren kein Padding verwendet. Die Chunks werden im Kontext von Restics Datenstrukturen auch Data-Blobs genannt. Sie repräsentieren

Name	Typ	Beschreibung
time	*Timestamp*	Zeitpunkt, der von Restic festgelegt wird (z.B. Der Beginn des *backup* Befehls). Dieser Zeitpunkt muss nicht der Zeitpunkt sein, an dem der Snapshot erstellt wurde.
parent	*Hexstring*[64]	ID des Parent-Snapshots.
root_tree	*Hexstring*[64]	ID des Root-Tree-Blobs dieses Snapshots.
paths	$List_{String}$	Target-Pfade, mit denen dieses Backup erstellt wurde.
hostname	*String*	Siehe Tabelle 2.9.
username	*String*	Siehe Tabelle 2.9.
uid	*Int*	ID, des Benutzers auf dem Betriebssystem (0 bei Windows).
gid	*Int*	ID, der Gruppe des Benutzers auf dem Betriebssystem (0 bei Windows).
excludes	$List_{String}$	Reguläre Ausdrücke für Pfade, die für dieses Backup ignoriert werden sollen. Diese regulären Ausdrücke können als Parameter dem *backup* Befehl übergeben werden.
tags	$List_{String}$	Tags, die durch den Benutzer einem *backup* Aufruf zugewiesen werden können.
original	*Hexstring*[64]	ID, des ursprünglichen Snapshots, falls dieser Snapshot durch Restic-Befehle nachträglich verändert wurde.
program_version	*String*	Die Version von Restic zum Zeitpunkt der Erstellung des Snapshots.
summary	*JSON*	Statistiken über die Durchführung dieses *backup* Befehls, wie beispielsweise Anzahl Data-Blobs, Anzahl veränderter Dateien oder Anzahl verarbeiteter Dateien.

Table 2.12.: Inhalt der Snapshot-Datenstruktur

reine Daten und werden nicht in ein JSON-Format konvertiert, bevor sie verschlüsselt und authentifiziert werden. In einem Verzeichnissystem mit dem zugehörigen Verzeichnisbaum $T = (V, E)$ gibt es insgesamt $m \in \mathbb{N}$ Verzeichnisse und $n \in \mathbb{N}$ Dateien. Jeder Datei ist wie in Kapitel 2.3.1.9 gezeigt ein Knoten v_i für $m + 1 \leq i \leq m + n$ zuordbar. Die Data-Blobs, die sich aus einer Datei v_i ergeben werden mit $DB_i = DB(v_i) := \{DB_{i,1}, \ldots, DB_{i,s}\}$ bezeichnet, wobei s die Anzahl aller Chunks ist, in die die Datei v_i zerlegt wird. $DB_{i,j}$ bezeichnet den j-ten Chunk der Datei v_i. $DB_{i,1}||DB_{i,2}|| \ldots ||DB_{i,s}$ ist der Inhalt der Datei, die v_i repräsentiert.
Data-Blobs werden nicht einzeln an S_{Backup} gesendet, sondern immer in sogenannten Packs zusammengepackt (siehe Tabelle 2.15). Data-Blobs werden komprimiert, verschlüsselt, authentifiziert und mit anderen Data-Blobs zusammen in Pack-Datenstrukturen gepackt. Die Komprimierung von Data-Blobs kann per Option eines Restic-Befehls ausgestellt werden.

Name	Typ	Beschreibung
nodes	$List_{Node}$	Dateien oder direkte Unterverzeichnisse, im Verzeichnis dieses Tree-Blobs liegen.

Table 2.13.: Inhalt der Tree-Blob-Datenstruktur

2.3.4.8. Tree-Blob

Ein Tree-Blob gehört immer zu einem Verzeichnis auf dem S_{Restic}. Folglich lässt sich bei $m \in \mathbb{N}$ Verzeichnissen jedem Knoten $v_i \in V$ mit $1 \leq i \leq m$ des Verzeichnisbaums $T = (V, E)$ ein Tree-Blob zuordnen. Der Tree-Blob für Verzeichnis v_i wird bezeichnet mit TB_i oder $TB(v_i)$. TB_1 ist der Root-Tree-Blob zu dem Root-Verzeichnis des Betriebssystems. Eine Tree-Blob-Datenstruktur besteht aus mehreren Nodes (siehe Tabelle 2.14). Jeder dieser Node-Einträge repräsentiert ein direktes Unterverzeichnis oder eine Datei in dem Verzeichnis, dass der Tree-Blob repräsentiert. Die Nodes enthalten alle Informationen über das jeweilige Unterverzeichnis oder die Datei, sowie die ID des Tree-Blobs, der zu einem Unterverzeichnis gehört. Das heißt, ein Tree-Blob selbst enthält keine Informationen über sein eigenes Verzeichnis und enthält nur die Informationen über die Inhalte des Verzeichnisses. Die Informationen über das Verzeichnis des Tree-Blobs sind in dem Tree-Blob seines Elternknotens gespeichert. Damit gilt folgendes $\forall v \in \{v_1, \ldots, v_m\} : \forall (v, v') \in E :$ $TB(v)$ hat einen Node-Eintrag für v'.
Genau wie ein Data-Blob wird auch ein Tree-Blob niemals einzeln an S_{Backup} gesendet. Tree-Blobs werden immer komprimiert, verschlüsselt, authentifiziert und mit anderen Tree-Blobs zusammen in Pack-Datenstrukturen gepackt. Der Inhalt einer Tree-Blob-Datenstruktur ist in Tabelle 2.13 beschrieben.

2.3.4.9. Packs

Pack-Datenstrukturen enthalten mehrere Data-Blobs oder Tree-Blobs, sowie zusätzliche Informationen zu diesen Blobs. Ein Pack kann niemals Data-Blobs und Tree-Blobs gleichzeitig enthalten. Packs enthalten immer nur Blobs der gleichen Art.
Da Packs bereits verschlüsselte und authentifizierte Blobs enthalten, werden Pack-Datenstrukturen nicht in ein JSON-Format konvertiert und verschlüsselt. Pack-Datenstrukturen setzten sich aus einer Liste an Blobs und einem Pack-Header zusammen, der weitere Informationen über die enthaltenen Blobs enthält. Dazu ist der Pack-Header eine Liste aus Blob-Headern, die in Tabelle 2.16 dargestellt sind. Ein Pack ist eindeutig identifizierbar durch die enthaltenen Blobs und die Reihenfolge der Blobs in dem Pack. Daher wird für eine Pack-Datenstruktur, die beispielsweise die Data-Blobs $DB_{1,1}$, $DB_{4,2}$ und $DB_{3,5}$ in dieser Reihenfolge enthält die Notation $\langle DB_{1,1}, DB_{4,2}, DB_{3,5} \rangle$ verwendet.
Bei einer Pack-Datenstruktur wird lediglich der Pack-Header komprimiert, verschlüsselt und authentifiziert. Dieser verschlüsselte Pack-Header wird ans Ende, der hintereinander gehängten bereits verschlüsselten und authentifizierten Blobs, angehängt. Danach wird unverschlüsselt die Länge des Headers ans Ende angehängt. Die Länge des Headers wird in

Name	Typ	Beschreibung
name	*String*	Name des Verzeichnisses oder der Datei.
type	*String*	Gibt an, ob es sich um ein Verzeichnis oder eine bestimmte Datei handelt.
mode	*Int*	Gibt an, welcher Benutzer welche Rechte für dieses Verzeichnis oder diese Datei besitzt.
modTime	*Timestamp*	Zeitpunkt, zu dem dieses Verzeichnis oder Datei das letzte Mal modifiziert wurde.
accessTime	*Timestamp*	Zeitpunkt, zu dem das letzte Mal auf dieses Verzeichnis oder Datei zugegriffen wurde.
changeTime	*Timestamp*	Zeitpunkt, zu dem sich das letzte Mal etwas für dieses Verzeichnis oder Datei geändert hat. Dazu zählen auch Zugriffsberechtigungen.
uid	*Int*	ID, des Benutzers auf dem Betriebssystem, dem dieses Verzeichnis oder Datei gehört (0 bei Windows).
gid	*Int*	ID, der Gruppe des Benutzers auf dem Betriebssystem, dem dieses Verzeichnis oder Datei gehört (0 bei Windows).
user	*String*	Der Username des Benutzers des Betriebssystems, dem dieses Verzeichnis oder Datei gehört.
group	*String*	Der Name, der Gruppe des Benutzers des Betriebssystems, dem dieses Verzeichnis oder Datei gehört.
size	*Int*	Größe der Datei, falls dieser Node zu einer Datei gehört.
content	$List_{Hexstring[64]}$	IDs aller Data-Blobs, die zu dieser Datei gehören, falls dieser Node zu einer Datei gehört. Die Reihenfolge der IDs entspricht der korrekten Anordnung der Data-Blobs, aus denen sich der Dateiinhalt zusammensetzt.
subtree	$Hexstring[64]$	ID des Tree-Blobs, der zu dem Verzeichnis gehört, falls dieser Node zu einem Verzeichnis gehört.

Table 2.14.: Inhalt einer Node-Datenstruktur

genau vier Bytes gespeichert.
Der Inhalt einer Pack-Datenstruktur ist in Tabelle 2.15 beschrieben.

2.3.4.10. Indizes

Index-Datenstrukturen enthalten die Informationen, in welcher Pack-Datenstruktur und an welcher Stelle innerhalb dieses Packs ein Blob liegt. Dazu werden alle Blobs der Index-Datenstruktur nach ihren Packs gruppiert. Im Laufe der Backuperstellung werden Index-Datenstrukturen erstellt, die nach einer gewissen Zeit oder bei einer gewissen Größe komprimiert, verschlüsselt und authentifiziert werden, bevor sie an S_{Backup} geschickt wer-

Name	Typ	Beschreibung
blobs	$List_{encrypt(comp(Blob))}$	Verschlüsselte, authentifizierte und komprimierte Data-Blobs oder Tree-Blobs. Für Tree-Blobs ist *Blob* der JSON-String dieses Tree-Blobs.
header	$encrypt(List_{Blob-Header})$	Header zu jedem Blob, das in diesem Pack gespeichert ist. Die Header besitzen die selbe Reihenfolge, wie die gespeicherten Blobs.
header_length	*Int*	Die Länge des verschlüsselten und authentifizierten Headers dieses Packs.

Table 2.15.: Inhalt der Pack-Datenstruktur

Name	Typ	Beschreibung
type	*Int*	Gibt an, ob es sich bei diesem Blob, um ein Data-Blob oder Tree-Blob handelt, und ob dieser Blob komprimiert wurde oder nicht.
length	*Int*	Länge des verschlüsselten, authentifizierten und komprimierten Blobs in Bytes (mit IV und mit MAC).
uncompressed_length	*Int*	Länge des unverschlüsselten und unkomprimierten Blobs in Bytes.
id	$Hexstring[64]$	ID, des Blobs, zu dem dieser Header gehört.

Table 2.16.: Inhalt der Blob-Header-Datenstruktur

den. Bei einem *backup* Befehl werden immer die Informationen alle Blobs eines Packs in demselben Index gespeichert.
Der Inhalt einer Index-Datenstruktur ist in Tabelle 2.17 beschrieben. Damit lässt sich durch Tabelle 2.19 bestimmen, dass ein verschlüsselter und authentifizierter Blob in seinem Pack an der Byte-Position *offset + 1* bis *offset + length* gespeichert wird, wobei die erste Position eines Packs 1 ist.

2.4. Angreifermodell und Angriffsszenario

In diesem Kapitel wird der in dieser Masterarbeit betrachte Angreifer beschrieben, seine Motivation erklärt und das reale Szenario diskutiert, in dem so ein Angreifer auftreten kann. Restic wird, wie in Kapitel 2.3.1.5 in das Resticsystem S_{Restic} und das Backupsystem S_{Backup} unterteilt. Das Backupsystem ist in der Realität selten das gleiche System, wie das Resticsystem, damit im Falle einer Störung oder anderweitigem Datenverlust auf dem Resticsystem

Name	Typ	Beschreibung
packs	$List_{Packs}$	Packs, die Blobs enthalten, für die dieser Index Informationen speichert.

Table 2.17.: Inhalt der Index-Datenstruktur

Name	Typ	Beschreibung
id	$Hexstring[64]$	ID des Packs.
blobs	$List_{Blobs}$	Informationen über Blobs, die in diesem Pack enthalten sind.

Table 2.18.: Inhalt der Packs-Datenstrukturen einer Index-Datenstruktur

Name	Typ	Beschreibung
id	$Hexstring[64]$	ID dieses Blobs.
type	Int	Gibt an, ob es sich bei diesem Blob, um ein Data-Blob oder Tree-Blob handelt.
offset	Int	Ab welchem Byte des zugehörigen Packs startet das Chiffrat des Blobs.
length	Int	Länge des verschlüsselten, authentifizierten und komprimierten Blobs in Bytes (mit IV und mit MAC).
uncompressed_length	Int	Länge des unverschlüsselten und unkomprimierten Blobs in Bytes.

Table 2.19.: Inhalt der Blobs-Datenstruktur einer Packs-Datenstruktur

das Repository auf dem Backupsystem unabhängig und unversehrt bleibt. Nur so kann garantiert werden, dass die Wiederherstellung von früheren Backups funktionsfähig bleibt, egal was auf S_{Restic} passiert. Diese Aussage schließt natürlich Szenarien aus, in denen das Resticsystem kompromittiert ist und damit das Repository auf S_{Backup} von dem kompromittierten Resticsystem verändert wird. Ein kompromittiertes Resticsystem bräuchte weiterhin ein Benutzerpasswort, um auf ein Repository zugreifen zu können und damit bietet Restic auch für diesen Fall einen gewissen Schutz. Allerdings besitzt ein Angreifer, der S_{Restic} kompromittiert hat, Zugriff auf die Daten des Resticsystems und hat damit auch Zugriff auf alle Benutzerdaten, von denen in Zukunft eventuell ein Backup erstellt werden würde. Ein Angreifer, der Zugriff auf S_{Restic} besitzt, findet aber ohne Benutzerpasswort nicht heraus, welche Daten in einem Repository auf S_{Backup} gespeichert sind.
Ein interessanteres Angriffsszenario ist das Szenario, in dem der Angreifer nur Zugriff auf S_{Backup} besitzt und das Repository beobachten und eventuell manipulieren kann. Dieses Szenario ist außerdem für den Betreiber des Resticsystems interessant, da dieser Betreiber eventuell keinen Einfluss auf die Sicherheit des Backupsystem hat, wenn das Backupsystem beispielsweise einen Cloud-Dienst oder externen Server darstellt. Folglich sollte es den Betreiber eines Resticsystems interessieren, welchen Schaden der Angreifer anrichten kann, sobald es dem Angreifer gelingt Zugriff auf S_{Backup} zu erlangen. Aus diesem Grund wird in dieser Masterarbeit vornehmlich ein Angreifer betrachtet, der Zugriff auf S_{Backup} hat.
Ein Angreifer, der Zugriff auf S_{Backup} hat und ein Repository beobachtet, sieht Datenstrukturen durch S_{Restic} verändert, hinzugefügt oder angefordert werden. Außerdem könnte ein manipulierender Angreifer das Repository, auf das S_{Restic} zugreift, verändern. Jede Veränderung des Repositorys oder Anfrage von Daten aus dem Repository, ist die Konsequenz einer Nachricht, die S_{Restic} an S_{Backup} schickt. Außerdem werden angefragte Daten ebenfalls per Nachricht von S_{Backup} an S_{Restic} geschickt. Damit ist ein Angreifer, der das Reposi-

tory auf S_{Backup} beobachtet, genauso mächtig, wie ein Angreifer, der alle ausgetauschten Nachrichten zwischen S_{Restic} und S_{Backup} sieht. Daraus folgt, dass ein Angreifer, der alle ausgetauschten Nachrichten zwischen S_{Restic} und S_{Backup} gesehen hat, genauso mächtig ist, wie ein Angreifer, der das Repository auf S_{Backup} beobachten kann. Voraussetzung für diese Aussage ist natürlich, dass die Nachrichten abseits von Restics Verschlüsselung für den Nachrichtenaustausch nicht erneut verschlüsselt wurden. Da Restic alle Daten, bis auf Key-Datenstrukturen verschlüsselt bietet sogar Restic selbst eine Implementierung [7] des S_{Backup} an, die HTTP verwendet. Diese Implementierung bietet zwar eine Möglichkeit an, den Nachrichtenaustausch mit TLS verschlüsseln zu lassen, dennoch sollte darauf hingewiesen werden, dass sämtliche Erkenntnisse dieser Masterarbeit auch für einen Angreifer gelten, der lediglich den Nachrichtenaustausch zwischen S_{Restic} und S_{Backup} beobachten kann.
Es gibt einen entscheidenden Unterschied zwischen einem Angreifer, der den Nachrichtenaustausch mitliest und einem Angreifer, der das Repository direkt beobachtet. Existierte das Repository bereits bevor die jeweiligen Angreifer ihre Beobachtung starten konnten, besitzt der Angreifer, der das Repository beobachtet einen Vorteil, da er das gesamte Repository sieht, ohne dass daraus Daten durch S_{Restic} angefordert werden müssen. Dies ist eine kleine Einschränkung für einen Nachrichten beobachtenden Angreifer, die jedoch erwähnt werden muss.

Betrachtetes Angriffsszenario:
In dieser Masterarbeit wird das Angriffsszenario betrachtet, in der ein Angreifer $\mathcal{A}$ Zugriff auf ein Repository von Restic besitzt. $\mathcal{A}$ kann alle Veränderungen des Repositorys beobachten und je nach Szenario auch selber das Repository verändern. Der Angreifer $\mathcal{A}$ besitzt im Allgemeinen jedoch keine Kenntnis über den Masterkey *mk* des Repositorys.

Warum werden nicht mehrere Repositorys betrachtet?:
In der Realität kann es vorkommen, dass sich mehrere verschiedene Repositorys auf S_{Backup} befinden. Selbst wenn alle diese Repositorys von demselben S_{Restic} stammen, besitzt jedes Repository einen eigenen Masterkey, eigene Parameter für das Chunkingverfahren und eine eigene interne Struktur. Damit sind Repository von Restic unabhängig voneinander und jedes Repository besitzt seine eigene Vertrauensgrenze. Überschreitet ein Angreifer eine Vertrauensgrenze, gilt das nur für das entsprechende Repository und nicht für alle anderen. Aus diesen Gründen wird in dieser Masterarbeit nur ein Repository pro Angreiffszenario betrachtet.

3. Restic als kryptografisches Protokoll

Dieses Kapitel betrachtet Restic als kryptografisches Protokoll. Die Aufgabe von Restic ist es Funktion zum Erstellen und Wiederherstellen eines Backups von Dateien und Dateiverzeichnissen bereitzustellen. Dabei soll Restic die Vertraulichkeit und Integrität der Daten wahren. Die drei Befehle, die als kryptografisches Protokoll modelliert werden, sind der *backup*, *restore* und *prune* Befehl von Restic. Der *backup* Befehle erstellt ein Backup für ein bereits initialisiertes Restic-Repository. Der *restore* Befehl stellt ein bereits erstelltes Backup wieder her. Der *prune* Befehl löscht nicht mehr verwendete Blobs aus dem Repository und fasst die übrigen Blobs in neuen Packs zusammen.
Für jeden Restic-Befehl werden zwei Modellierungen vorgestellt. Eine detaillierte Modellierung mit Pseudocode, die sich an dem Quellcode von Restic orientiert und damit Restics interne Abläufe detailliert modelliert. Die detaillierte Modellierung wird auch reale Modellierung genannt, da ihre Abläufe sehr detailliert an der realen Implementierung von Restic modelliert sind. Damit die vorgestellten Sicherheitsspiele verständlich und nachvollziehbar sind, wird außerdem eine abstrakte Modellierung der Restic-Befehle als kryptografische Protokolle vorgestellt. In den vorgestellten Sicherheitsspielen kann der Angreifer nur das Repository und das Backupsystem beobachten und sieht damit nicht den genauen Ablauf von Restic auf S_{Restic}. Der Angreifer sieht nur die ausgetauschten Datenstrukturen und die Reihenfolge und Zeitpunkte zu denen diese Austausche stattfinden. Daher wird in der abstrakten Modellierung der Restic-Befehle auch nur dieser Datenstrukturaustausch modelliert. Bei der abstrakten Modellierung liegt der Fokus auf Formalität, einfacher Verständlichkeit, den zeitlichen Abläufen, sowie Größe und Zusammensetzung der ausgetauschten Nachrichten. Trotz der Einfachheit der abstrakten Modellierung werden keine sicherheitsrelevanten Eigenschaften und Abläufe im Vergleich zur detaillierten Modellierung vernachlässigt.

3.1. Datenstrukturen

Dieses Kapitel beschreibt den Aufbau von Restics internen Datenstrukturen. Diese Datenstrukturen werden nur im Pseudocode verwendet und niemals an S_{Backup} gesendet.

3.1.1. Repository

Zu jedem modellierten Restic-Befehl gibt es immer genau eine Repository-Datenstruktur. Die Repository-Datenstruktur beinhaltet alle Informationen über das betrachtete Repository,

Name	Typ	Beschreibung
masterkey	$Masterkey$	Speichert die von dem Repository verwendeten AES256 und Poly1305 Schlüssel k_{AES256} und (k_{Poly}, k_{AES128}). Der Masterkey dieses Repositorys wird für alle Aufrufe, der Funktionen 16 und 17 verwendet, die bei der Ausführung eines Restic-Befehls bezüglich diesem Repository ausgeführt werden.
config	$Config$	Speichert die Config-Datenstruktur dieses Repositorys.
masterindex	$Index$	Eine große Index-Datenstruktur, die alle Einträge aus allen Index-Datenstrukturen dieses Repositorys enthält.
snapshot	$Snapshot$	Speichert den ausgewählten Parent-Snapshot für einen *backup* Befehl oder den ausgewählten Snapshot zum Wiederherstellen eines Backups für einen *restore* Befehl.
all_indexes	$Liste_{Index}$	Speichert eine Liste von neu erstellten Index-Datenstrukturen, zu denen Einträge für neu erstellte Packs hinzugefügt werden, sobald diese Packs voll sind.
used_blobs	$List_{Hexstring[64]}$	Diese Liste wird für den Prune-Befehl verwendet und enthält die Restic-IDs aller Tree-Blobs und Data-Blobs des Repositorys, die in mindestens einem Backup eines Snapshots des Repositorys verwendet werden.

Table 3.1.: Inhalt der Repository-Datenstruktur

die wichtig für den jeweiligen Restic-Befehl sind. Eine Repository-Datenstruktur existiert nur intern für Restic und wird niemals an S_{Backup} gesendet.
Die Repository-Datenstruktur wird mit `repo` im Pseudocode bezeichnet.
Der Inhalt einer Repository-Datenstruktur ist in Tabelle 3.1 beschrieben.

3.1.2. Prune-Plan

Der *prune* Befehl verwendet die Datenstruktur Prune-Plan. Die Prune-Plan-Datenstruktur enthält Informationen darüber, welche Packs gelöscht werden sollen und welche für das Repacking ausgewählt wurden. Die Prune-Plan-Datenstruktur existiert nur intern für Restic und wird niemals an S_{Backup} gesendet.
Der Inhalt einer Prune-Plan-Datenstruktur ist in Tabelle 3.2 beschrieben.

Name	Typ	Beschreibung
removePacksFirst	$List_{Hexstring[64]}$	Liste aller Pack-IDs von Packs aus dem Repository, die nicht im Masterindex vorkommen. Diese Packs werden direkt zu Beginn der Ausführung des Prune-Plans aus dem Repository entfernt.
removePacks	$List_{Hexstring[64]}$	Liste aller Pack-IDs von Packs, denen kein Used-Blob zugeordnet wurde. Dies Packs werden ebenfalls im Laufe der Ausführung aus dem Repository entfernt.
repackSmallCandidates	$List_{Hexstring[64]}$	Liste aller Pack-IDs von Packs, die nur aus Used-Blobs bestehen und eine sehr kleine Größe haben. Bei genügend Einträgen in dieser Liste werden alle diese Pack-IDs ebenfalls zur Liste *repackPacks* hinzugefügt.
repackPacks	$List_{Hexstring[64]}$	Liste aller Packt-IDs von Packs, denen mindestens ein Used-Blob zugeordnet wurde. Die Used-Blobs dieser Packs werden im Laufe der Ausführung in neue Packs gepackt und die alten Packs werden aus dem Repository entfernt.
ignorePacks	$List_{Hexstring[64]}$	Liste aller Pack-IDs von Packs, die im Masterindex vorkommen, aber sich nicht im Repository befinden. Zu dieser Liste werden im Laufe der Ausführung alle Pack-IDs der anderen Listen des Prune-Plans gesammelt, sobald die anderen Listen abgearbeitet wurden.

Table 3.2.: Inhalt der Prune-Plan-Datenstruktur

3.2. Backup Befehl

Der *backup* Befehl von Restic kann für ein gegebenes Restic-Repository aufgerufen werden. Restic erstellt daraufhin basierend auf den übergebenen Target-Pfaden ein neues Backup für das Verzeichnissystem des S_{Restic} und speichert das Backup in diesem Repository.
Zuerst werden die möglichen Optionen vorgestellt, mit denen ein *backup* Befehl ausgeführt werden kann.
Daraufhin stellt dieses Kapitel zwei Modellierungen des *backup* Befehls als kryptografisches Protokoll vor. Zunächst wird die reale Modellierung vorgestellt, bei der der *backup* Befehl in mehrere detailliert beschriebene Blöcke unterteilt wird. Diese Blöcke sind unter anderem auch im Anhang beschrieben, falls sie in anderen Restic-Befehlen wiederverwendet werden (siehe Kapitel A.1).
Danach wird die abstrakte Modellierung des *backup* Befehls als kryptografisches Protokoll vorgestellt. Diese Modellierung wird für die später vorgestellten Sicherheitsspiele verwendet. Bei dieser Modellierung wird der Fokus auf den Nachrichtenaustausch zwischen S_{Restic} und S_{Backup} gelegt.

Name	Beschreibung
REPOSITORY_PATH	Ort, an dem das Repository liegt. Dies wird benötigt, um eine Verbindung zum S_{Backup} aufzubauen.
KEY_HINT	ID-Präfix einer Key-Datei des Repositorys. Zum gezielten Anfordern einer Key-Datei.
CONNECTIONS	Restic verwendet diesen Wert, um die Anzahl paralleler Prozesse zu bestimmen, die für einige Funktionen erstellt werden.
FILES_FROM	Liste von Dateien, die reguläre Ausdrücke enthalten. Alle Pfade des Resticsystems, die einem dieser regulären Ausdrücke passen, werden zu den Target-Pfaden hinzugefügt.
FILES_FROM_VERBATIM	Liste von Dateien, die Pfade auf S_{Restic} enthalten. Diese Pfade werden zu den Target-Pfaden hinzugefügt.
FILES_FROM_RAW	Das Gleiche wie *FILES_FROM_VERBATIM* mit dem Unterschied, dass die Pfade durch einen speziellen Character voneinander getrennt werden und nicht durch Zeilenumbrüche.
PARENT	ID-Präfix des Parent-Snapshots, der für diesen *backup* Aufruf verwendet werden soll.
PACK_SIZE	Restic verwendet diesen Wert, um die angestrebte Größe eines Packs festzulegen. Ein Pack gilt erst als voll und wird gespeichert, wenn seine Größe größer ist als dieser Wert.

Table 3.3.: Optionen des *backup* Befehls

3.2.1. Prämisse

Es wird davon ausgegangen, dass durch die Ausführung von Restics *init* Befehls bereits ein Repository auf S_{Backup} initialisiert wurde. Ferner bedeutet das, dass mindestens eine Key-Datei und eine verschlüsselte und authentifizierte Config-Datei in diesem Repository auf dem Backupsystem existieren.

3.2.2. Optionen des *backup* Befehls

Bei dem Aufruf des *backup* Befehls können verschiedene Optionen gesetzt oder konfiguriert werden, die den Ablauf des *backup* Befehls modifizieren. Diese Optionen sind in Tabelle 3.3 aufgelistet.

3.2.3. Restic-Funktionen für den *backup* Befehl

In diesem Kapitel werden Restic-Funktionen (Blöcke) erklärt, die nur von dem *backup* Befehl verwendet werden. Diese Funktionen werden in einem späteren Kapitel verwendet, um den Ablauf des *backup* Befehls zu modellieren.

3.2.3.1. Target-Pfade für dieses Backup bestimmen

Für einen *backup* Befehl muss Restic die Target-Pfade auf dem Resticsystem bestimmen, von denen ein Backup erstellt werden soll. Dazu erstellt Restic eine deterministische Target-Liste aus existierenden Pfaden auf S_{Restic}. Der Benutzer kann mit dem *backup* Befehl in dem Optionen *files-from*, *files-from-verbatim* und *files-from-raw* Pfade zu Textdateien übergeben. Diese Textdateien enthalten Target-Pfade für diesen *backup* Aufruf. Die Optionen werden immer in der gleichen Reihenfolge abgegangen und deren Pfade zur Target-Liste hinzugefügt.

1. **files-from:** Restic führt in der übergebenen Reihenfolge auf jeder dieser Textdateien folgendes aus. Jede Zeile wird nach und nach eingelesen. Die Leerzeichen am Anfang und Ende der Zeile werden entfernt und Zeilen, die mit einem '#' beginnen, werden verworfen. Die getrimmte eingelesene Zeile wird als regulärer Ausdruck interpretiert und Restic bestimmt die Pfade aller Dateien auf dem System, deren Pfad den regulären Ausdruck erfüllt. Die Pfade zu allen passenden Dateien werden am Ende der Target-Liste hinzugefügt.
2. **files-from-verbatim:** Restic führt in der übergebenen Reihenfolge auf jeder dieser Textdateien folgendes aus. Jede Zeile wird nach und nach eingelesen und am Ende der Target-Liste hinzugefügt.
3. **files-from-raw:** Restic führt in der übergebenen Reihenfolge auf jeder dieser Textdateien folgendes aus. Restic liest so viele Bytes aus der Datei, bis ein Null-Byte gelesen wird. Diese gelesenen Bytes werden im Folgenden als ein Dateipfad interpretiert. Endet der Dateiinhalt mit einem Null-Byte, werden alle gelesen Dateipfade in der gelesenen Reihenfolge am Ende der Target-Liste hinzugefügt.

Als Letztes werden ebenfalls alle, beim *backup* Aufruf, übergebenen Pfade in der übergebenen Reihenfolge am Ende der Target-Liste hinzugefügt. Restic überprüft am Ende, ob jeder Pfad der Target-Liste auf dem Resticsystem existiert und entfernt alle nicht existierenden Pfade. Das Ergebnis der Funktion 1 ist eine deterministische Liste von Target-Pfaden.

3.2.3.2. Erstellen des Verzeichnisbaums und speichern des Backups

Beginnend mit dem Wurzelknoten des Verzeichnisbaum $T_{backup} := (V_{backup}, E_{backup})$ für das virtuelle Verzeichnissystem dieses *backup* Aufrufs traversiert Restic T_{backup} in Form einer Tiefensuche. Parallel zum Traversierung von T_{backup} wird anhand des Parent-Snapshots ein zweiter Verzeichnisbaum T_{parent} aufgebaut. Konnte kein Parent-Snapshot bestimmt werden, ist T_{parent} für die Ausführung des *backup* Befehls ein leerer Baum. Das Traversieren des

Funktion 1 Collects target paths to backup

```
Require: FILES_FROM, FILES_FROM_VERBATIM, FILES_FROM_RAW
  function getTargetPaths(commandTargetPaths)
    List_String targets := {}
    for all String filePath in FILES_FROM do
      for all String line in getFile(filePath) do
        regex := trimAndFilter(line)
        if regex is not empty then
          for all targetPath in getMatchingPaths(regex) do
            targets := targets||targetPath
          end for
        end if
      end for
    end for
    for all String filePath in FILES_FROM_VERBATIM do
      for all String line in getFile(filePath) do
        targets := targets||line
      end for
    end for
    for all String filePath in FILES_FROM_RAW do
      file := getFile(filePath)
      loop
        String targetPath := getNextPath(file)
        if targetPath == "" and isEndReached(file) then
          break
        end if
        targets := targets||targetPath
      end loop
    end for
    for all String targetPath in commandTargetPaths do
      targets := targets||targetPath
    end for
    removeNonExistingPaths(targets)
    return targets
  end function
```

Verzeichnisbaums und das jeweilige Erstellen der Tree-Blobs und Data-Blobs ist in Funktion 2 modelliert. Dazu liefert die Funktion `getSelectedSubPaths(path)` für den Knoten $v \in V_{backup}$ mit dem Pfad `path` eine nichtdeterministische Liste aller Pfade, die zu Knoten $v' \in V_{backup}$ gehören mit $(v, v') \in E_{backup}$.
Bevor Restic jedoch beginnt T_{backup} zu traversieren, werden von Restic parallele Prozesse gestartet, die dafür verantwortlich sind Tree-Blobs oder Data-Blobs zu speichern. Diese parallelen Prozesse werden Saver-Prozesse genannt. Die Anzahl der gestarteten Saver-Prozesse wird anhand der vorhandenen logischen Kerne der CPU bestimmt. Immer wenn

Unterverzeichnis	Datei
Ein neuer Tree-Blob wird für das Unterverzeichnis erstellt und die Funktion 2 wird rekursiv für den neuen Tree-Blob und dessen Verzeichnispfad aufgerufen.	Restic überprüft mit `isDifferentFromParent(subpath)`, ob sich die Modifikationszeit oder die Dateigröße im Vergleich zum Zeitpunkt des Parent-Snapshots verändert haben. Konnte kein Knoten in T_{parent} mit dem Pfad *subPath* gefunden werden, gibt `isDifferentFromParent(subpath)` immer *true* zurück. Wenn Änderungen detektiert werden konnten, teilt Restic den Inhalt der Datei mit seinem Chunking-Verfahren in mehrere Chunks auf. Jeder dieser Chunks wird auf einen der parallelen Saver-Prozesse verteilt mit `dispatch(chunk, false)`.

Daten an einen Saver-Prozess mit `dispatch(...)` übergeben werden startet der Saver-Prozess mit der Funktion 31.
Danach beginnt Restic mit dem Root-Verzeichnis-Pfad den Verzeichnisbaum T_{backup} rekursiv zu traversieren mit Funktion 2. Bei jedem `createBackup(treeBlob, path)` Aufruf wird mit der Funktion `loadParentTreeBlob(path)` überprüft, ob es in T_{parent} einen Knoten für den Pfad `path` gibt. Wenn T_{parent} einen passenden Knoten besitzt, wird der zugehörige Tree-Blob mit der Funktion 28 geladen und der Verzeichnisbaum T_{parent} um die Node-Einträge des geladenen Tree-Blobs erweitert. Beim Traversieren betrachtet Restic jedes Unterverzeichnis und jede Datei, deren Pfade in der Liste von `getSelectedSubPaths(path)` enthalten sind. Je nachdem, ob es sich um ein Unterverzeichnis oder eine Datei handelt, passiert etwas anderes.

Nach jedem Unterverzeichnis und jeder Datei wird dem aktuellen Tree-Blob `treeBlob` ein Node-Eintrag mit `addNode(treeBlob, subPath)` hinzugefügt. Dieser neue Node enthält die ID des Tree-Blobs des neuen Unterverzeichnisses mit Pfad `subPath` oder die IDs der Data-Blobs, aus denen sich die Datei mit Pfad `subPath` zusammensetzt.
Nachdem der komplette Inhalt des Verzeichnisses des aktuellen Tree-Blobs `treeBlob` rekursiv von Restic bearbeitet wurde, wird der aktuelle Tree-Blob auf einen der parallelen Saver-Prozesse verteilt mit `dispatch(treeBlob, false)`.

3.2.3.3. Erstellen eines Snapshots

Zum Erstellen eines neuen Snapshots wird ein Verzeichnisbaum T_{backup} für das zum *backup* Befehl gehörige virtuelle Verzeichnissystem benötigt. Ein Snapshot verweist immer auf den Tree-Blob (Root-Tree-Blob) des Wurzelknotens von T_{backup}. Damit kann T_{backup} zu einem späteren Zeitpunkt aus dem Snapshot rekonstruiert werden.
Restic erstellt eine neue Snapshot-Datenstruktur, die die ID des Root-Tree-Blobs enthält. Die Snapshot-Datenstruktur wird komprimiert, verschlüsselt und authentifiziert. S_{Restic} schickt den verschlüsselten und authentifizierten Snapshot an S_{Backup}, welches dafür verantwortlich ist die Daten in eine Snapshot-Datei zu schreiben und im Repository zu speichern.

Funktion 2 Create and store backup

```
procedure createBackup(Tree − Blob treeBlob, String path)
    loadParentTreeBlob(path)
    for all String subPath in getSelectedSubPaths(path) do
        if isDir(subPath) then
            Tree − Blob newTreeBlob := newTreeBlob(subPath)
            createBackup(newTreeBlob, subPath)
        else if isFile(subPath) then
            if isDifferentFromParent(subPath) then
                for all Bytestring chunk in devideFileInChunks(node.path) do
                    dispatch(chunk, false)
                end for
            end if
        end if
        addNode(treeBlob, subPath)
    end for
    dispatch(treeBlob, false)
end procedure
```

Funktion 3 Creates a snapshot

```
procedure createSnapshot(rootTreeBlobId)
    Snapshot S̃S_new := newSnapshot(rootTreeBlobId)
    send(encrypt(compress(S̃S_new)))                    ▷ S_Restic ⟶ S_Backup
end procedure
```

3.2.4. Ablauf der realen Modellierung des *backup* Befehls

Dieses Kapitel beschreibt den detaillierten Ablauf nach dem Aufruf des Restic *backup* Befehls. Dieser Ablauf ist auch in Diagramm 3.1 dargestellt.
Als erstes bestimmt Restic die Target-Pfade auf S_{Restic} von denen der Benutzer ein Backup erstellen möchte. Die Bestimmung der Target-Pfade ist deterministisch. Restic konstruiert nun einen Verzeichnisbaum T_{backup}, dessen Blätter die Target-Pfade darstellen und dessen Wurzelknoten das Root-Verzeichnis des realen Verzeichnissystems ist. Zu T_{backup} werden dann neue Knoten und Kanten hinzugefügt, sodass alle Verzeichnisse und Dateien, für die ein Target-Pfad ein Präfix für ihren eigenen Pfad darstellt, in T_{backup} enthalten sind. Damit entspricht T_{backup} dem Verzeichnisbaum für das virtuelle Verzeichnissystem für dieses Backup.
Danach sucht Restic das vom Benutzer angegebene Repository auf S_{Backup} und baut eine interne Repository-Datenstruktur auf.
Bevor Restic weiter mit dem Repository auf S_{Backup} interagiert, wird eine nicht exklusive Lock-Datei dem Repository hinzugefügt. Dadurch wird verhindert, dass mögliche andere Restic-Distributionen exklusiv auf das Repository zugreifen können.
Restic holt sich entweder einen durch die *parent* Option spezifizierten Snapshot oder den

Latest-Snapshot, dessen Target-Pfade die Targets dieses *backup* Aufrufs enthalten. Dieser Snapshot wird Parent-Snapshot genannt. Es kann auch dazu kommen, dass kein Parent-Snapshot gefunden wurde und damit enthält der Verzeichnisbaum T_{parent} für den Parent-Snapshot keine Knoten.
Nun lässt sich Restic alle Indices des Repositorys von S_{Backup} schicken und speichert diese in seiner internen Repository-Datenstruktur.
Jetzt beginnt der Kern des *backup* Befehls. Bevor es mit der Ausführung des Backups weiter geht, startet Restic parallele Pack-Uploader-Prozesse und parallele Saver-Prozesse. Es wird rekursiv mit Funktion 2, beginnend bei dem Wurzelknoten, der Verzeichnisbaum T_{backup} in Form einer Tiefensuche nichtdeterministisch traversiert. Gleichzeitig wird auch immer ein Verzeichnisbaum T_{parent} für den damaligen Zustand des Parent-Snapshots aufgebaut, sofern zu Beginn ein Parent-Snapshot gefunden wurde. Dazu wird bei jedem Aufruf von `createBackup(...)` überprüft, ob es einen Knoten in T_{parent} gibt, der den gleichen Pfad wie das aktuelle Verzeichnis hat und daraufhin wird der zugehörige Tree-Blob geladen. Konnte ein Knoten gefunden werden, wird T_{parent} um die Node-Einträge des geladenen Tree-Blobs erweitert. T_{parent} stellt den Verzeichnisbaum des virtuellen Verzeichnissystems des Backups des Parent-Snapshots dar und wird für die Änderungsdetektion bei Dateien verwendet.
Trifft der rekursive Algorithmus auf eine Datei, wird mit Hilfe des Verzeichnisbaums des Parent-Snapshots überprüft, ob die Datei seit dem Parent-Snapshot verändert wurde. Konnte eine Änderung detektiert werden, wird mit dem Chunking-Verfahren von Restic, der Inhalt der Datei in mehrere Data-Blobs zerlegt. Jeder dieser Data-Blobs wird an einen Saver-Prozess übergeben. Die Informationen der Datei werden beim Tree-Blob des Verzeichnisses, in dem sich die Datei befindet hinzugefügt.
Trifft der rekursive Algorithmus auf ein Verzeichnis, wird der gesamte Inhalt des Verzeichnisses rekursiv abgearbeitet, indem die `createBackup(...)` für dieses Verzeichnis aufgerufen wird. Wurden alle Dateien und Unterverzeichnisse dieses Verzeichnisses erfasst, wird der Tree-Blob des betrachteten Verzeichnisses einem Saver-Prozess übergeben. Die Informationen über das betrachtete Verzeichnis werden dem Tree-Blob dessen Oberverzeichnisses als Node-Eintrag hinzugefügt.
Ein Saver-Prozess packt Blobs verschlüsselt und authentifiziert in einem zufällg ausgewählten Pack zusammen und übergibt dieses Pack an einen Pack-Uploader-Prozess, sobald das Pack eine gewisse Größe überschreitet.
Ein Pack-Uploader-Prozess schickt mit Funktion 27 das Pack an S_{Backup} und fügt dessen Informationen zu einem Index hinzu. Ist ein Index zu alt oder enthält er zu viele Packinformationen wird er verschlüsselt, authentifiziert und ebenfalls an S_{Backup} geschickt.
Nachdem der komplette Verzeichnisbaum aufgebaut und in Packs gespeichert wurde, besitzt Restic eventuell noch nicht auf S_{Backup} gespeicherte Packs oder Indices. Die übrigen Packs und Indices, die noch nicht an S_{Backup} geschickt wurden, werden verschlüsselt, authentifiziert und an S_{Backup} geschickt.
Am Ende des *backup* Aufrufs erstellt Restic einen Snapshot, der auf den Root-Tree-Blob von T_{backup} verweist. Dieser Snapshot wird verschlüsselt und authentifiziert an S_{Backup} gesendet.

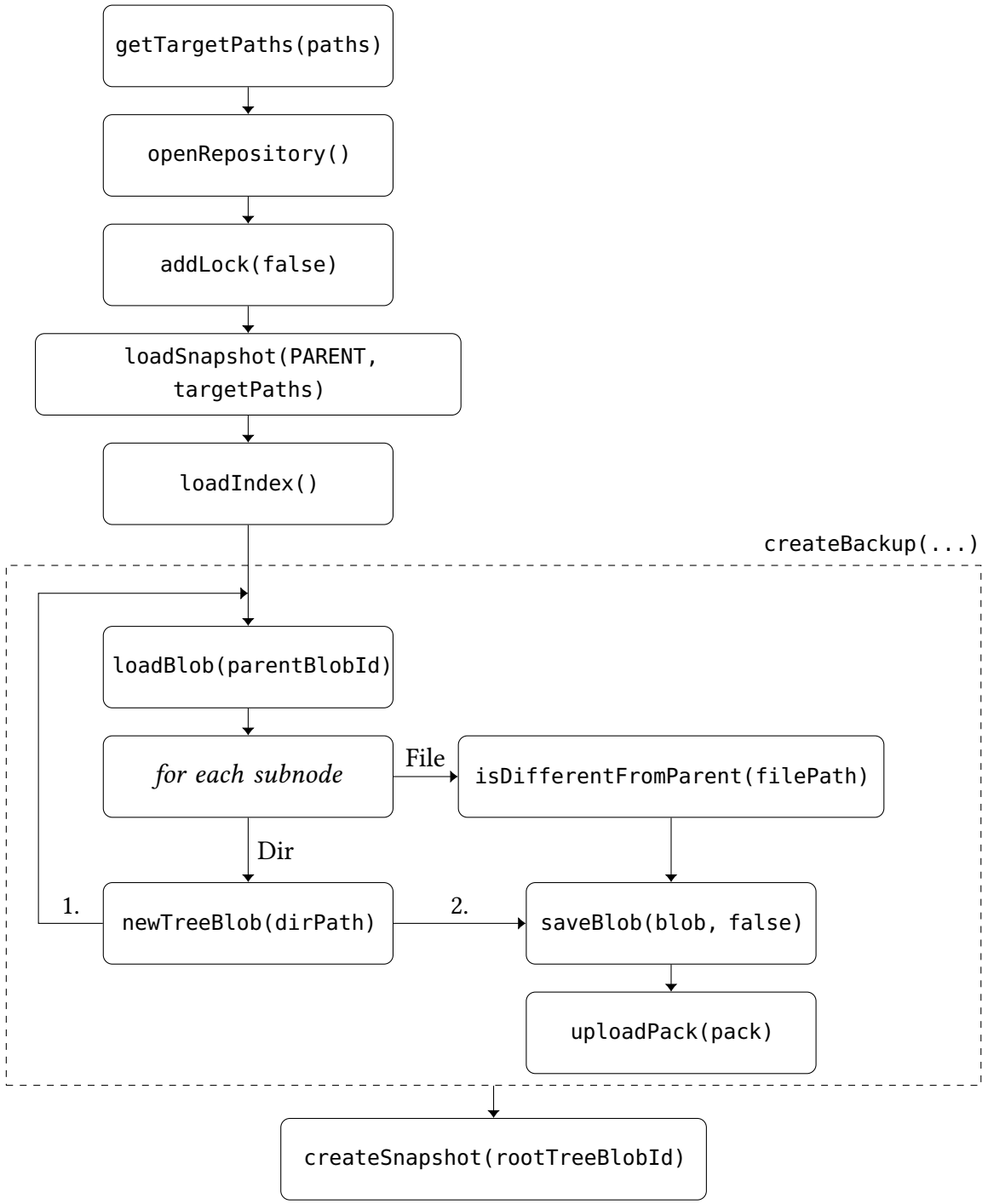

Figure 3.1.: Ablauf von Restics *backup* Befehl

3.2.5. Abstrakte Modellierung des *backup* Befehls

Dieses Kapitel stellt die abstrakte Modellierung des *backup* Befehls vor. Der Fokus dieser Modellierung liegt darauf, ein möglichst einheitliche Modellierung der drei betrachteten Restic-Befehle aus der Sicht von S_{Backup} zu erreichen. Damit lassen sich die Restic-Befehle

einheitlicher in die vorgestellten Sicherheitsspiele einbinden. Die abstrakte Modellierung abstrahiert keine Sicherheitsprimitive und orientiert sich an der realen Modellierung und dem vorgestellten Pseudocode. Die abstrakte Modellierung stellt lediglich eine einheitliche, abstrakte Betrachtungsweise der Restic-Befehle mit dem Fokus auf dem Nachrichtenaustausch aus der Sicht von S_{Backup} vor. Es wird explizit die Sicht von S_{Backup} verwendet, da in dieser Masterarbeit nur ein Bedrohungsszenario betrachtet wird, in dem ein Angreifer S_{Backup} korrumpiert hat.
Es wird für die abstrakte Modellierung der Aufruf des *backup* Befehls zu einem Tupel abstrahiert, das ein virtuelles Verzeichnissystem verwendet. Die Verwendung des virtuellen Verzeichnissystems und dessen Verwendung in der abstrakten Modellierung wird im Folgenden erklärt.

3.2.5.1. Modellierung mit Verzeichnissystem

Um die abstrakte Modellierung formaler betrachten zu können, werden *backup* und *restore* Befehle jeweils direkt mit einem virtuellen Verzeichnissystem modelliert. Dazu wird einem *backup* Befehl direkt ein virtuelles Verzeichnissystem VS übergeben, sowie Werte für die Optionen O des *backup* Befehls und den Target-Pfaden TP. Der Befehl wird daraufhin auf dem virtuellen Verzeichnissystem mit den entsprechenden Optionen und Target-Pfaden ausgeführt.
In der Realität bestimmt Restic erst zur Laufzeit durch die Target-Pfade von welchen Verzeichnissen und Dateien ein Backup erstellt werden soll. Außerdem besitzt Restic in der Realität eine Filter-Funktion, um Pfade und Dateien zu filtern und vom Backup zu exkludieren. Des Weiteren können in der Realität einem *backup* Befehl Pfade übergeben werden, die zusätzlich exkludiert werden sollen. Durch die Übergabe eines virtuellen Verzeichnissystems können sowohl die Target-Pfade, als auch die Filter-Funktion, abstrahiert werden (siehe Kapitel 2.1.6). Wie in Kapitel 2.1.6 beschrieben, beinhaltet ein virtuelles Verzeichnissystem nur die Dateien und Verzeichnisse, von denen auch ein Backup durchgeführt wird. Das heißt, in einem virtuellen Verzeichnissystem sind bereits die Dateien und Verzeichnisse entfernt worden, die durch Restics Filter-Funktion exkludiert werden würden. Damit ist die Ausführung eines *backup* Befehls auf einem virtuellen Verzeichnissystem ohne Filter-Funktionen bis auf kleine Unterschiede identisch zu der Ausführung auf einem realen Verzeichnissystem mit den entsprechenden Filter-Funktionen und den gleichen Target-Pfaden. Diese Unterschiede bei der Ausführung sind zum einen Verzögerungen bei der Ausführung und der Aufbau der Snapshot-Datei, die für diesen *bachup* Befehl erstellt wird. In der Realität bekommt ein Angreifer mit Zugriff auf S_{Backup} nicht mit, wenn eine Datei oder ein Verzeichnis exkludiert wird, aber es kommt zu einer längeren Verarbeitungszeit durch Restic. Diese Verarbeitungszeit ist eventuell durch den Angreifer messbar, aber für den Angreifer ist nicht ersichtlich, ob diese minimale Verzögerung durch das Exkludieren entstanden ist, oder durch andere Ereignisse, wie den Nachrichtenaustausch oder eine hohe Auslastung des Systems, auf dem Restic läuft. Relevanter ist jedoch die Auswirkung auf die Snapshot-Datenstruktur.
Da in einem Snapshot die exkludierten Pfade eingetragen werden, unterscheiden sich diese

Datenstrukturen bei beiden Modellierungen. Für die exkludierten Pfade wird bei der abstrakten Modellierung nie etwas für dieses Feld in die Snapshot-Datenstrukturen eingetragen. In der Realität würden sich zwei Snapshot-Datenstrukturen mit einer unterschiedlichen Anzahl exkludierter Pfade in ihrer Größe unterscheiden. Das würde für die Leakage des *backup* Befehls bedeuten, dass die Anzahl und Größe der exkludierten Pfade auch Teil der Leakage wäre. Ein Angreifer könnte damit eventuell Informationen über die mögliche Größe des Verzeichnissystems erlangen, da mit steigender Größe der exkludierten Pfade auch die Wahrscheinlichkeit für ein umfangreiches Verzeichnissystem steigt. Allerdings können auch Pfade exkludiert werden, die nicht in dem Verzeichnissystem vorkommen, wodurch diese Erkenntnis mit Respekt betrachtet werden sollte. Da die Modellierung der exkludierten Pfade mehr Aufwand mit sich bringt als Vorteile, werden die exkludierten Pfade in der abstrakten Modellierung durch ein virtuelles Verzeichnissystem ersetzt. In der abstrakten Modellierung ist das Feld für die exkludierten Pfade in allen Snapshot-Datenstrukturen konsistent leer. Dadurch beeinflussen die exkludierten Pfade die Leakage eines *backup* Befehls nicht und die Chance des Angreifers eins der Sicherheitsspiele zu gewinnen steigt nicht durch diese Änderung. Außerdem werden die exkludierten Pfade in der Snapshot-Datenstruktur bei keinem der betrachteten Befehle verwendet und dadurch wirkt sich diese Änderung bei der abstrakten Modellierung nicht auf den Ablauf anderer betrachteter Restic-Befehle aus.

3.2.5.2. Modellierte Optionen des *backup* Befehls

Es ist nicht nötig alle Optionen aus Tabelle 3.3 für den *backup* Befehl in dem abstrakt modellierten Protokoll abzubilden. Dadurch, dass die Target-Pfade in einer Liste TP zusammengefasst werden, fallen die Optionen *files_from*, *files_from_verbatim* und *files_from_raw* weg. Außerdem wird bei der abstrakten Modellierung davon ausgegangen, dass der *backup* Befehl auf einem bekannten Repository von einem bekannten Benutzer ausgeführt wird. Daher sind ebenfalls die *repository_path* und *key_hint* Optionen für dieses Protokoll überflüssig. Die übrigen Optionswerte, mit denen ein *backup* Befehl durchgeführt wird, werden in einem Tupel $O := (CONNECTIONS, PARENT, PACK\_SIZE)$ zusammengefasst.

3.2.5.3. Fehlendes Benutzerpasswort bei der Modellierung

Die abstrakte Modellierung wird in einem späteren Kapitel benutzt, um zwei Sicherheitsspiel für Restic zu definieren. Bei den Sicherheitsspielen nimmt der Angreifer die Rolle des Benutzers ein, indem der Angreifer bestimmt, welche Restic-Befehle ausgeführt werden. In der Realität kennt der Benutzer das Benutzerpasswort, aber der Angreifer nicht und der Benutzer wurde das Benutzerpasswort zusammen mit dem Tupel des Restic-Befehls an Restic übergeben (siehe Diagramm 3.2). Durch die Verschmelzung von Benutzer und Angreifer in den Sicherheitsspielen wird in der abstrakten Modellierung jedoch darauf verzichtet den Benutzerpasswort-Austausch zu modellieren und es wird davon ausgegangen, dass Restic bereits das Benutzerpasswort des Benutzers oder den daraus abgeleiteten Userkey

kennt. Durch diese Annahme ist die abstrakte Modellierung direkt auf die Sicherheitsspiele übertragbar, da dort das Resticsystem ebenfalls den Userkey kennt.

3.2.5.4. Modellierter Nachrichtenaustausch

S_{Backup} stellt selbst keine Berechnungen an und dient nur als Datenlieferant und Datenspeicher für S_{Restic}. Das bedeutet ein Angreifer mit Zugriff auf S_{Backup} kann nur den Datenaustausch beobachten oder manipulieren. Daher wird für die abstrakte Modellierung des *backup* Befehls ein Protokoll mit dem Fokus auf den Nachrichtenaustausch vorgestellt. Dieses Protokoll 3.2 modelliert für eine bessere Übersichtlichkeit nicht alle ausgetauschten Nachrichten. Vor der Anfrage einer bestimmten Datei kommt es oft dazu, dass alle Metadaten von Dateien dieses Typs von S_{Restic} bei S_{Backup} angefragt werden. Damit kann Restic eine bestimmte Datei auswählen und diese von S_{Backup} anfordern. Die Anfrage aller Metadaten erfolgt jedoch immer identisch und ist für einen beobachtenden Angreifer irrelevant. Falls die Anfrage aller Metadaten für einen manipulierenden Angreifer relevant wird, wird darauf explizit Bezug genommen. Daher werden die ausgetauschten Metadaten in der abstrakten Modellierung nicht modelliert und S_{Restic} fragt direkt die ausgewählte Datei an, als ob S_{Restic} Zugriff auf alle Metadaten aller Dateien im Repository besitzt. Für die bessere Übersichtlichkeit werden Anfragen von S_{Restic} für bestimmte Dateien nicht als Nachrichtenaustausch modelliert und es werden von S_{Backup} direkt zum entsprechenden Zeitpunkt die angefragten Datenstrukturen an S_{Restic} gesendet.

3.2.5.5. Formale Betrachtung des *backup* Befehls

In den betrachteten Szenarien dieser Masterarbeit werden die Restic-Befehle immer aus der Sicht des Angreifers und damit aus der Sicht des Backupsystems S_{Backup} betrachtet. Damit ist jeder Restic-Befehl und damit auch der *backup* Befehl durch die zwischen S_{Restic} und S_{Backup} ausgetauschten Datenstrukturen ohne Informationsverlust für einen Angreifer eindeutig beschreibbar. Die Notation $backup_R(VS, O, TP)$ beschreibt die abstrakte Modellierung des *backup* Befehls bezüglich einem Repository R über dem Verzeichnissystem VS, mit den Optionen O und auf den Target-Pfaden TP. $B := (VS, O, TP)$ wird auch als Befehlstupel bezeichnet.
Für eine formale Betrachtung der Restic-Befehle wird ein Tupel τ eingeführt, das die komplette Befehlsausführung eines Restic-Befehls vollständig beschreibt. Dieses Tupel $\tau := (e_1, \ldots, e_{max})$ besteht aus mehreren Events e_i und wird Event-Tupel genannt. Für das Event-Tupel τ eines *backup* Befehls wird auch geschrieben $\tau := backup_R(VS, O, TP)$.

Definition 3.2.1 (Event-Tupel) *Ein Event-Tupel $\tau := events(B, type(B), R)$ gehört immer zu einem Restic-Befehl B vom Typ $type(B)$ und beschreibt, dessen Ausführung auf dem Repository R aus Sicht des Backupsystems S_{Backup}. Das Event-Tupel $\tau := (e_1, \ldots, e_{max})$ besteht aus mehreren Events e_i. τ besitzt einen Event-Eintrag e_i für jede zwischen S_{Restic} und S_{Backup} ausgetauschte Restic-Datenstruktur DS_i, während der Ausführung eines Restic-Befehls. Ohne*

Beschränkung der Allgemeinheit wird davon ausgegangen, dass die e_i in τ so angeordnet sind, dass gilt $delay_i \leq delay_{i+1}$.

Definition 3.2.2 (Event) *Ein Event beschreibt eine Interaktion zwischen S_{Restic} und S_{Backup}, bei der eine Datenstruktur zwischen diesen beiden Systemen ausgetauscht wird. Interaktionen, bei denen lediglich Metadaten, wie Restic-IDs und Dateigrößen, zwischen S_{Restic} und S_{Backup} ausgetauscht werden, werden nicht als Event bezeichnet.*
Ein Event $e_i := (op_i, delay_i, type_i, data_i, blobInfo_i)$ ist ebenfalls ein Tupel mit den folgenden Einträgen. Jede Interaktion zwischen S_{Restic} und S_{Backup} ist durch eine von drei Operationen $op_i \in \{send, receive, delete\}$ darstellbar. op_i gibt an, ob die zu e_i gehörige Datenstruktur DS_i von S_{Backup} angefordert wird ($receive$), an S_{Backup} gesendet und im Repository gespeichert wird ($send$) oder von S_{Backup} aus dem Repository gelöscht werden soll ($delete$).
Der Zeitpunkt, zu dem eine Interaktion stattfindet, wird mit $delay_i$ angegeben. Events gehören zwangsläufig zu einem Event-Tupel und der Wert $delay_i$ ist nur innerhalb dieses Event-Tupels aussagekräftig. $delay_i$ gibt die zeitliche Verzögerung seit dem ersten Event e_1 des zugehörigen Event-Tupels an, bis es zu der Interaktion kommt, die dieses Event e_i beschreibt.
Jedes Event enthält den Typ $type_i := type(DS_i)$ der Datenstruktur DS_i, für die die Interaktion stattfindet. Es gilt $type_i \in \{Key, Lock, Config, Snapshot, Index, Pack, BlobChunk\}$. Bei den drei betrachteten Restic-Befehlen ist die einzige Interaktion mit Blobs aus dem Repository das Anfordern von einzelnen Blobs oder Blob-Chunks. In beiden Fällen gilt für ein Event e_i, das die Interaktion bezüglich eines oder mehrere Blobs aus dem gleichen Pack beschreibt, $op_i = receive$ und $type_i = BlobChunk$.
Der Inhalt $data_i$ der Datenstruktur DS_i, wie sie in dem Repository auf S_{Backup} gespeichert wird, ist ebenfalls Teil eines Events. In den meisten Fällen ist $data_i$ damit die komprimierte, verschlüsselte und authentifizierte Datenstruktur $data_i := encrypt(comp(DS_i))$. Jede Datenstruktur bis auf einzelne Blobs oder Blob-Chunks wird im Repository in einer eigenen Datei gespeichert. Damit ist $data_i$ für alle Events mit $type_i \neq BlobChunk$ genau der Inhalt der Datei, die zu der Datenstruktur DS_i gehört. Für Events mit $type_i = BlobChunk$ ist $data_i$ der Inhalt der Pack-Datei, aus der die Blobs angefordert werden. Es gibt niemals eine Interaktion, bei der meherer Blobs aus unterschiedlichen Packs mit einer einzigen Interaktion angefordert werden, wodurch $data_i$ auch für Blob-Chunks wohldefiniert ist. Wenn $data_i$ jedoch das ganze Pack ist, ist aus dem zugehörigen Event nicht herauslesbar, welche Blobs genau aus diesem Pack angefordert wurden. Dafür existiert der letzte Eintrag $blobInfo_i$ eines Events.
Es gilt $blobInfo_i = \perp$ für alle Events e_i mit $type_i \neq BlobChunk$. Für Events mit $type_i = BlobChunk$ ist die Blob-Info definiert als $blobInfo_i = (start, end)$. Eine Interaktion bei der ein oder mehrere Blobs aus einem Pack angefordert werden, fordert immer eine zusammenhängende Menge von Blobs aus diesem Pack an. Damit fordert eine solche Interaktion immer einen zusammenhängenden Bytestring aus der Pack-Datei an. Dieser Bytestring und damit auch der Blob-Chunk ist eindeutig definiert durch die Startposition und Endposition dieses Bytestrings in der Pack-Datei. Damit gibt $start$ die Position an, an der die Menge von Blobs in dem zugehörigen Pack startet und end gibt die Position an, an der die Menge von Blobs in dem zugehörigen Pack endet.

Theorem 1 *Die Betrachtung eines Repositorys R_{init} während der realen Ausführung eines Restic-Befehls B auf R_{init} ist bezüglich des Informationsgewinns äquivalent zu der Betrachtung*

des Repository-Zustands R_{init} vor der Ausführung des Befehls B und dem zu B gehörigen Event-Tupel τ.

Es gibt genau drei Hauptarten von Interaktion zwischen S_{Restic} und dem Repository auf S_{Backup}. Diese Arten von Interaktionen sind das Speichern von Daten in dem Repository, das Lesen von Daten aus dem Repository und das Löschen von Daten aus dem Repository. Es gibt in der realen Modellierung eine weitere Art von Interaktionen, die das Anfordern aller Metadaten von einem Datenstrukturtypen darstellt. Metadaten bezeichnen hierbei die Restic-ID der Datenstruktur, sowie die Größe der Datei in der diese Datenstruktur gespeichert ist. Dieser Interaktion folgt immer das Anfordern aller Datenstrukturen dieses Typs mit der Hauptinteraktion "Anfordern" (Receive-Event). In der Hauptinteraktion "Anfordern" sind die Metadaten ebenfalls enthalten. Die Größe der Datei entspricht der Größe von $data_i$ des Receive-Events und die Restic-ID der Datenstruktur lässt sich ebenfalls aus $data_i$ berechnen, wie gleich zu sehen ist. Das Weglassen dieser zusätzlichen Interaktionsart führt zu keinem Informationsverlust.
Jede der drei Hauptarten von Interaktion wird durch ein Event $e_i \in \tau$ modelliert. Dabei bezeichnet op_i die Art der Interaktion. In der realen Modellierung wird für ein Event mit $op_i = delete$ die Restic-ID und der Typ der zu löschenden Datenstruktur an S_{Backup} gesendet. Der Typ der Datenstruktur ist ebenfalls in $type_i$ definiert. In Restic wird die ID einer Datenstruktur deterministisch mit einer Hashfunktion (siehe Kapitel 2.3.3.2) berechnet. Dabei wird kein geheimer Schlüssel verwendet und die ID wird nur über $data_i$ berechnet. Damit ist auch die Restic-ID, aus dem Event e_i herleitbar, sofern gilt $type_i \neq BlobChunk$, da für Blob-Chunks $data_i$ eine Pack-Datenstruktur ist. Das ist im Falle einer Delete-Interaktion jedoch kein Problem, da eine Delete-Interaktion niemals für einen Blob oder einen Blob-Chunk ausgeführt wird.
Bei einer Send-Interaktion wird in der realen Modellierung ebenfalls der Inhalt der Datenstruktur an das Repository gesendet, sowie der Typ der Datenstruktur und die ID der Datenstruktur. In dem zugehörigen Event befinden sich ebenfalls die beiden Informationen $data_i$ und $type_i$, wobei die ID ebenfalls wieder aus $data_i$ berechnet werden kann. Auch bei einer Send-Interaktion gilt wieder, dass diese Aktion niemals für einzelne Blobs oder Blob-Chunks ausgeführt wird.
Die letzte Interaktionsart ist die Receive-Interaktion, bei der S_{Restic} eine Datenstruktur, einen Blob oder einen Blob-Chunk aus dem Repository anfordert. Im Falle einer Datenstruktur, die kein Blob ist, wird die ID und der Typ der Datenstruktur an S_{Backup} geschickt, worauf S_{Backup} den Inhalt der zugehörigen Datei auf dem Repository zurückschickt. In einem Receive-Event ist ebenfalls der Type ($type_i$) und der Inhalt der entsprechenden Datei enthalten ($data_i$) und die angeforderte ID lässt sich ebenfalls aus $data_i$ berechnen. Für einen Blob oder Blob-Chunk wird die ID des Packs, das diese Blobs enthält an S_{Backup} gesendet. Außerdem wird die Startposition und Endposition des Bytestrings, der den Blob oder Blob-Chunk repräsentiert, an S_{Backup} gesendet. S_{Backup} schickt den entsprechenden Bytestring an S_{Restic} zurück. Ein Receive-Event mit $type_i = BlobChunk$ enthält immer die Blob-Info, die Startposition ($start$) und Endposition (end) des Bytestrings in dem zugehörigen Pack darstellt. Außerdem ist für ein solches Event $data_i$ der Inhalt des gesamten Packs, aus dem die Blobs stammen. Damit kann aus $data_i$ die ID der Pack-Datei berechnet werden, aus der der Bytestring angefordert

wird und mit *start* und *end* kann auch der Bytestring, der zurückgeschickt wird aus $data_i$ extrahiert werden.
Jedes Event eines Event-Tupels ist außerdem mit einem eindeutigen Zeitstempel $dalay_i$, wodurch der genaue Zeitpunkt der Interaktion festgelegt werden kann.
Kennt der Beobachter also den Zustand des Repositorys vor der Ausführung eines Befehls und das Event-Tupel dieses Befehls, kann der Beobachter durch das Durchführen aller Interaktionen (Events), den Repository-Zustand nach Ausführung des Befehls herstellen. Außerdem erhält der Beobachter durch das Event-Tupel alle Informationen, die er auch durch die reale Beobachtung der Ausführung des Restic-Befehls B auf R_{init} erhalten hätte. □

Definition 3.2.3 (Ausführen eines Befehls) *Sei R ein Repository und B ein Befehlstupel vom Typ $type(B)$.*

$execute_R(\tau)$ oder $execute(backup_R(VS, O, TP))$ oder bezeichnet die Ausführung des durch τ repräsentieren *backup* Befehls. Für alle $e_i \in \tau$ wird zum Zeitpunkt $delay_i$ von S_{Restic} die Operation op_i für $data_i$ ausgeführt. Damit ist für S_{Backup} und einen Angreifer $execute(\tau)$ nicht unterscheidbar von der eigentlichen Ausführung des *backup* Befehls nach Modellierung 3.2.

3.2.5.6. Ablauf der abstrakten Modellierung

In der abstrakten Modellierung bekommt S_{Restic} ein virtuelles Verzeichnissystem VS dieses *backup* Befehls, die Werte O für die Optionen dieses *backup* Befehls und die realen Target-Pfade TP dieses *backup* Befehls übergeben. S_{Restic} emuliert ein System, das nur aus den Verzeichnissen und Dateien des virtuellen Verzeichnissystems VS besteht und führt daraufhin den *backup* Befehl auf diesem System aus. Der *backup* Befehl wird mit den Optionen O ausgeführt und erhält die Target-Pfade TP beim Start dieses Befehls. Die abstrakte Modellierung von Restics *backup* Befehl für ein Repository R ist in Diagramm 3.2 dargestellt. Da in dieser Masterarbeit ein *backup* Befehl in der Regel immer für dasselbe Repository R und den gleichen Benutzer ausgeführt wird, wird sowohl der Verbindungsaufbau zu R und die genaue Bestimmung des Keys K nicht modelliert.

1. Als Erstes übergibt der Benutzer das Tupel (VS, O, TP) an S_{Restic}. VS stellt dabei das virtuelle Verzeichnissystem dar, auf dem der *backup* Befehl aufgerufen wird und $O := (CONNECTIONS, PARENT, PACK\_SIZE)$ enthält, die betrachteten Optionen für diesen *backup* Befehl. TP sind die realen Target-Pfade mit denen dieser *backup* Befehl aufgerufen wird.
2. Durch Funktion 19 erhält S_{Restic} die Key-Datenstruktur K für den Benutzer, der den *backup* Befehl ausführt und damit das Benutzerpasswort kennt. Außerdem erhält S_{Restic} die Config-Datei C des Repositorys R, für das der *backup* Befehl ausgeführt wird.
3. Die Funktion 21 fordert zunächst alle Lock-Dateien $\langle L_1, \ldots, L_{max} \rangle$ des Repositorys an und schickt dann selbst eine nicht exklusive Lock-Datei $\tilde{L}_{new}$ an S_{Backup}.

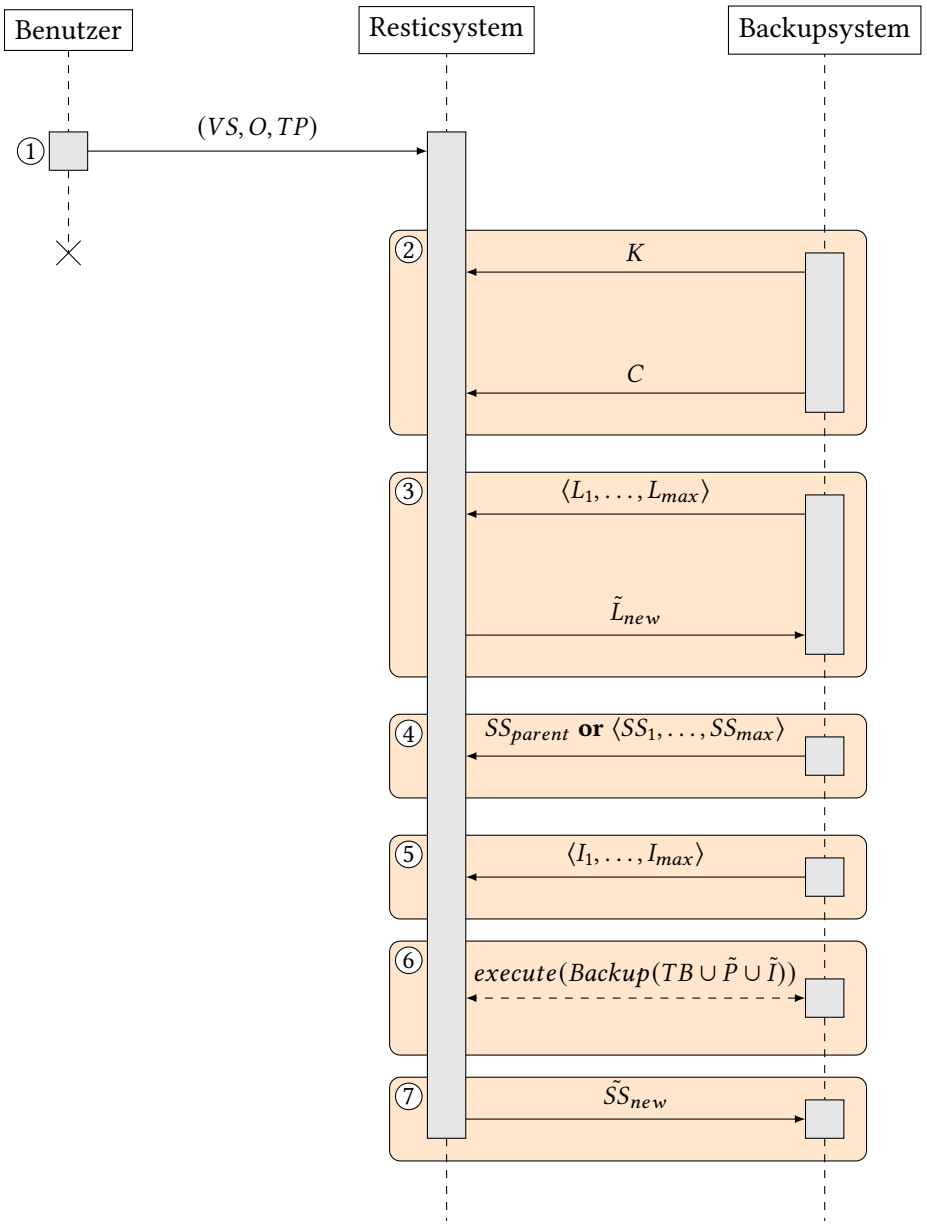

Figure 3.2.: Abstraktes Kommunikationsprotokoll des *backup* Befehls

4. Enthält O ein gültiges Snapshot-ID-Präfix für $PARENT$, wird die Snapshot-Datei SS_{parent} von S_{Restic} angefordert und von S_{Backup} an S_{Restic} geschickt. Ein gültiges Snapshot-ID-Präfix ist ein ID-Präfix für einen Snapshot, dessen Target-Pfade alle Pfade des Verzeichnissystem VS und damit auch alle Target-Pfade TP enthalten. Enthält O keinen gültigen Wert für $PARENT$, werden alle Snapshot-Dateien $\langle SS_1, \ldots, SS_{max} \rangle$ an S_{Restic} gesendet. Restic bestimmt daraufhin selbst einen Parent-Snapshot SS_{parent}, wie in Funktion 23 beschrieben ist.

5. Die Funktion 25 fordert alle Index-Dateien $\langle I_1, \ldots, I_{max} \rangle$ des Repositorys R von S_{Backup} an.
6. Im Laufe der Ausführung von Funktion 2 werden nach und nach Tree-Blobs $TB := \{TB_1, \ldots, TB_{max}\}$ des Verzeichnisbaums T_{parent} zu SS_{parent} von S_{Backup} angefordert. Konnte kein SS_{parent} bestimmt werden, gilt $TB = \{\}$. Außerdem werden Pack-Dateien $\tilde{P} := \{\tilde{P}_1, \ldots, \tilde{P}_{max}\}$ von S_{Restic} erstellt und an S_{Backup} zum Speichern gesendet. Diese Packs bestehen aus den für VS erstellten Data-Blobs und Tree-Blobs. Zu diesen Packs und Blobs, werden ebenfalls Index-Dateien $\tilde{I} := \{\tilde{I}_1, \ldots, \tilde{I}_{max}\}$ von S_{Restic} erstellt und an S_{Backup} zum Speichern gesendet. Der Verzeichnisbaum T_{backup} wird nichtdeterministisch traversiert und die Blobs werden nichtdeterministisch zu den erstellten Packs zugeordnet. Ebenso werden die Packs und Indices in eigenen parallelen Prozessen an S_{Backup} geschickt, wodurch auch die Sendereihenfolge der Packs und Indices und die Empfangsreihenfolge der Tree-Blobs nichtdeterministisch ist. Daher wird dieser Nachrichtenaustausch wie folgt mit $\tau_{backup} := Backup(TB \cup \tilde{P} \cup \tilde{I})$ abgebildet, wobei τ_{backup} das Teiltupel von $\tau := backup_R(VS, O, TP)$ ist, das nur die Datenstrukturen aus den Mengen TB, $\tilde{P}$ und $\tilde{I}$ enthält. Damit ist $Backup(TB, \tilde{P}, \tilde{I})$ der Teil von τ, der durch die Funktion 2 entsteht.
   Genau wie mit $execute(\tau)$ werden auch mit $execute(\tau_{backup})$ oder $executeBackup(TB, \tilde{P}, \tilde{I})$ alle $e_i \in \tau_{backup}$ zum Zeitpunkt $delay_i$ von S_{Restic} ausgeführt.
7. Zum Schluss wird durch Funktion 3 eine Snapshot-Datei SS_{new} für diesen *backup* Aufruf erstellt und an S_{Backup} gesendet.

3.3. Restore Befehl

Der *restore* Befehl von Restic kann für ein gegebenes Restic-Repository aufgerufen werden. Restic wählt daraufhin basierend auf dem übergebenen Target-Pfad und dem Wert der *snapshot_id* Option einen Snapshot eines früheren Backups. Basierend auf diesem Snapshot werden die Dateien und Verzeichnisse des zugehörigen Backups an dem Target-Pfad auf S_{Restic} wiederhergestellt.
Zuerst werden die möglichen Optionen vorgestellt, mit denen ein *restore* Befehl ausgeführt werden kann.
Daraufhin stellt dieses Kapitel zwei Modellierungen des *restore* Befehls als kryptografisches Protokoll vor. Zunächst wird die reale Modellierung vorgestellt, bei der der *restore* Befehl in mehrere detailliert beschriebene Blöcke unterteilt wird. Diese Blöcke sind unter anderem auch im Anhang beschrieben, falls sie in anderen Restic-Befehlen wiederverwendet werden (siehe Kapitel A.1).
Danach wird die abstrakte Modellierung des *restore* Befehls als kryptografisches Protokoll vorgestellt. Diese Modellierung wird für die später vorgestellten Sicherheitsspiele verwendet. Bei dieser Modellierung wird der Fokus auf den Nachrichtenaustausch zwischen S_{Restic} und S_{Backup} gelegt.

3.3.1. Prämisse

Es wird davon ausgegangen, dass bereits ein Repository auf S_{Backup} initialisiert wurde. Außerdem wird davon ausgegangen, dass mit dem *backup* Befehl mindestens ein Backup von Dateien auf S_{Restic} für dieses Repository erstellt wurde. Das bedeutet, dass mindestens eine Key-Datei, eine verschlüsselte und authentifizierte Config-Datei, sowie eine Snapshot-Datei und zugehörige Pack-Dateien und Index-Dateien in diesem Repository auf dem Backupsystem existieren.

3.3.2. Optionen des *restore* Befehls

Bei dem Aufruf des *restore* Befehls können verschiedene Optionen gesetzt oder konfiguriert werden, die den Ablauf des *restore* Befehls modifizieren. Diese Optionen sind in Tabelle 3.4 aufgelistet.

3.3.3. Restic-Funktionen für den *restore* Befehl

In diesem Kapitel werden Restic-Funktionen erklärt, die nur von dem *restore* Befehl verwendet werden. Diese Funktionen werden in einem späteren Kapitel verwendet, um den Ablauf des *restore* Befehls zu modellieren.

3.3.3.1. Dateien wiederherstellen

Die Funktion 4 stellt alle Dateien auf S_{Restic} aus einer Liste `restoreInfo` wieder her. Die übergebene Liste `restoreInfo` enthält unter anderem die IDs alle Data-Blobs, die für die wiederherzustellende Dateien benötigt werden. Dazu sind die IDs der Data-Blobs in der Liste `restoreInfo` nach den Packs gruppiert in denen diese Data-Blobs vorkommen. Jedes Pack in `restoreInfo` besitzt eine Liste `packInfo` von Tupeln. Die Tupel in `packInfo` enthalten Informationen über die wiederherzustellende Datei `fileInfo` und die Liste von Data-Blob-IDs `blobIds`, der dafür benötigten Data-Blobs in diesem Pack.
Restic startet so viele parallele Restore-Prozesse, wie die *connections* Option angibt. Jeder dieser Restore-Prozesse ist für die Wiederherstellung von Data-Blobs aus einem Pack für eine Datei verantwortlich. Dazu werden auf die parallelen Restore-Prozesse die IDs der einzelnen Packs und die IDs der in dem Pack enthalten Data-Blobs für die Datei nach und nach verteilt mit `dispatch(packId, blobIds)`. Jeder Prozess, der diese Daten erhält, startet seine Ausführung mit der Funktion 29.
Dabei fordert S_{Restic} Blob-Chunks von S_{Backup} an und führt die Funktion 5 auf jedem Blob aus der übergebenen Liste `blobIds` aus. Die Funktion 5 entspricht der `execute()` Funktion, die die Funktion 29 benötigt. Der an die Funktion 5 übergebene Data-Blob wird in der wiederherzustellenden Datei an der richtigen Position in dieser Datei wiedereingesetzt mit `restoreBlob(fileInfo, blob)`.

Name	Beschreibung
REPOSITORY_PATH	Ort, an dem das Repository liegt. Dies wird benötigt, um eine Verbindung zum S_{Backup} aufzubauen.
KEY_HINT	ID-Präfix einer Key-Datei des Repositorys. Zum gezielten Anfordern einer Key-Datei.
CONNECTIONS	Restic verwendet diesen Wert, um die Anzahl paralleler Prozesse zu bestimmen, die für einige Funktionen erstellt werden.
SNAPSHOT_ID	Die Restic-ID des Snapshots des Backups, dessen Daten wiederhergestellt werden sollen.
OVERWRITE	Diese Option kann verschiedene Werte annehmen (z.B. *OverwriteIfChanged* oder *OverwriteIfNewer*). Je nachdem welcher Wert für diese Option verwendet wird, wird entschieden, ob Dateiinhalte gelöscht werden und durch die Backup-Daten ersetzt werden oder nicht. *OverwriteAlways*: Dateiinhalte werden immer durch die Backup-Daten ersetzt *OverwriteIfChanged*: Dateiinhalte werden nur durch die Backup-Daten ersetzt, wenn sich die Modifikationszeit oder die Dateigröße von der der Backup-Daten unterscheiden. *OverwriteIfNewer*: Dateiinhalte werden nur durch die Backup-Daten ersetzt, wenn die Modifikationszeit der Backup-Daten aktueller ist *OverwriteNever*: Existiert eine Datei auf S_{Restic}, wird deren Dateiinhalt niemals durch Backup-Daten ersetzt

Table 3.4.: Optionen des *restore* Befehls

Nachdem alle wiederherzustellenden Dateien mit den angegebenen Data-Blobs wiederhergestellt wurden, ist die Funktion 4 zu Ende.

3.3.4. Ablauf der realen Modellierung des *restore* Befehls

Dieses Kapitel beschreibt den Ablauf nach dem Aufruf des Restic *restore* Befehls.
Restic sucht das vom Benutzer angegebene Repository auf S_{Backup}, baut eine Verbindung zu dem Repository auf und baut eine interne Repository-Datenstruktur auf mit Funktion 19.
Bevor Restic weiter mit dem Repository auf S_{Backup} interagiert, wird eine nicht exklusive Lock-Datei mit Funktion 21 dem Repository hinzugefügt. Dadurch wird verhindert, dass mögliche andere Restic-Distributionen exklusiv auf das Repository zugreifen können.

Funktion 4 Restores all files from a list

```
Require: CONNECTIONS
  procedure restoreFiles(restoreInfo)
      startParallelProcesses(CONNECTIONS)
      for all (packId, packInfo) in restoreInfo do
          for all (fileInfo, blobIds) in packInfo do
              dispatch(packId, blobIds)          ▷ Use fileInfo in execute_restoreBlob()
          end for
      end for
      waitBarrier()
  end procedure
```

Funktion 5 Restores a blob from a file

```
Require: fileInfo
  procedure execute_restoreBlob(blob)
      restoreBlob(fileInfo, blob)
  end procedure
```

Für den *restore* Befehl kann beim Aufruf ein Präfix einer Snapshot-ID als Wert für die *snapshot_id* Option verwendet werden. Außerdem muss dem Befehl ein Target-Pfad übergeben werden, der wiederhergestellt werden soll. Restic fordert entweder den durch den ID-Präfix spezifizierten Snapshot an oder den Latest-Snapshot, dessen Target-Pfade den Target-Pfad dieses *restore* Aufrufs enthalten. Dafür wird die Funktion 23 verwendet. Der ausgewählte Snapshot SS_{parent} wird in der internen Repository-Datenstruktur gespeichert. Um die Betrachtung der Befehle einheitlicher zu gestalten, wird der ausgewählte Snapshot, wie beim *backup* Befehl als SS_{parent} bezeichnet. Konnte kein Snapshot für den *restore* Befehl bestimmt werden, bricht die Befehlsausführung mit einem Fehler im Vergleich zum *backup* Befehl ab. Nun lässt sich Restic mit Funktion 25 alle Indices des Repositorys von S_{Backup} schicken und speichert diese in seiner internen Repository-Datenstruktur.
Restic traversiert nun den Verzeichnisbaum $T_{restore}$ des geladenen Snapshots beginnend bei dem Tree-Blob, der den Target-Pfad des *restore* Befehls repräsentiert. Dieser Tree-Blob des Target-Pfads ist der Wurzelknoten von $T_{restore}$ und damit ist der Target-Pfad das Root-Verzeichnis des zu $T_{restore}$ gehörigen Verzeichnissystems. S_{Restic} lässt sich diesen Tree-Blob von S_{Backup} mit Funktion 28 schicken. Restic iteriert über die Node-Einträge eines Tree-Blobs und überprüft, ob diese ein Unterverzeichnis oder eine Datei sind.
Bei einem Unterverzeichnis wird dieses Unterverzeichnis auf S_{Restic} falls noch nicht vorhanden erstellt und es wird der Tree-Blob dieses Verzeichnisses geladen. So wird der Verzeichnisbaum $T_{restore}$ rekursiv und nach dem Tiefensuche-Schema aufgebaut und die Verzeichnisse auf S_{Restic} erstellt.
Trifft Restic beim Traversieren auf einen Node, der eine Datei repräsentiert, wird dieser Node Datei-Node genannt. Ist die Datei, die zu dem Datei-Node gehört auf S_{Restic} vorhanden, Restic bestimmt basierend auf der *overwrite* Option und den Metadaten der vorhanden Datei, ob Teile dieser Datei wiederherstellt werden sollen (siehe Tabelle 3.4). Falls die Datei wiederhergestellt werden soll wird der Inhalt des *content* Felds der Datei-Node-Datenstruktur

(siehe Tabelle 2.14) und der Inhalt der vorhandenen Datei verglichen. Dazu wird die auf S_{Restic} vorhandene Datei in einzelnen Chunks zerlegt und die IDs dieser Chunks mit den IDs aus dem *content* Felds der Datei-Node-Datenstruktur verglichen. Unterscheiden sich zwei IDs, interpretiert Restic dies als eine Änderung in dem jeweiligen Chunk und wählt den Data-Blob, der zu der ID aus dem *content* Feld gehört, zum Wiederherstellen aus. Die für eine Datei ausgewählten Data-Blobs `blobIdsToLoad` werden nach den Packs, in denen sie enthalten sind, gruppiert und zu einer Liste `restoreInfo` mit `addBlobs(blobIdsToLoad)` hinzugefügt.
Nachdem $T_{restore}$ vollständig traversiert wurde und alle Data-Blobs zu der Liste `restoreInfo` hinzugefügt wurden, stellt Restic die für das Wiederherstellen ausgewählten Data-Blobs wieder her. Dazu iteriert Restic über die `restoreInfo` Liste der Packs und stellt nichtdeterministisch alle Dateien basierend auf den ausgewählten Data-Blobs mit der Funktion 4 wieder her.
Zum Schluss traversiert Restic erneut den Verzeichnisbaum $T_{restore}$, des wiederhergestellten Target-Pfads und ruft erneut rekursiv die Funktion 28 auf den einzelnen Tree-Blobs auf. Dabei werden Metadaten der Verzeichnisse, Metadaten der Dateien und Links wiederhergestellt.

3.3.5. Abstrakte Modellierung des *restore* Befehls

Dieses Kapitel stellt die abstrakte Modellierung des *restore* Befehls vor. Diese Modellierung ist möglichst einheitlich zwischen den drei betrachteten Befehlen, um diese einheitlicher in die vorgestellten Sicherheitsspiele einbinden zu können. Dazu wird für die abstrakte Modellierung der Aufruf des *restore* Befehls zu einem Tupel abstrahiert, das ein virtuelles Verzeichnissystem verwendet. Die Verwendung des virtuellen Verzeichnissystems und dessen Verwendung in der abstrakten Modellierung wird im Folgenden erleutert.

3.3.5.1. Modellierung mit Verzeichnissystem

Um die abstrakte Modellierung formaler betrachten zu können, werden *backup* und *restore* Befehle jeweils direkt mit einem virtuellen Verzeichnissystem modelliert. Wie in Kapitel 3.2.5.1 wird auch dem *restore* Befehl ein Verzeichnissystem VS, sowie Werte für die Optionen O des *restore* Befehls und der Target-Pfad TP übergeben. Anders als bei dem *backup* Befehl ist TP keine Liste von Target-Pfaden, sondern ein einzelner Target-Pfad, der auch beim realen Aufruf des *restore* Befehls übergeben wird. Wie auch bei der abstrakten Modellierung des *backup* Befehls, wird der *restore* Befehl auf dem virtuellen Verzeichnissystem mit den entsprechenden Optionen ausgeführt.
Das virtuelle Verzeichnissystem VS des *restore* Befehls ist eine Kopie, des eigentlichen Verzeichnisses und dessen Inhalt, auf das der Target-Pfad verweist. Zusätzlich enthält VS ebenfalls das Root-Verzeichnis des Betriebssystems von S_{Restic} und alle Verzeichnisse auf dem Pfad von dem Root-Verzeichnis bis hin zu dem Target-Pfad des *restore* Befehls. Damit kann S_{Restic} dieses VS für die abstrakte Modellierung emulieren und den *restore* Befehl darauf ausführen, ohne dass sich diese Ausführung von der realen Ausführung unterscheidet.

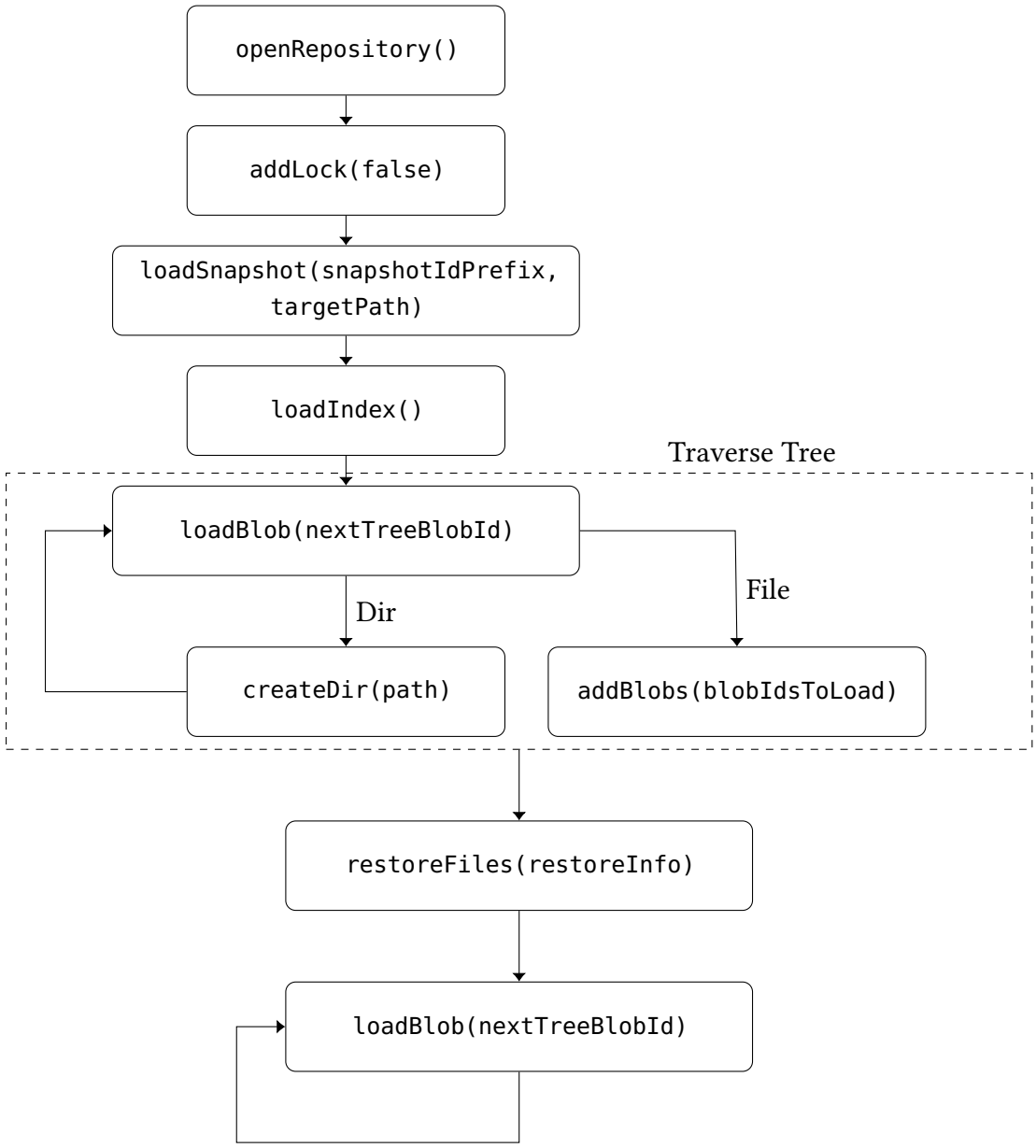

Figure 3.3.: Ablauf von Restics *restore* Befehl

3.3.5.2. Modellierte Optionen des *restore* Befehls

Es ist nicht nötig alle Optionen aus Tabelle 3.4 für den *restore* Befehl in dem abstrakt modellierten Protokoll abzubilden. Für die abstrakte Modellierung wird davon ausgegangen, dass der *restore* Befehl auf einem bekannten Repository von einem bekannten Benutzer ausgeführt wird. Daher sind ebenfalls die *repository_path* und *key_hint* Optionen für dieses Protokoll überflüssig. Die übrigen Optionswerte, mit denen ein *restore* Befehl durchgeführt wird, werden in einem Tupel $O := (CONNECTIONS, SNAPSHOT\_ID, OVERWRITE)$ zusammengefasst.

3.3.5.3. Modellierter Nachrichtenaustausch

Genau, wie für den *backup* Befehl wird die Übergabe des Benutzerpassworts von dem Benutzer an Restic nicht modelliert und es wird davon ausgegangen, dass Restic das Benutzerpasswort bereits kennt. Außerdem werden ausgetauschte Metadaten auch für den *restore* Befehl nicht modelliert. Falls die Anfrage der Metadaten von Dateien aus dem Repository für einen manipulierenden Angreifer relevant wird, wird darauf explizit Bezug genommen. Ebenfalls werden für die bessere Übersichtlichkeit Anfragen von S_{Restic} für bestimmte Dateien oder Blobs nicht als Nachrichtenaustausch modelliert. Es wird nur das Senden der entsprechenden Datenstrukturen von S_{Backup} an S_{Restic} zum entsprechenden Zeitpunkt modelliert.

3.3.5.4. Formale Betrachtung des *restore* Befehls

Der *restore* Befehl kann ebenfalls wie der *backup* Befehl aus Kapitel 3.2.5.5 nur mit den ausgetauschten Datenstrukturen ohne Informationsverlust dargestellt werden. Mit der Notation $restore_R(VS, O, TP)$ wird die abstrakte Modellierung des *restore* Befehls bezüglich einem Repository R über dem Verzeichnissystem VS, mit den Optionen O und auf dem Target-Pfad TP beschrieben. Daraus resultiert ebenfalls ein Tupel $\tau := restore_R(VS, O, TP)$, das alle Informationen über den Nachrichtenaustausch enthält, der bei der Ausführung des *restore* Befehls entsteht. Dieses Tupel τ ist wie das durch den *backup* Befehl entstehende Tupel ein Event-Tupel und besteht ebenfalls aus mehreren Events $\tau := (e_1, \ldots, e_{max})$. Für jeder zwischen S_{Restic} und S_{Backup} ausgetauschte Restic-Datenstruktur besitzt τ einen Eintrag $e_i : (op_i, delay_i, type_i, data_i, blobInfo_i)$, der wie in Kapitel 3.2.5.5 aufgebaut ist.
$execute(\tau)$ oder $executerestore_R(VS, O, TP)$ bezeichnet die Ausführung des durch τ repräsentieren *restore* Befehls. Für alle $e_i \in \tau$ wird zum Zeitpunkt $delay_i$ von S_{Restic} die Operation op_i für $data_i$ ausgeführt. Damit ist für S_{Backup} und damit auch einen Angreifer $execute(\tau)$ nicht unterscheidbar von der eigentlichen Ausführung des *restore* Befehls nach Modellierung 3.4.

3.3.5.5. Ablauf der abstrakten Modellierung

In der abstrakten Modellierung bekommt S_{Restic} ein virtuelles Verzeichnissystem VS dieses *restore* Befehls, die Werte O für die Optionen dieses *restore* Befehls und den realen Target-Pfad TP dieses *restore* Befehls übergeben. S_{Restic} emuliert ein System, das nur aus den Verzeichnissen und Dateien des virtuellen Verzeichnissystems VS besteht und führt daraufhin den *restore* Befehl auf diesem System aus. Der *restore* Befehl wird mit den Optionen O ausgeführt und erhält den Target-Pfad TP beim Start dieses Befehls. Die abstrakte Modellierung von Restics *restore* Befehl für ein Repository R ist in Diagramm 3.4 dargestellt. Da in dieser Masterarbeit ein *restore* Befehl in der Regel immer für dasselbe Repository R und den gleichen Benutzer ausgeführt wird, wird sowohl der Verbindungsaufbau zu R und die genaue Bestimmung des Keys K nicht modelliert.

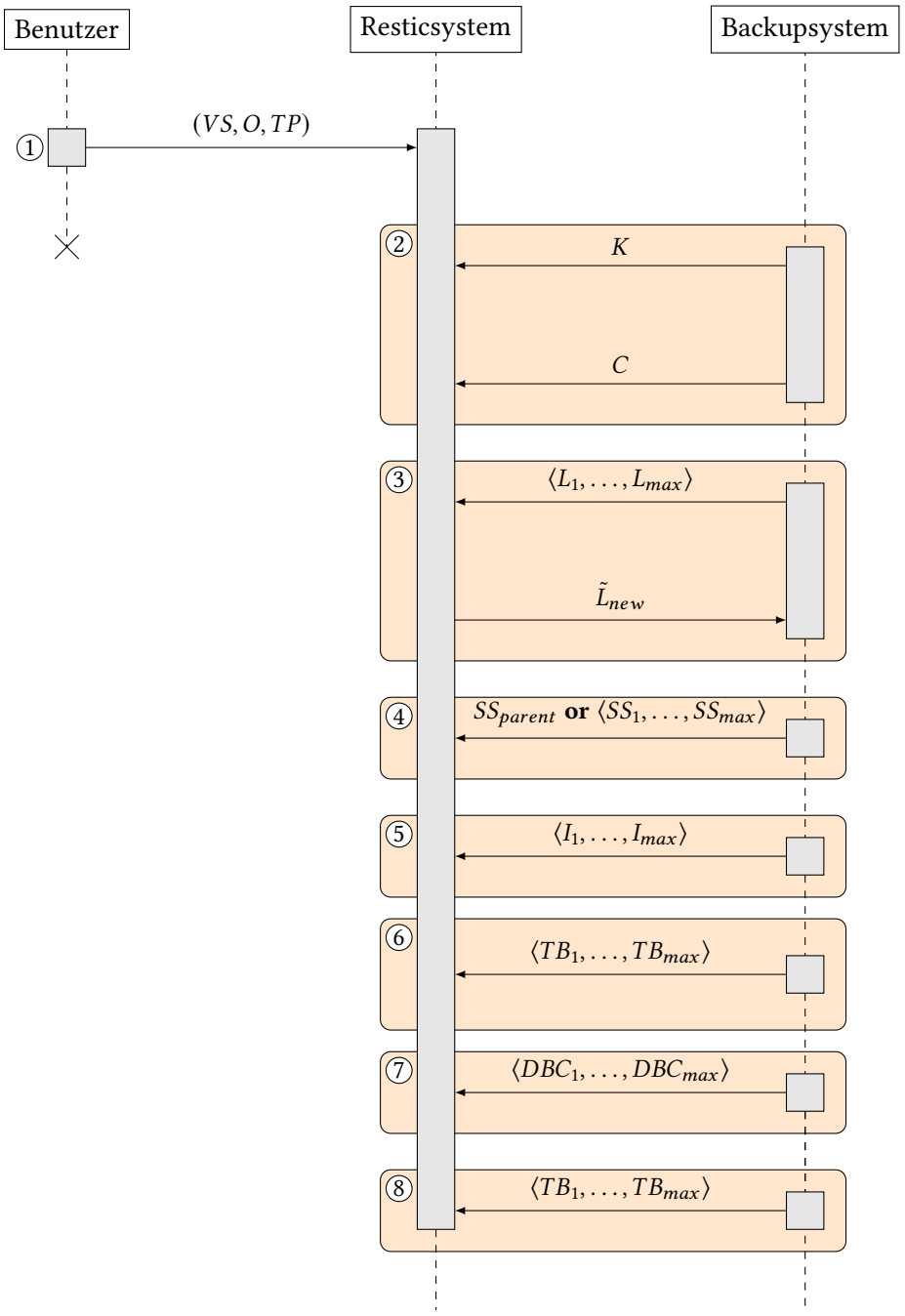

Figure 3.4.: Abstraktes Kommunikationsprotokoll des *restore* Befehls

1. Als Erstes übergibt der Benutzer das Tupel (VS, O, TP) an S_{Restic}. VS stellt dabei das virtuelle Verzeichnissystem dar, auf dem der *restore* Befehl aufgerufen wird und $O := (CONNECTIONS, SNAPSHOT\_ID, OVERWRITE)$ enthält, die Werte für die betrachteten Optionen dieses *restore* Befehl. TP ist der reale Target-Pfad, mit dem dieser *restore* Befehl aufgerufen wird.

2. Durch Funktion 19 erhält S_{Restic} die Key-Datenstruktur K für den Benutzer, der den *restore* Befehl ausführt und damit das Benutzerpasswort kennt. Außerdem erhält S_{Restic} die Config-Datei C des Repositorys R, für das der *restore* Befehl ausgeführt wird.
3. Die Funktion 21 fordert zunächst alle Lock-Dateien $\langle L_1, \ldots, L_{max} \rangle$ des Repositorys an und schickt dann selbst eine nicht exklusive Lock-Datei $\tilde{L}_{new}$ an S_{Backup}.
4. Enthält O ein gültiges Snapshot-ID-Präfix für $SNAPSHOT\_ID$, wird die Snapshot-Datei SS_{parent} von S_{Restic} angefordert und von S_{Backup} an S_{Restic} geschickt. Ein gültiges Snapshot-ID-Präfix ist ein ID-Präfix für einen Snapshot, dessen Target-Pfade den Target-Pfad TP enthalten. Enthält O keinen gültigen Wert für $SNAPSHOT\_ID$, werden alle Snapshot-Dateien $\langle SS_1, \ldots, SS_{max} \rangle$ an S_{Restic} gesendet. Restic bestimmt daraufhin selbst einen wiederherzustellenden Snapshot SS_{parent}, wie in Funktion 23 beschrieben ist.
5. Die Funktion 25 fordert alle Index-Dateien $\langle I_1, \ldots, I_{max} \rangle$ des Repositorys R von S_{Backup} an.
6. Der *restore* Befehl traversiert zunächst den kompletten Verzeichnisbaum $T_{restore}$ des Snapshots SS_{parent} in einer Tiefensuche. Dabei werden nach und nach die Tree-Blobs TB_1 bis TB_{max} von S_{Backup} angefordert. Bei dieser Traversierung werden zunächst nur die Data-Blobs bestimmt, die wiederhergestellt werden sollen. Allerdings werden diese Data-Blobs noch nicht von S_{Backup} angefordert.
7. Erst nachdem $T_{restore}$ vollständig traversiert wurde, werden Data-Blobs mit Funktion 4 gruppiert nach ihren Packs von S_{Backup} angefordert. Dazu wird die Funktion 29 verwendet, mit der mehrere Blobs eines Packs auf einmal von S_{Backup} angefordert werden. Dadurch erhält S_{Backup} nur noch die Startposition und Endposition eines Bytestrings aus einem Pack und erhält keine Informationen über die darin enthalten Data-Blobs. Diese zusammenhängend angeforderten Data-Blob werden als ein Data-Blob-Chunk (DBC) bezeichnet. Alle durch S_{Restic} von S_{Backup} angeforderten Data-Blob-Chunks $\langle DBC_i, \ldots, DBC_{max} \rangle$ werden an S_{Restic} geschickt.
8. Zum Schluss wird $T_{restore}$ nochmal traversiert und alle Tree-Blobs $\langle TB_i, \ldots, TB_{max} \rangle$ dieses Baums werden erneut von S_{Restic} angefordert. Dadurch stellt Restic Metadaten der einzelnen Dateien und Verzeichnisse wieder her.

3.4. Prune Befehl

Der *prune* Befehl von Restic kann für ein gegebenes Restic-Repository aufgerufen werden. Mit diesem Befehl können alte ungenutzte Daten aus dem Repository automatisiert durch Restic gelöscht werden und so die Größe des Repositorys verkleinert werden. Der *prune* Befehl entfernt ungenutzte Data-Blobs, Tree-Blob, Packs und Indices. Dabei orientiert sich der *prune* Befehl an den im Repository gespeicherten Snapshots und den Verzeichnisbäumen, die diese Snapshots mit ihren Root-Tree-Blobs aufspannen. Blobs, die in keinem einzigen Verzeichnisbaum der Snapshots des Repositorys verwendet werden, werden unbenutzt Blobs oder Unused-Blobs genannt.
Zuerst werden die möglichen Optionen vorgestellt, mit denen ein *prune* Befehl ausgeführt werden kann.

Daraufhin stellt dieses Kapitel zwei Modellierungen des *prune* Befehls als kryptografisches Protokoll vor. Zunächst wird die reale Modellierung vorgestellt, bei der der *prune* Befehl in mehrere detailliert beschriebene Blöcke unterteilt wird. Diese Blöcke sind unter anderem auch im Anhang beschrieben, falls sie in anderen Restic-Befehlen wiederverwendet werden (siehe Kapitel A.1).
Danach wird die abstrakte Modellierung des *prune* Befehls als kryptografisches Protokoll vorgestellt. Diese Modellierung wird für die später vorgestellten Sicherheitsspiele verwendet. Bei dieser Modellierung wird der Fokus auf den Nachrichtenaustausch zwischen S_{Restic} und S_{Backup} gelegt.

3.4.1. Prämisse

Es wird davon ausgegangen, dass bereits ein Repository auf S_{Backup} initialisiert wurde. Außerdem wird davon ausgegangen, dass mit dem *backup* Befehl mindestens ein Backup von Dateien auf S_{Restic} für dieses Repository erstellt wurde. Das bedeutet, dass mindestens eine Key-Datei, eine verschlüsselte und authentifizierte Config-Datei, sowie eine Snapshot-Datei und zugehörige Pack-Dateien und Index-Dateien in diesem Repository auf dem Backupsystem existieren.
Damit die volle Funktionalität des *prune* Befehls betrachtet werden kann, muss es möglich sein, dass Snapshots aus dem Repository entfernt werden, bevor der *prune* Befehl aufgerufen wird. Das kann in der Praxis mit dem *forget* Befehl von Restic erreicht werden, der einzelne Snapshots aus einem Repository löscht. Bei der abstrakten Modellierung wird jedoch eine andere Möglichkeit zum Löschen von Snapshots aus dem Repository vorgestellt, die für die restliche Masterarbeit als einzige Methode betrachtet wird.

3.4.2. Optionen des *prune* Befehls

Bei dem Aufruf des *prune* Befehls können verschiedene Optionen gesetzt oder konfiguriert werden, die den Ablauf des *prune* Befehls modifizieren. Diese Optionen sind in Tabelle 3.5 aufgelistet.

3.4.3. Restic-Funktionen für den *prune* Befehl

In diesem Kapitel werden Restic-Funktionen erklärt, die nur von dem *prune* Befehl verwendet werden. Diese Funktionen werden in einem späteren Kapitel verwendet, um den Ablauf des *prune* Befehls zu modellieren.

3.4.3.1. Alle Root-Tree-Blob-IDs aller Snapshots des Repositorys laden

Restic erstellt eine Liste, in der alle Snapshot-Datenstrukturen des Repositorys zwischengespeichert werden. S_{Restic} fordert die Metadaten aller Snapshot-Dateien dieses Repositorys

Name	Beschreibung
REPOSITORY_PATH	Ort, an dem das Repository liegt. Dies wird benötigt, um eine Verbindung zum S_{Backup} aufzubauen.
KEY_HINT	ID-Präfix einer Key-Datei des Repositorys. Zum gezielten Anfordern einer Key-Datei.
CONNECTIONS	Restic verwendet diesen Wert, um die Anzahl paralleler Prozesse zu bestimmen, die für einige Funktionen erstellt werden.
PACK_SIZE	Restic verwendet diesen Wert, um die angestrebte Größe eines erstellten Packs festzulegen. Ein Pack gilt erst als voll und wird gespeichert, wenn seine Größe größer ist als dieser Wert. Außerdem werden Packs, die kleiner sind als 4% dieses Wertes auch für das Repacking ausgewählt, auch wenn sie nur aus Used-Blobs bestehen. (Weitere nicht betrachtete Restic-Optionen können diese Grenze von 4% verändern)
MAX_UNUSED	Diese Option gibt die maximale Anzahl Bytes aller Packs an, die nach der Ausführung des *prune* Befehls von keinem der Snapshots des Repositorys benötigt werden. Ist der Wert dieser Option größer als 0, kann es dazu führen, dass Packs, die ungenutzt Daten enthalten nicht zum Repacking ausgewählt werden und unverändert im Repository gespeichert bleiben.
MAX_REPACK_SIZE	Diese Option gibt die maximale Anzahl Bytes aller Packs an, die durch Restic für das Repacking ausgewählt werden dürfen. Dabei zählt bei einem für das Repacking ausgewählte Pack jedes seiner Bytes für dieses Limit (auch die Unused-Blobs dieses Packs).
UNSAFE_RECOVER	Wenn diese Option auf *true* gesetzt ist, werden alle Indices des Repositorys zuerst gelöscht und dann neu erstellt. Damit kann auch auf einem System, das keinen oder nur sehr wenig freien Speicherplatz besitzt der *prune* Befehl ausgeführt werden. Ist diese Option nicht *true*, werden die Indices nur einer nach dem anderen aktualisiert, was zu einem temporären Speicheroverhead führen kann.

Table 3.5.: Optionen des *prune* Befehls

von S_{Backup} an.
Restic startet so viele parallele Prozesse, wie die *connections* Option angibt. Auf die parallelen Prozesse werden die IDs einzelner Snapshots nach und nach verteilt mit `dispatch(m.id)`. Jeder Prozess, der die ID eines Snapshots des Repositorys erhält, startet seine Ausführung mit der Funktion 7.
Jeder parallele Prozess fordert den Snapshot, der zu der ID gehört, bei S_{Backup} an. S_{Backup} schickt den Inhalt der Snapshot-Datei für diese ID an S_{Restic} zurück. Restic verifiziert, entschlüsselt und dekomprimiert die erhaltene Snapshot-Datenstruktur. Die parallelen Prozesse legen eine Liste mit allen geladenen Snapshots an.

Nachdem alle Snapshot-Datenstrukturen in einer Liste zusammengefasst wurden, iteriert Restic über diese Liste. Die Root-Tree-Blob-IDs alle Snapshots werden in einer Liste `rootTreeBlobIds` gespeichert (siehe Tabelle 3.1). Die Liste der Root-Tree-Blob-IDs wird daraufhin zurückgegeben.

Funktion 6 Loads Root-Tree-Blobs-IDs of all Snapshots of the Repository

```
Require: CONNECTIONS
  function loadRootTreeIDs
      List_Snapshot allSnapshots := {}
      send(TYPE_SNAPSHOT)                          ▷ S_Restic ⟶ S_Backup
      Metadata metadatas ← receive()               ▷ S_Restic ⟵ S_Backup
      startParallelProcesses(CONNECTIONS)
      for all m in metadatas do
          dispatch(m.id)
      end for
      waitBarrier()
      List_Hexstring[64] rootTreeBlobIds := {}
      for all SS_i in allSnapshots do
          rootTreeBlobIds := rootTreeBlobIds||SS_i.root_tree
      end for
      return rootTreeBlobIds
  end function
```

Funktion 7 Start procedure for loading snapshot

```
Require: Shared variable: allSnapshots
  procedure start(id)
      send(id, TYPE_SNAPSHOT)                      ▷ S_Restic ⟶ S_Backup
      Bytestring encryptedSnapshot := receive()    ▷ S_Restic ⟵ S_Backup
      if id(encryptedSnapshot[size_IV : size_MAC]) == id then
          Snapshot SS_i := decompress(decrypt(encryptedSnapshot))
          if SS_i is not error then
              allSnapshots := allSnapshots||SS_i
              return
          end if
      end if
      return error
  end procedure
```

3.4.3.2. Sammeln aller Tree-Blobs und Data-Blobs

Die Funktion 8 legt die Liste `used_blobs` in der internen Repository-Datenstruktur an. Zu der Liste `used_blobs` werden die IDs alle Tree-Blobs und Data-Blobs hinzugefügt, die Teil der Verzeichnisbäume sind, die zu den übergebenen Root-Tree-Blob-IDs gehören. Dabei ist die

Reihenfolge, in der die Blobs zu der Liste `used_blobs` hinzugefügt werden im Allgemeinen nichtdeterministisch.
Zunächst traversiert Restic die Verzeichnisbäume der einzelnen Snapshots SS_i und sammelt die Restic-IDs alle Tree-Blobs und Data-Blobs, die Teil der Backups der Snapshots SS_i sind, in der Liste `used_blobs`. Dazu werden von Restic parallele Prozesse gestartet, die dafür verantwortlich sind Tree-Blobs TB_i aus dem Repository zu laden. Die Anzahl der gestarteten Prozesse wird anhand der vorhandenen logischen Kerne der CPU und dem Wert der *connections* Option bestimmt. Tree-Blob-IDs werden an diese Prozesse mit `dispatch(nextId)` verteilt. Außerdem wird ein weiterer paralleler Prozess gestartet, der das selbe macht, wie die anderen parallelen Prozesse aber für besonders große Tree-Blobs zuständig ist. An diesen Prozess wird eine Tree-Blob-ID mit `dispatch_huge(nextId)` verteilt.
Die Funktion mit der alle parallelen Prozesse starten ist die Funktion 9. Diese Funktion lädt einen Tree-Blob TB_i mit der Funktion 28 `loadBlob(treeBlobId)` und fügt den geladenen Blob am Ende einer Liste `loadedTreeBlobs` hinzu. Die Liste `loadedTreeBlobs` wird sich mit allen anderen parallelen Prozessen und der Funktion 8 geteilt.
Bevor die IDs aller Blobs in der Liste `used_blobs` gesammelt werden legt Restic eine Variable `nextTreeBlob` für einen zu verarbeitenden Tree-Blob an. Außerdem wird die Liste `subTreeBlobIds` mit den Root-Tree-Blob-IDs der Snapshots, die dieser Funktion übergeben werden, initialisiert. Nun führt Restic eine der drei folgenden Aktionen nichtdeterministisch solange aus, bis jeder Tree-Blob alle Verzeichnisbäume der Root-Tree-Blobs verarbeitet wurde:

- Wenn ein `subTreeBlobIds` eine Tree-Blob-ID enthält, wird die erste Tree-Blob-ID ausgewählt. Falls die Tree-Blob-ID nicht in `repo.used_blobs` vorkommt, wird sie zu `repo.used_blobs` hinzugefügt und einem der gestarteten parallelen Prozesse übergeben.
- Wurde ein Tree-Blob von einem der parallelen Prozesse geladen und zur Verarbeitung ausgewählt, ist die entschlüsselte Tree-Blob-Datenstruktur in der Variablen `nextTreeBlob` gespeichert. Enthält `nextTreeBlob` eine Tree-Blob-Datenstruktur, wird für jeden Node-Eintrag, der eine Datei repräsentiert Folgendes durchgeführt. Alle IDs von Data-Blobs, aus denen sich die Datei zusammensetzt, werden zu der Liste `repo.used_blobs` hinzugefügt. Am Ende wird der verarbeitete Tree-Blob aus `nextTreeBlob` entfernt.
- Wurde noch kein Tree-Blob zur Verarbeitung ausgewählt und wurden Tree-Blobs durch die parallelen Prozesse geladen, wird der erste Tree-Blob aus der Liste `loadedTreeBlobs` der geladenen Tree-Blobs entfernt. Der entfernte Tree-Blob wird für die Verarbeitung ausgewählt, indem er in der Variable `nextTreeBlob` gespeichert wird. Daraufhin werden die IDs aller Tree-Blobs, die Unterverzeichnisse des ausgewählten Tree-Blobs repräsentieren zu `subTreeBlobIds` hinzugefügt.

Durch die Überprüfung der Variable `nextTreeBlob` in der zweiten und dritten Aktion werden die Aktionen zwei und drei immer abwechselnd ausgeführt.

3.4.3.3. Erstellen eines Plans für die Neubildung der Packs eines Repositorys

Bevor der *prune* Befehl neue Pack-Datenstrukturen bildet und alte Pack-Datenstrukturen löscht, wird mit der Funktion 10 ein Prune-Plan erstellt. Dieser Prune-Plan teilt Packs in

Funktion 8 Creates a list of all Blobs, which are part of rootTreeBlob Trees

```
Require: CONNECTIONS
  procedure collectAllBlobs(rootTreeBlobIds)
    startParallelProcesses(CONNECTIONS + getNumberBasedOnCPU())
    startParallelProcesses(1)
    List_Tree−Blob loadedTreeBlobs := {}
    Tree − Blob nextTreeBlob := nil
    repo.used_blobs := {}
    List_Hexstring[64] subTreeBlobIds := rootTreeBlobIds
    loop
      rn ←$ getRandomNumber(1, 3)
      if rn == 1 and size(subTreeBlobIds) > 0 then
        nextId := popFirst(subTreeBlobIds)
        if nextId is not in repo.used_blobs then
          repo.used_blobs := repo.used_blobs||(nextId, 1)
          if getBlobSize(nextId) > 50MiByte then
            dispatch_huge(nextId)
          else
            dispatch(nextId)
          end if
        end if
      else if rn == 2 or nextTreeBlob is defined then
        for all Node node in getFileNodes(nextTreeBlob.nodes) do
          for all dataBlobId in node.content do
            repo.used_blobs := repo.used_blobs||(dataBlobId, 1)
          end for
        end for
        nextTreeBlob := nil
      else if rn == 3 or nextTreeBlob is not definied then
        if size(loadedTreeBlobs) > 0 then
          nextTreeBlob := popFirst(loadedTreeBlobs)
          List_Hexstring[64] newIds := {}
          for all Node node in getSubtreeNodes(nextTreeBlob.nodes) do
            newIds := newIds||node.subtree
          end for
          subTreeBlobIds := newIds||subTreeBlobIds
        end if
      end if
    end loop
  end procedure
```

Funktion 9 Start procedure for loading a Tree-Blob

Require: Shared variable: $loadedTreeBlobs$
procedure START($treeBlobId$)
$Tree - BlobTB_i$:= LOADBLOB($treeBlobId$)
$loadedTreeBlobs := loadedTreeBlobs||TB_i$
end procedure

verschiedene Listen ein, die angeben, ob ein Pack gelöscht wird oder neugebildet wird (siehe Tabelle 3.2).
Als Erstes werden alle Blobs-IDs aus `repo.used_blobs` mit `assignBlobsToPacks()` einem Pack des Repositorys, in dem der Blob vorkommt, zugeordnet. Kommt ein Blob in mehreren Packs vor, erfolgt die Zuordnung nichtdeterministisch zu einem dieser Packs mit der folgenden Zuordnungsstrategie. Restic iteriert nichtdeterministisch über alle Packs, die diesen Blob enthalten und wählt das Erste Pack für das Folgendes gilt. Dem Pack muss entweder bereits mindestens ein Blob aus `repo.used_blobs` zugeordnet worden sein, das Pack darf nur aus mehrfach vorkommenden Blobs bestehen oder es ist das letzte Pack, das Restic für diesen Blob betrachtet. Ein Blob wird als mehrfach vorkommend bezeichnet, wenn dieser Blob in mehreren Packs desselben Repositorys gespeichert ist. Ein Blob, der in einem Pack vorkommt, dem der Blob nicht zugeordnet wurde, wird auch als Unused-Blob für dieses Pack bezeichnet. Die Funktion `assignBlobsToPacks()` gibt eine Liste von IDs aller Packs zurück, die in dem Masterindex des Repositorys vorkommen.
Restic lässt sich die Metadaten aller Pack-Datenstrukturen des Repositorys von S_{Backup} schicken und iteriert über diese Metadaten. Die Funktion `getUsedBlobs(packId)` bestimmt die Blobs aus dem Pack mit der ID *packId*, die diesem Pack zugeordnet wurden und gibt die Anzahl dieser Blobs zurück. Packs, die nicht im Masterindex vorkommen, werden zu der `removePacksFirst` Liste hinzugefügt. Packs, denen kein Blob zugeordnet wurde, werden zu der `removePacks` Liste hinzugefügt. Packs, die nur aus zugeordneten Blobs bestehen und kleiner sind als $\frac{PACK\_SIZE}{25}$, werden zu der `repackSmallCandidates` Liste hinzugefügt. Packs, die nicht nur aus zugeordneten Blobs bestehen, werden zu der `repackPacks` Liste hinzugefügt. Alle Packs, die im Masterindex vorkommen, aber sich nicht unter den empfangenen Metadaten befunden haben, werden zu der Liste `ignorePacks` hinzugefügt. Enthält eins der Packs aus `ignorePacks` einen zugeordneten Blob, kommt es zu einem Fehler und der *prune* Befehl wird beendet.
Gibt es mindestens zehn Packs in `repackSmallCandidates` werden diese Pack-IDs ebenfalls zu der Liste `repackPacks` hinzugefügt.
Restic sortiert die Liste `repackPacks` der Packs, die für ein Repacking ausgewählt wurden. Dabei werden Packs, die Tree-Blobs enthalten, vor Packs, die Data-Blobs enthalten, sortiert. Innerhalb der Tree-Blob-Packs und Data-Blob-Packs werden Packs, die kleiner sind als $\frac{PACK\_SIZE}{25}$, vor alle anderen Packs sortiert. Ansonsten werden die Packs aufsteigend nach ihrem Anteil von Blobs, die ihnen nicht zugeordnet wurden, zu Blobs, die ihnen zugeordnet wurden, sortiert ($\frac{unusedBlobs_{Pack}}{usedBlobs_{Pack}}$).
Mit der Funktion `onlyKeepNecessaryPacks()` werden basierend auf den Optionen *max-unused* und *max-repack-size* Packs aus der Liste `repackPacks` entfernt. Die Liste `repackPacks` enthält nach der Ausführung dieser Funktion nur noch die Pack-IDs für Packs, die

die Werte der *max-unused* Option und *max-repack-size* Option nicht überschreiten. Die restlichen Pack-IDs werden aus der Liste entfernt. Die Optionen *max-unused* und *max-repack-size* sind in Tabelle 3.5 erklärt.
Am Ende wird der Prune-Plan zurückgegeben.

Funktion 10 Creates list of packs, which should be removed or repacked

```
Require: PACK_SIZE
  function createPrunePlan
    Prune − Plan plan
    List_Hexstring[64] packIds := assignBlobsToPacks()()
    send(TYPE_PACK)                                   ▷ S_Restic ⟶ S_Backup
    List_Metadata metadatas := receive()              ▷ S_Restic ⟵ S_Backup
    for all m in metadatas do
      if not m.id in packIds then
        plan.removePacksFirst := plan.removePacksFirst||m.id
      else if getUsedBlobs(m.id) == 0 then
        plan.removePacks := plan.removePacks||m.id
      else if getUsedBlobs(m.id) == amountOfBlobs(m.id) then
        if m.size < PACK_SIZE/25 then
          plan.repackSmallCandidates := plan.repackSmallCandidates||m.id
        end if
      else
        plan.repackPacks := plan.repackPacks||m.id
      end if
      remove(packIds, m.id)
    end for
    plan.ignorePacks := packIds
    for all ids in packIds do
      if getUsedBlobs(ids) > 0 then
        return error
      end if
    end for
    if size(plan.repackSmallCandidates) >= 10 then
      plan.repackPacks := plan.repackPacks||plan.repackSmallCandidates
    end if
    sortRepackCandidates(plan.repackPacks)
    onlyKeepNecessaryPacks(plan.repackPacks)
    return plan
  end function
```

3.4.3.4. Ausführen des Prune-Plans

Das Ausführen des Prune-Plans löscht alle Packs, die in dem Prune-Plan zum Löschen oder Repacking vorgesehen sind. Alle Blobs in diesen gelöschten Packs, die weiterhin von

den existierenden Snapshots referenziert werden, werden in neuen Pack-Datenstrukturen hinzugefügt und zum Speichern an S_{Backup} geschickt. Am Ende werden die Indices des Repositorys für die neuen Packs angepasst. Das Ausführen eines Prune-Plans wird mit der Funktion 11 beschrieben und stellt den letzten Schritt des *prune* Befehls dar.
Bevor die Funktion 11 ausgeführt wird, startet Restic im Hintergrund *CONNECTIONS* viele Pack-Uploader-Prozesse (siehe Pack-Uploader-Start-Funktion 27).
Die Funktion selbst startet als erstes *CONNECTIONS* viele parallele Remove-Prozesse. Ein Remove-Prozess startet seine Ausführung mit der Funktion 12. Ein Remove-Prozess sendet, die ihm übergebene ID an S_{Backup}, welches die zugehörige Datei im Repository löscht.
Danach werden $CONNECTIONS-1$ viele parallele Repack-Prozesse gestartet. Jeder Repack-Prozess startet seine Ausführung mit der Funktion 29, mit der mehrere Blobs aus einem Pack von S_{Backup} geladen werden können. Dazu wird jedem Repack-Prozess eine Pack-ID und ein Liste an Blob-IDs übergeben. Für jeden Blob aus der Liste an Blob-IDs wird die Funktion 13 aufgerufen, die diesen Blob in einem Pack speichert. Die Funktion 13 entspricht der `execute()` Funktion, die die Funktion 29 benötigt. Für das Speichern eines Blobs wird die Funktion 31 verwendet, die bei einem vollen Pack dieses Pack in einem Index speichert.
Restic verteilt an die Remove-Prozesse zuerst die Pack-IDs aus der Liste `plan.removePacksFirst` mit `dispatch_remove(id)`.
Als nächstes bestimmt Restic für alle Packs deren IDs in der Liste `plan.repackPacks` liegen, welche Blobs aus `repo.used_blobs` diesen Packs zugeordnet wurden. Denn nur diese zugeordneten Blobs werden auch weiterhin von den Snapshots des Repositorys benötigt und müssen in einem neuen Pack gespeichert werden. Die IDs dieser Blobs werden zusammen mit der Pack-ID an die Repack-Prozesse verteilt mit `dispatch_repack(packId, blobIds)`.
Danach wartet Restic bis alle Remove-Prozesse und Repack-Prozesse beendet wurden. Eventuell besitzt Restic noch neu erstellte Packs oder Indices durch die Pack-Uploader-Prozesse, die nicht auf S_{Backup} gespeichert wurden. Daher werden die übrigen Packs und Indices, die noch nicht an S_{Backup} geschickt wurden, verschlüsselt, authentifiziert und an S_{Backup} geschickt mit `flushPacks()` und `flushIndices()`.
Restic fügt alle Pack-IDs aus `plan.repackPacks` zu der Liste `plan.removePacks` hinzu. Alle Pack-IDs aus der neuen `plan.removePacks` Liste werden auch zu der `plan.ignorePacks` Liste hinzugefügt.
Wurde die *unsafe-recover* Option benutzt erstellt Restic *CONNECTIONS* viele parallele Remove-Prozesse und verteilt die IDs aller Indices an die Remove-Prozesse mit `dispatch_remove(id(index))`. Damit werden alle Index-Dateien des Repositorys gelöscht. Restic wartet bis alle Remove-Prozesse beendet wurden.
Wurde die *unsafe-recover* Option nicht benutzt, schreibt Restic die Indices des Repositorys mit der Funktion `rewriteAllIndexes()` um, anstatt sie zu löschen und später neu zu erstellen. Dazu wird jeder Index, der ein Pack enthält, dass nicht in `plan.ignorePacks` ist, von S_{Backup} angefordert. Die Einträge des Index, die zu Packs gehören, die nicht in `plan.ignorePacks` vorkommen, werden in einem neuen Index gespeichert und der alte Index wird aus dem Repository gelöscht.
Um alle veralteten Packs aus dem Repository zu löschen, startet Restic *CONNECTIONS* viele Remove-Prozesse. An jeden dieser Remove-Prozesse wird eine Pack-ID aus der Liste `plan.ignorePacks` verteilt mit `dispatch_remove(id)`. Restic wartet bis alle Remove-Prozesse beendet wurden.

Falls die *unsafe-recover* Option verwendet wurde, müssen alle Indices neu erstellt werden. Dazu itteriert Restic über alle Pack-IDs, die nicht Teil der Liste `plan.ignorePacks` sind und fügt dieses Pack und die Informationen über seine enthaltenen Blobs zu einer neuen Index-Datenstruktur hinzu. Diese Index-Datenstruktur wird wie in der Funktion 27 erstellt, als *final* markiert und an S_{Backup} gesendet.
Eventuell besitzt Restic nach dem Umschreiben oder Neuerstellten der Indices noch Indices, die nicht auf S_{Backup} gespeichert wurden, weil sie noch nicht die Bedingungen zum speichern erfüllt haben. Daher werden die übrigen Indices, die noch nicht an S_{Backup} geschickt wurden, verschlüsselt, authentifiziert und an S_{Backup} geschickt mit `flushIndices()`.

3.4.4. Ablauf der realen Modellierung des *prune* Befehls

Dieses Kapitel beschreibt den Ablauf nach dem Aufruf des Restic *prune* Befehls.
Restic sucht das vom Benutzer angegebene Repository auf S_{Backup}, baut eine Verbindung zu dem Repository auf und baut eine interne Repository-Datenstruktur auf mit Funktion 19.
Bevor Restic weiter mit dem Repository auf S_{Backup} interagiert, wird eine exklusive Lock-Datei mit Funktion 21 dem Repository hinzugefügt. Dadurch wird verhindert, dass mögliche andere Restic-Distributionen auf das Repository zugreifen können.
Nun lässt sich Restic mit Funktion 25 alle Indices des Repositorys von S_{Backup} schicken und speichert diese in seiner internen Repository-Datenstruktur.
Restic lässt sich alle Snapshots des Repositorys von S_{Backup} mit Funktion 6 schicken und speichert die Root-Tree-Blob-IDs dieser Snapshots in `rootTreeBlobIds`.
Danach wird ein Verzeichnisbaum für jeden Root-Tree-Blob aufgebaut und traversiert mit der Funktion 8. Dazu wird jeder Tree-Blob der Verzeichnisbäume der Root-Tree-Blobs in nichtdeterministischer Reihenfolge geladen. Dabei werden ebenfalls mit Funktion 8 alle IDs der traversierten Tree-Blobs und alle IDs der in diesen Tree-Blobs referenzierten Data-Blobs in einer Liste `repo.used_blobs` gesammelt. Nach der Ausführung der Funktion 8 enthält `repo.used_blobs` die IDs aller Tree-Blobs und Data-Blobs, die zur vollständigen Darstellung der Verzeichnisbäume aller Snapshots gehören.
Restic erstellt einen Prune-Plan mit der Funktion 10. Dieser Prune-Plan teilt Packs in verschiedene Listen ein, die angeben, ob ein Pack nur gelöscht werden soll oder ob ein Pack neugebildet werden soll (siehe Tabelle 3.2).
Als Letztes wird der erstellte Prune-Plan mit der Funktion 11 ausgeführt. Die Blobs, die von mindestens einem Verzeichnisbaum der Snapshots des Repositorys verwendet werden, werden aus den ausgewählten Packs in neue Pack-Datenstrukturen gepackt und die alten Packs werden gelöscht. Die Indices werden ebenfalls umgeschrieben und veraltete Einträge gelöscht. Wenn die Option *unsafe-recover* des *prune* Befehls verwendet wird, werden sogar erst alle Indices gelöscht und komplett neue für das gesamte Repository angelegt und an S_{Backup} zum Speichern geschickt.

Funktion 11 executePrunePlan

```
Require: CONNECTIONS, UNSAFE_RECOVER
  procedure startPrunePlan(Prune − Plan plan)
    startParallelProcesses(CONNECTIONS)
    startParallelProcesses(CONNECTIONS − 1)
    for all id in plan.removePacksFirst do
      dispatch_remove(id, TYPE_PACK)
    end for
    for all packId in plan.repackPacks do
      List_Hexstring[64] blobIds := getUsedBlobsFromPack(packId, repo.used_blobs)
      dispatch_repack(packId, blobIds)
    end for
    waitBarrier()
    flushPacks
    flushIndices
    repo.removePacks := repo.removePacks||repo.repackPacks
    repo.ignorePacks := repo.ignorePacks||repo.removePacks
    if UNSAFE_RECOVER is defined then
      startParallelProcesses(CONNECTIONS)
      for all Index index in repo.masterindex do
        dispatch_remove(id(index), TYPE_INDEX)
      end for
      waitBarrier()
    else
      rewriteAllIndexes
    end if
    startParallelProcesses(CONNECTIONS)
    for all id in plan.removePacks do
      dispatch_remove(id, TYPE_PACK)
    end for
    waitBarrier()
    if UNSAFE_RECOVER is defined then
      for all packId in getAllPackIds do
        if not packId in repo.ignorePacks then
          saveInIndex(packId)
        end if
      end for
      flushIndices
    end if
  end procedure
```

Funktion 12 Removes a data structure from the repository

```
procedure START(id, type)
    DELETE(id, type)                                  ▷ S_Restic ⟶ S_Backup
end procedure
```

Funktion 13 Repacks all given Blob-IDs of a given pack

```
procedure EXECUTE_REPACK(blob)
    if ID(blob) in repo.used_blobs then
        REMOVE(repo.used_blobs, ID(blob))
        SAVEBLOB(blob, true)
    end if
end procedure
```

3.4.5. Abstrakte Modellierung des *prune* Befehls

Dieses Kapitel stellt die abstrakte Modellierung des *prune* Befehls vor. Diese Modellierung ist möglichst einheitlich zwischen den drei betrachteten Befehlen, um diese einheitlicher in die vorgestellten Sicherheitsspiele einbinden zu können. Dazu wird für die abstrakte Modellierung der Aufruf des *prune* Befehls zu einem Tupel abstrahiert. Dieses Tupel erhält im Gegensatz zu dem betrachteten *backup* Befehl und *restore* Befehl kein virtuelles Verzeichnissystem oder Target-Pfade, da der *prune* Befehl auf keinem Verzeichnissystem ausgeführt wird.
Anstelle den *forget* Befehl von Restic zu benutzen, um vor der Ausführung des *prune* Befehls Snapshots aus dem Repository zu löschen, wird für die abstrakte Modellierung eine abstraktere Version betrachtet. Als Teil der abstrakten Darstellung des *prune* Befehls als Tupel wird eine Liste von Snapshot-IDs *SID* übergeben. Bevor die reguläre Ausführung des *prune* Befehls beginnt, werden zunächst alle Snapshot-Datenstrukturen, deren ID in der Liste *SID* vorhanden ist aus dem Repository gelöscht.

3.4.5.1. Modellierte Optionen des *prune* Befehls

Es ist nicht nötig alle Optionen aus Tabelle 3.5 für den *prune* Befehl in dem abstrakt modellierten Protokoll abzubilden. Für die abstrakte Modellierung wird davon ausgegangen, dass der *prune* Befehl auf einem bekannten Repository von einem bekannten Benutzer ausgeführt wird. Daher sind die *repository_path* und *key_hint* Optionen für dieses Protokoll überflüssig. Die übrigen Optionswerte, mit denen ein *prune* Befehl durchgeführt wird, werden in einem Tupel

$$O := (CONNECTIONS, PACK\_SIZE, MAX\_UNUSED, MAX\_REPACK\_SIZE, UNSAFE\_RECOVER)$$

zusammengefasst.

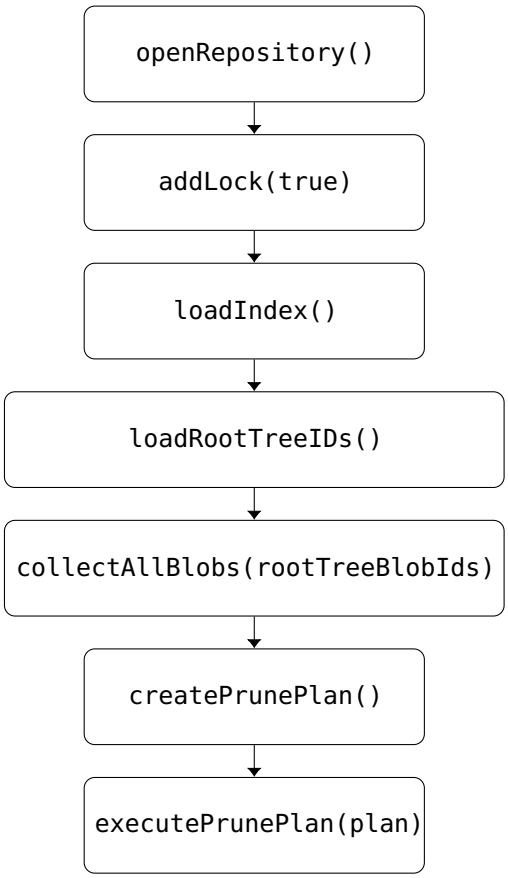

Figure 3.5.: Ablauf von Restics *prune* Befehl

3.4.5.2. Modellierter Nachrichtenaustausch

Genau, wie für den *backup* Befehl wird die Übergabe des Benutzerpassworts von dem Benutzer an Restic nicht modelliert und es wird davon ausgegangen, dass Restic das Benutzerpasswort bereits kennt. Außerdem werden ausgetauschte Metadaten auch für den *prune* Befehl nicht modelliert. Falls die Anfrage der Metadaten von Dateien aus dem Repository für einen manipulierenden Angreifer relevant wird, wird darauf explizit Bezug genommen. Ebenfalls werden für die bessere Übersichtlichkeit Anfragen von S_{Restic} für bestimmte Dateien oder Blobs nicht als Nachrichtenaustausch modelliert. Es wird nur das Senden der entsprechenden Datenstrukturen von S_{Backup} an S_{Restic} zum entsprechenden Zeitpunkt modelliert.

3.4.5.3. Formale Betrachtung des *prune* Befehls

Der *prune* Befehl kann ebenfalls wie der *backup* Befehl aus Kapitel 3.2.5.5 nur mit den ausgetauschten Datenstrukturen ohne Informationsverlust dargestellt werden. Mit der Notation $prune_R(SID, O)$ wird die abstrakte Modellierung des *prune* Befehls bezüglich einem

Repository R, mit den Optionen O und den zu entfernenden Snapshots SID. Daraus resultiert ebenfalls ein Event-Tupel $\tau := prune_R(SID, O)$, das alle Informationen über den Nachrichtenaustausch enthält, der bei der Ausführung des *prune* Befehls entsteht. Dieses Tupel τ besteht wie das durch den *backup* Befehl entstehende Tupel ebenfalls aus mehreren Events $\tau := (e_1, \ldots, e_{max})$. Für jeder zwischen S_{Restic} und S_{Backup} ausgetauschte Restic-Datenstruktur besitzt τ einen Eintrag $e_i : (op_i, delay_i, type_i, data_i, blobInfo_i)$, der wie in Kapitel 3.2.5.5 aufgebaut ist.
$execute(\tau)$ oder $executeprune_R(SID, O)$ bezeichnet die Ausführung des durch τ repräsentieren *prune* Befehls. Für alle $e_i \in \tau$ wird zum Zeitpunkt $delay_i$ von S_{Restic} die Operation op_i für $data_i$ ausgeführt. Damit ist für S_{Backup} und damit auch einen Angreifer $execute(\tau)$ nicht unterscheidbar von der eigentlichen Ausführung des *prune* Befehls nach Modellierung 3.6.

3.4.5.4. Ablauf der abstrakten Modellierung

In der abstrakten Modellierung bekommt S_{Restic} eine Liste SID von zu löschenden Snapshots bestehend aus deren Restic-IDs und die Werte O für die Optionen dieses *prune* Befehls übergeben. S_{Restic} entfernt zunächst alle Snapshots der Liste SID aus dem Repository, indem die IDs aus SID an S_{Backup} geschickt werden mit der Aufforderung diese Snapshot-Datei zu löschen. Dann wird der *prune* Befehl mit den Optionen O ausgeführt. Die abstrakte Modellierung von Restics *prune* Befehl für ein Repository R ist in Diagramm 3.6 dargestellt. Da in dieser Masterarbeit ein *prune* Befehl in der Regel immer für dasselbe Repository R und den gleichen Benutzer ausgeführt wird, wird sowohl der Verbindungsaufbau zu R und die genaue Bestimmung des Keys K nicht modelliert.

1. Als Erstes übergibt der Benutzer das Tupel (SID, O) an S_{Restic}. SID stellt dabei die Liste von Snapshots-IDs dar, die zunächst aus dem Repository entfernt werden müssen, bevor die eigentliche Ausführung des *prune* Befehls startet. $O := (CONNECTIONS, PACK\_SIZE, MAX\_UNUSED, MAX\_REPACK\_SIZE, UNSAFE\_RECOVER)$ enthält, die Werte für die betrachteten Optionen dieses *prune* Befehl.
2. Zunächst werden nichtdeterministisch alle IDs $SSID_i$ aus der Liste SID an S_{Backup} gesendet, mit der Aufforderung die Snapshots mit diesen IDs zu löschen.
3. Durch Funktion 19 erhält S_{Restic} die Key-Datenstruktur K für den Benutzer, der den *prune* Befehl ausführt und damit das Benutzerpasswort kennt. Außerdem erhält S_{Restic} die Config-Datei C des Repositorys R, für das der *prune* Befehl ausgeführt wird.
4. Die Funktion 21 fordert zunächst alle Lock-Dateien $\langle L_1, \ldots, L_{max} \rangle$ des Repositorys an und schickt dann selbst eine exklusive Lock-Datei $\tilde{L}_{new}$ an S_{Backup}.
5. Die Funktion 25 fordert alle Index-Dateien $\langle I_{1,1}, \ldots, I_{1,max} \rangle$ des Repositorys R von S_{Backup} an.
6. Die Funktion 6 fordert nichtdeterministisch alle Snapshot-Dateien $\langle SS_1, \ldots, SS_{max} \rangle$ des Repositorys R von S_{Backup} an und speichert die IDs der Root-Tree-Blobs in einer Liste.
7. Unter Verwendung der erzeugten Root-Tree-Blob Liste werden alle Verzeichnisbäume, die durch die Root-Tree-Blobs aus dieser Liste aufgespannt werden, traversiert. Dazu

werden nichtdeterministisch alle Tree-Blobs T_i aller Verzeichnisbäume der Root-Tree-Blobs mit Funktion 8 geladen. Dabei werden die IDs aller Tree-Blobs der Verzeichnisbäume und aller Data-Blobs, die in den Tree-Blobs der Verzeichnisbäume vorkommen, als Used-Blobs in der Liste `repo.used_blobs` gespeichert.

8. Basierend auf `repo.used_blobs` wird ein Prune-Plan mit der Funktion 10 erstellt (siehe Tabelle 3.2). Die Erstellung des Prune-Plans führt zu keinem Datenaustausch zwischen S_{Restic} und S_{Backup}. Danach wird jedoch dieser Prune-Plan mit der Funktion 11 ausgeführt, was zu einem Datenaustausch führt.
   Zuerst sendet S_{Restic} alle Pack-IDs $PID_{1,i}$ aus der Prune-Plan Liste `removePacksFirst` an S_{Backup} mit der Aufforderung diese Pack-Dateien aus R zu löschen. Im Laufe der Prune-Plan-Ausführung durch Funktion 11 werden Packs deren ID in der Prune-Plan Liste `repackPacks` vorhanden ist neu erstellt. Dazu werden Blobs, die für eins der Packs aus `repackPacks` ausgewählt wurden durch die Funktion 29 als Blob Chunks $BC := \{BC_1, \ldots, BC_{max}\}$ von S_{Backup} angefordert. Die ausgewählten Blobs werden in neuen Packs $\tilde{P} := \{\tilde{P}_1, \ldots, \tilde{P}_{max}\}$ gespeichert, die an S_{Backup} zum Speichern gesendet werden. Für die neuen Packs werden ebenfalls neue Indices $\tilde{I}_1 := \{\tilde{I}_{1,1}, \ldots, \tilde{I}_{1,max}\}$ erstellt und an S_{Backup} zum Speichern gesendet. Die Blob-Chunks werden in nichtdeterministischer Reihenfolge angefordert und zu den neu erstellten Packs hinzugefügt. Ebenso werden die Packs und Indices in eigenen parallelen Prozessen an S_{Backup} geschickt, wodurch auch die Sendereihenfolge der Packs und Indices und die Empfangsreihenfolge der Tree-Blobs nichtdeterministisch ist. Daher wird dieser Nachrichtenaustausch mit $\tau_{repacking} := Repacking(BC \cup \tilde{P} \cup \tilde{I}_1)$ abgebildet, wobei $\tau_{repacking}$ das Teiltupel von $\tau := prune_R(SID, O)$ ist, das nur die Datenstrukturen aus den Mengen BC, $\tilde{P}$ und $\tilde{I}_1$ enthält.
   Als nächstes sendet S_{Restic} alle Pack-IDs $PID_{2,i}$ aus der Prune-Plan Liste `removePacks` an S_{Backup} mit der Aufforderung diese Pack-Dateien aus R zu löschen.
   Zum Schluss werden abhängig von der *unsafe-recover* Option bestimmte Indices $I_2 := \{I_{2,1}, \ldots, I_{2,max}\}$ des Repositorys erneut angefordert. Außerdem werden die Index-IDs $IID := IID_1, \ldots, IID_{max}$ von veralteten Indices an S_{Backup} gesendet mit der Aufforderung diese Index-Dateien zu löschen. Abhängig von der *unsafe-recover* Option werden veraltete Indices neu erstellt oder umgeschrieben und dadurch werden neue Indices $\tilde{I}_2 := \{\tilde{I}_{2,1}, \ldots, \tilde{I}_{2,max}\}$ erstellt und an S_{Backup} zum Speichern gesendet. Der Austausch dieser drei Arten von Datenstrukturen passiert größtenteils parallel und wird daher mit $\tau_{indexing} := Indexing(I_2, IID, \tilde{I}_2)$ modelliert. Dabei ist $\tau_{indexing}$ das Teiltupel von $\tau := prune_R(SID, O)$, das nur die Datenstrukturen aus den Mengen I_2, IID und $\tilde{I}_2$ enthält.

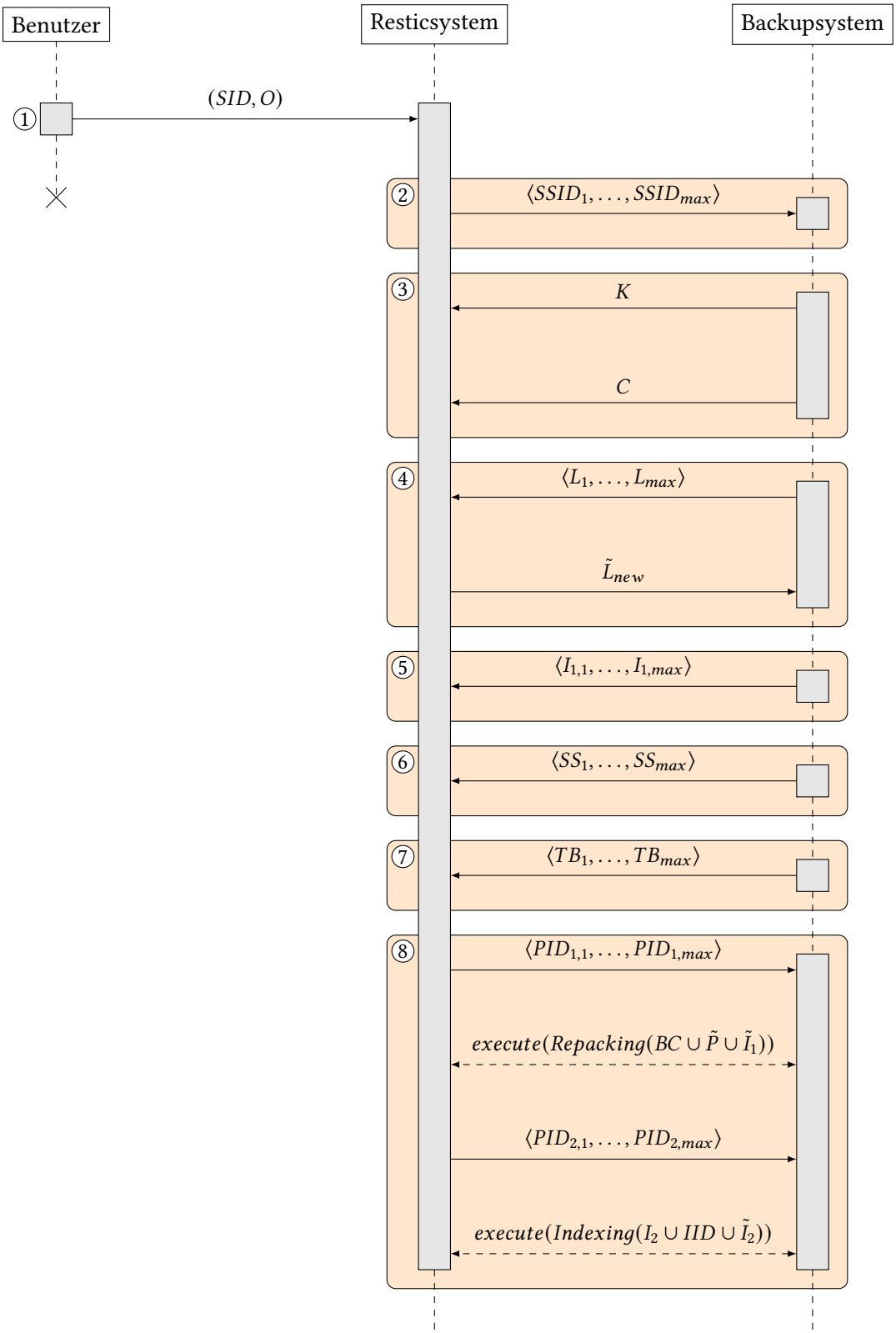

Figure 3.6.: Abstraktes Kommunikationsprotokoll des *prune* Befehls

4. Informationstheoretische Sicherheit der kryptografischen Primitive

Bevor Sicherheitsgarantien für Sicherheitseigenschaften auf Repository-Ebene gegeben werden können, müssen die von Restic verwendeten kryptografischen Primitive genauer betrachtet werden. Die Sicherheit auf Repository-Ebene kann nur gewährleistet werden, wenn es entsprechende Sicherheitsgarantien für die eingesetzten Primitive gibt.
Dazu werden in diesem Kapitel die beiden von Restic eingesetzten kryptografischen Primitive AES256-CTR und Poly1305-AES128 betrachtet und Sicherheitsgarantien für diese Primitive herausgearbeitet. Restic verwendet AES256-CTR um die Sicherheitseigenschaft Vertraulichkeit (siehe Kapitel 2.2.1) zu garantieren, indem alle Daten mit diesem Verfahren verschlüsselt werden. Poly1305-AES128 wird von Restic verwendet, um Integrität (siehe Kapitel 2.2.2) zu garantieren, indem alle verschlüsselten Daten mit MACs versehen werden. Dafür werden für jedes Primitiv jeweils zwei Einsatzszenarien betrachtet. Ein Szenario, das den Einsatz von Restic in einem professionellen Umfeld, beispielsweise einer Softwareentwicklungsfirma, simulieren soll mit sehr großen Datenmengen pro Backup. Das zweite Szenario soll den Einsatz von Restic bei einem privaten Benutzer simulieren mit vielen Backups von kleineren Datenmengen. Daraus wird eine minimale Obergrenze für die Primitive bestimmt, die angibt, welche Datenmenge Restic mit dem gleichen Masterkey maximal verarbeitet werden darf, bevor die informationstheoretische Sicherheit dieser Primitive nicht mehr garantiert werden kann.

4.1. Grundlagen

In diesem Kapitel werden Grundlagen erklärt, die für eine informationstheoretische Sicherheitsbetrachtung von Restics kryptografischen Primitiven relevant sind. Es wird die Struktur von durch Restic verschlüsselte und authentifizierte Datenstrukturen erklärt. Außerdem wird eine Abschätzung für die Komprimierungsrate durch Restics Komprimierungsverfahren eingeführt, damit später eine Obergrenze für die maximale Benutzerdatenmenge bestimmt werden kann, für die Backups erzeugt werden dürfen.

4.1.1. Verschlüsselte Daten

Wie in Kapitel 2.1.5 erwähnt, wird eine verschlüsselte und authentifizierte Restic-Datenstruktur (siehe Kapitel 2.3.4) als verschlüsselte Daten bezeichnet. Dabei enthalten die verschlüsselten

Daten sowohl das Chiffrat der Datenstruktur, als auch den verwendeten IV (16 Byte) und MAC (16 Byte). Eine Pack-Datenstruktur kann per Definition nicht als ein verschlüsseltes Datum bezeichnet werden, sondern stellt eine Menge aus einem oder mehreren verschlüsselten und authentifizierten Blobs und genau einem verschlüsselten und authentifizierten Pack-Header dar. Damit ist eine Pack-Datenstruktur eine Menge oder auch Liste von verschlüsselten Daten.
Restic verwendet wie in Kapitel 2.3.2.2 erklärt AES256 im Counter-Mode (CTR), um eine Restic-Datenstruktur zu verschlüsseln. Restic verwendet außerdem Poly1305-AES128, um für jede verschlüsselte Datenstruktur ebenfalls einen MAC zu berechnen (siehe Kapitel 2.3.2.3). Sowohl AES256-CTR als auch Poly1305-AES128 brauchen zusätzlich zu einem Bytestring, der verschlüsselt oder authentifiziert werden soll, eine weitere Eingabe zur Berechnung des Chiffrats oder des MACs. Diese zusätzliche Eingabe stellt bei AES256-CTR einen 16 Byte großen IV dar und bei Poly1305-AES128 eine 16 Byte große Nonce. Außerdem wird ein MAC in Restic nur für eine Datenstruktur berechnet, wenn diese zuvor verschlüsselt wurde und damit bereits ein IV für diese Datenstruktur gewählt wurde. Restic zieht diesen IV, wie in Kapitel 2.3.2.2 beschrieben, gleichverteilt zufällig als Nonce aus dem Raum $\{0,1\}^{128} \setminus 0^{128}$. Dementsprechend bietet es sich für Restic an, den IV sowohl als zusätzliche Eingabe für AES256-CTR und Poly1305-AES128 zu verwenden und genau das wird auch durch Restic gemacht.
Für Poly1305-AES128 wird der IV verschlüsselt und zur Berechnung des MACs verwendet. Bei der Verschlüsselung mit AES256-CTR wird der Bytestring der Datenstruktur in mehrere 16 Byte große Datenblöcke zerlegt und der IV für jeden Datenblock um 1 inkrementiert (siehe Kapitel 2.2.1.1).

4.1.2. Komprimierung

Daten können von Restic mit einem ZSTD-Algorithmus (siehe Kapitel 2.3.2.1) komprimiert werden, bevor sie verschlüsselt und authentifiziert werden. Diese Komprimierung wirkt sich nur auf den Klartext der Datenstruktur selbst aus und nicht auf den IV oder den MAC einer verschlüsselten Datenstruktur. Im weiteren Verlauf wird die Komprimierungsrate nach unten und oben abgeschätzt, um ein aussagekräftige Worst-Case-Analyse durchführen zu können.
In Restics Komprimierungsalgorithmus können verschiedene Komprimierungslevel eingestellt werden (siehe Tabelle 2.4). Level 1 stellt dabei die geringste Komprimierungsrate und dafür die schnellste Komprimierung dar. Level 4 stellt die größte Komprimierungsrate und langsamste Komprimierung dar. [6] zeigt, dass die Go-Implementierung des ZSTD-Algorithmus eine Komprimierungsrate zwischen 2,8 und 3,5 für die Silesia Corpus Daten aufweist. Für hoch komprimierbare JSON-Daten kommt der ZSTD-Algorithmus auf ein Komprimierungsrate zwischen 8 und 11. Die hoch komprimierbaren JSON-Daten umfassen im Test allerdings eine Größe von ca. 6 GiByte und die Silesia Corpus Daten eine Größe von ca. 200 MiByte. Daten, die von Restic komprimiert werden überschreiten selten 20 MiByte (siehe Tabelle 2.6).
Mit abnehmender Datengröße nimmt auch die Komprimierungsrate ab. Daher lassen sich die gefundenen Komprimierungsraten für den verwendeten ZSTD-Algorithmus nicht ohne

weiteres übernehmen. Diese Masterarbeit verwendet daher Abschätzungen für die Komprimierungsrate. Als größte Komprimierungsrate wird 4 für Data-Blobs gewählt, da Data-Blobs echte Benutzerdaten repräsentieren und keine strukturierten JSON-Datenstrukturen sind. Für alle anderen Restic-Datenstrukturen wird als größte Komprimierungsrate 11 gewählt, da es sich bei allen Datenstrukturen, die keine Data-Blobs sind, um JSON-Datenstrukturen handelt. Als kleinste Komprimierungsrate wird 1 gewählt, womit die Größe der Daten durch die Komprimierung nicht beeinflusst wird. In der Praxis ist es allerdings möglich, dass der Komprimierungsalgorithmus Metadaten zu dem Komprimierungsergebnis hinzufügt, obwohl die Daten selbst nicht komprimiert werden konnten. Damit kann sich im Worst-Case die Größe der Daten nach der Komprimierung sogar vergrößern. In [5] ist erklärt, dass ein Bytestring vom ZSTD-Algorithmus in mehrere Frames mit mehreren Blöcken unterteilt wird. Der Frame-Header besitzt eine maximale Größe von 22 Byte und jeder Block-Header besitzt eine Größe von 3 Byte. In der Regel passen 128 KiByte Daten in einen Block. Daraus folgt mit der kleinsten Komprimierungsrate von 1, dass sich die Größe der Daten pro 128 KiByte im Worst-Case um 25 Byte vergrößert. Um besser mit den Werten rechnen zu können, wird für die kleinste Komprimierungsrate von 1 ein zusätzlicher Komprimierungsoverhead von $2^5 = 32$ Byte pro 128 KiByte (2^{17} Byte) komprimierter Daten angenommen.

4.2. Betrachte Einsatzszenarien von Restic

Dieses Kapitel stellt realistische Einsatzszenarien vor, die nach der Analyse der informationstheoretischen Sicherheit von Restics eingesetzten Primitiven AES256-CTR und Poly1305-AES128 für diese betrachtet werden. Für die Einsatzszenarien wird als Basis das Angriffsszenario aus Kapitel 2.4 betrachtet. Der Angreifer hat vollen Zugriff auf das Backupsystem S_{Backup} und kann über einen beliebig langen Zeitraum Daten des Backupsystems lesen und verändern. Die Festlegung auf einen beliebig langen Angriffszeitraum ist wichtig, da ein typischer Anwendungsfall für eine Backup-Software wie Restic vorsieht, dass regelmäßig Backups erstellt werden. Daher sollte ein Angriff nicht auf die Sicht eines einzelnen Restic-Befehls reduziert werden.
Um eine brauchbare Sicherheitsabschätzung treffen zu können, wird für die Einsatzszenarien immer ein Worst-Case-Szenario betrachtet. Die Szenarien sollten so aufgebaut sein, dass für eine feste Größe von Benutzerdaten die Erfolgswahrscheinlichkeit des Angreifers maximiert wird. Steigt die Erfolgswahrscheinlichkeit des Angreifers beispielsweise mit der Anzahl an Datenblöcken, die er mitliest, werden von Restic im Worst-Case-Szenario so viele Datenblöcke wie möglich erstellt.
Damit lässt sich am Ende eine Aussage über die maximale Größer der Daten treffen, von denen ein Backup erstellt werden darf, ohne die informationstheoretische Sicherheit der kryptografischen Verfahren zu gefährden. Zur Analyse der kryptografischen Verfahren wird für ein Worst-Case-Szenario außerdem davon ausgegangen, dass nur neue, bisher nicht im Repository gespeicherte, Data-Blobs bei einem *backup* Befehl aus den Benutzerdaten entstehen. Denn Data-Blobs, die bereits in einem Repository gespeichert wurden, werden wiederverwendet und damit stehen dem Angreifer weniger Daten zur Verfügung, was eventuell seine Erfolgswahrscheinlichkeit verschlechtert.

Wie gerade angedeutet, wird für die Erstellung neuer Datenstrukturen nur der *backup* Befehl betrachtet. In der Praxis könnte man zusätzlich noch den *restore* Befehl betrachten, der zwar keine neuen Datenstrukturen erstellt, aber bereits im Repository vorhandene Datenstrukturen neu verschlüsselt und zum Repository hinzufügt, während die alten Datenstrukturen aus dem Repository entfernt werden. Das würde für die informationstheoretische Sicherheit keinen Unterschied machen, aber die Grenze für die maximale Datenmenge, von der ein Backup erstellt werden darf, würde sich verringern. Die Obergrenze für Benutzerdaten, von denen ein Backup erstellt werden darf, soll nur ein Richtwert gesehen werden und nicht als verbindliche und eindeutige Grenze. Außerdem hängt die informationstheoretische Sicherheit nicht von dieser Obergrenze ab, sondern von Werten, die sich nicht durch Betrachtung anderer Restic-Befehle ändern. Aus diesem Grund wird in den nachfolgenden Szenarien nur der *backup* Befehl betrachtet, wenn es darum geht neue Daten zu verschlüsseln und dem Repository hinzuzufügen.

4.2.1. Szenario 1 (Tägliche Backups großer Daten)

Dieses Einsatzszenario soll den Einsatz von Restic in einem größeren System darstellen, beispielsweise bei einem kleinen bis mittleren IT-Unternehmen.
In diesem Szenario wird ein Ausgangsrepository mit 40 TiByte an initialen Daten betrachtet. Einmal am Tag wird auf S_{Restic} der *backup* Befehl für dieses Repository ausgeführt. Von ungefähr 40 TiByte Benutzerdaten ändern sich jeden Tag 10%. Im Worst-Case wären das 4 TiByte komplett neuer Data-Blobs, die bisher nicht im Repository gespeichert sind. Die 10% dienen als konservative Schätzung für ein aktives IT-Unternehmen.

4.2.2. Szenario 2 (Viele Backups mit kleinen Daten)

Das zweite Einsatzszenario stellt den Einsatz Restics durch ein Privatperson dar. Eine Privatperson erstellt in der Regel nicht täglich neue Backups mit 4 TiByte neuer Benutzerdaten, sondern viele kleine Backups mit wenig Daten. Durch dieses Verhalten könnte es sein, dass der Overhead von Restics Datenstrukturen, die zur Verwaltung des Repositorys erstellt werden, viel größer ist als bei wenig Backups mit einer großen Menge von Daten. Aus diesem Grund wird dieses zweite Einsatzszenario für eine Privatperson betrachtet.
In diesem Szenario wird ein Ausgangsrepository mit 8 TiByte an initialen Daten betrachtet. Alle 30min wird auf S_{Restic} der *backup* Befehl für dieses Repository ausgeführt. Von den 8 TiByte ändern sich bei jedem Backup 8 GiByte. Im Worst-Case wären das 8 GiByte komplett neuer Data-Blobs, die bisher nicht im Repository gespeichert sind. 8 GiByte pro 30 Minuten ist für eine Privatperson ebenfalls eine sehr konservative Schätzung. Wie sich bei der späteren Betrachtung der Einsatzszenarien herausstellt, ist das allerdings eine notwendige Schätzung, da bei weniger Daten mehrere Jahrzehnte vergehen würde, bis die informationstheoretische Sicherheitsschranke von Restics kryptografischen Primitiven erreicht ist.

4.3. Vertraulichkeit

Das kryptografische Primitiv, das von Restic eingesetzt wird, um die Sicherheitseigenschaft Vertraulichkeit zu gewährleisten, ist AES256-CTR. Dieses Kapitel betrachtet die informationstheoretische Sicherheit des von Restic verwendeten Verschlüsselungsverfahren AES256-CTR. Dazu wird zunächst eine Abschätzung für die Erfolgswahrscheinlichkeit eines Angreifers auf das IND$ Sicherheitsspiel für Restics AES256-CTR betrachtet.
Es wird ein NIST Standard, für eine akzeptable Kollisionswahrscheinlichkeit der IVs bei GCM verwendet, um die IND$ Sicherheit abzuschätzen.
Danach werden Worst-Case-Bedingungen für die Einsatzszenarien in Bezug auf Restics AES256-CTR herausgearbeitet und die vorgestellten Einsatzszenarien aus Kapitel 4.2 untersucht. Damit wird eine minimale obere Schranke der Benutzerdatenmenge bestimmt, von denen Restic ein Backup erstellen darf, ohne die Sicherheit von AES256-CTR zu gefährden. Außerdem wird bestimmt nach welcher Zeitspanne in den jeweiligen Einsatzszenarien diese Schranke erreicht ist.

4.3.1. IND$ Sicherheit

In diesem Kapitel wird eine Abschätzung für die Erfolgswahrscheinlichkeit eines Angreifers $\mathcal{A}$ für die IND$ Sicherheit von Restics AES256-CTR Implementierung aufgestellt.
[9] kommt zu der Abschätzung 4.1 für die Erfolgswahrscheinlichkeit $Adv^{IND\$}_{CTR[AES256]}(\mathcal{A})$, dass ein Angreifer $\mathcal{A}$ ein Chiffrat für einen von ihm gewählten Klartext von zufällig gleichverteilt gezogenen Bytestrings (uniform) gleicher Länge unterscheiden kann. Dabei stellt σ die Anzahl an Datenblöcken dar, die A vorher von einem Orakel mit dem gleichen Schlüssel verschlüsseln ließ und $k := 128$ ist hier die Größe der verwendeten Datenblöcke und damit auch die Größe des IVs. Diese Abschätzung basiert auf der Annahme, dass eine echte Zufallsfunktion im Counter-Mode ohne Einschränkungen IND$ sicher ist, solange kein für die Verschlüsselung eines Datenblocks verwendetes $IV + i$ für die Verschlüsselung eines anderen Datenblocks erneut verwendet wird. Da es sich bei AES um eine Permutation handelt, verwendet [9] das PRP/PRF Switching Lemma 2.3, um auf die Abschätzung 4.1 zu kommen.

$$Adv^{IND\$}_{CTR[AES256]}(\mathcal{A}) \leq Adv^{PRF}_{AES256}(\mathcal{B}) \leq Adv^{PRP}_{AES256}(\mathcal{B}) + \frac{\sigma^2}{2^{k+1}} \tag{4.1}$$

Der Term $Adv^{PRP}_{AES256}(\mathcal{B})$ ist die Wahrscheinlichkeit, dass ein Angreifer $\mathcal{B}$ die Blockchiffre AES256 von einer echten Zufallspermutation unterscheiden kann. [9] verwendet das Switching Lemma, um auszunutzen, dass es sich bei AES um eine Permutation handelt. Denn AES256 gilt als sehr sicher bezüglich seiner PRP-Eigenschaft und gilt über die Jahre als sehr gut erforscht. Daher ist allgemein anerkannt, dass der Term $Adv^{PRP}_{AES256}(\mathcal{B})$ sehr klein ist, aber genau bestimmt werden kann er nicht.
In dieser Masterarbeit wird die Wahrscheinlichkeit $Adv^{PRP}_{AES256}(\mathcal{B})$ weiterhin mitgeführt, auch wenn davon ausgegangen wird, dass sie für alle PPT-Angreifer vernachlässigbar in

ihrer Eingabegröße ist. [9] betrachtet ein Counter-Mode Verfahren, bei dem für jeden zu verschlüsselnden Bytestring D nicht nur ein IV gewählt wird, sondern ein Vektor von aufsteigenden Zahlen. Dieser Vektor von Zahlen besitzt genau einen Eintrag für jeden Datenblock des Bytestrings D. Jeder Eintrag in dem Vektor gehört zu einem Datenblock D_i des Bytestrings $D := D_0 || \ldots || D_m$ und der Wert dieses Eintrags ist $IV + i$, wobei IV der erste Eintrag in dem Vektor ist. Damit enthält diese Liste alle vom Counter-Mode zur Verschlüsselung verwendeten Nonces für einen Bytestring. Die Abschätzung 4.1 gilt nach [9] nur für einen Angreifer, der für jeden Klartext, den er sich verschlüsseln lässt, einen Vektor wählt, der nur Einträge enthält, die zuvor in keinem anderen vom Angreifer gewählten Vektor vorkamen. Damit kann diese Abschätzung nicht direkt auf Restic übertragen werden, da Restic bei der Wahl eines IVs nicht überprüft, ob der IV oder einer der $IV + i$ bereits für ein anderes Chiffrat verwendet wurde. Ein weiterer wichtiger Unterschied ist, dass bei [9] vorgestellter Abschätzung 4.1, der Angreifer $\mathcal{A}$ den IV selbst wählen darf, der zur Erzeugung des Chiffrats verwendet wird. In Restic wird die Wahl des IVs durch Restic und damit durch das Protokoll selbst übernommen und nicht durch einen Außenstehenden. Damit würde die Wahrscheinlichkeit $Adv^{IND\$}_{CTR[AES256]}$ für Restic sinken, da ein Angreifer, der einen Vorteil durch die Wahl des IVs hat, diesen Vorteil bei Restic verliert. Dadurch gilt jedoch die Abschätzung 4.1 insbesondere auch für Restic, sofern die Abschätzung an Restics Wahl der IVs angepasst wird.

Das Problem der IV-Kollisionen:
Bevor betrachtet wird, wie die Abschätzung 4.1 an Restics AES256-CTR Umsetzung angepasst wird, wird betrachtet, warum eine Kollision der IVs problematisch wäre. In Restic wird für jeden Klartext der verschlüsselt werden soll ein IV zufällig gleichverteilt gezogen. Der Counter-Mode inkrementiert für jeden Datenblock des Klartextes den gezogenen IV um eins (siehe Kapitel 2.3.2.2). Dadurch kann es in Restic nicht nur zu einer Kollision der gezogenen IVs kommen, sondern auch zu einer Kollision innerhalb zwei verschiedener Klartexte D und E mit zwei verschiedenen IVs IV_D und IV_E, wenn beispielsweise gilt $IV_D + i = IV_E + j$ für die Datenblöcke D_i und E_j. Eine solche Kollision führt mit dem gleichen Schlüssel k für AES256-CTR zur Verletzung der Sicherheit und sollte vermieden werden.
Angenommen beim Verschlüsseln eines Bytestrings D mit AES256-CTR wird für einen der Datenblöcke D_i eine Nonce $N = IV_D + i$ verwendet, die bereits zu einem früheren Zeitpunkt als Nonce $N = IV_B + j$ zur Verschlüsselung eines Datenblocks B_j eines anderen Bytestrings B verwendet wurde. Wie in Kapitel 2.2.1.1 beschrieben, ist für den Datenblock D_i das Chiffrat $AES256_k(N) \oplus D_i$ und für B_j ist das Chiffrat $AES256_k(N) \oplus B_j$, wobei k der verwendete Schlüssel ist. Ein Angreifer kann nun die Chiffrat-Blöcke der beiden Datenblöcke D_i und B_j mit einem XOR miteinander verrechnen und erhält die XOR-Summe der beiden Klartext-Datenblöcke $D_i \oplus B_j$, da sich $AES256_k(N)$ wegkürzt. Damit kann es dem Angreifer möglich sein, die beiden Klartext-Blöcke zu rekonstruieren. Durch das deterministische Inkrementieren im Counter-Mode kollidieren außerdem auch die Nonces aller nachfolgenden Blöcke D_{i+z} und B_{j+z} für $0 \leq z$. Wenn der Angreifer beispielsweise einen der Datenblöcke kennt, kann er durch ein weiteres XOR den anderen Klartext-Datenblock aus dem XOR-Ergebnis der Chiffratblöcke berechnen. So kann bei einer solchen Nonce-Kollision die IND$ Sicherheit gebrochen werden, da der Angreifer einen der beiden Klartextblöcke bereits zu einem

früheren Zeitpunkt verschlüsseln ließ und so den anderen Klartextblock berechnen kann. Damit kann der Angreifer herausfinden, ob sein echter Klartext verschlüsselt wurde oder, ob das Verschlüsselungsorakel dem Angreifer irgendwelche zufälligen Bytestrings zurückgibt. Auch in Restic kann dies ein realistisches Angriffsszenario sein, da die meisten Datenstrukturen in ein JSON-Format umgewandelt werden, bevor sie verschlüsselt werden. Dadurch gibt es für die meisten verschlüsselten Datenstrukturen Datenblöcke, zu denen der Angreifer durch das JSON-Format den Klartext kennt.

Anpassung der Abschätzung 4.1 an Restics AES256-CTR:
Damit die Abschätzung 4.1 auch für Restic verwendet werden kann, muss die Abschätzung für die veränderte Wahl von Restics IVs angepasst werden. Dazu wird der Unterschied zwischen dem Szenario aus [9] und Restic betrachtet. Der einzige Unterschied ist, dass es bei Restic dazu kommen kann, dass ein $IV + i$ für zwei verschiedene Datenblöcke verwendet wird und es damit zu einer Kollision der verwendeten $IV + i$ kommt. Dazu wird im Folgenden die Kollisionswahrscheinlichkeit für die verwendeten $IV + i$ betrachtet. Mit der Kollisionswahrscheinlichkeit wird die Abschätzung 4.5 für die IND$ Sicherheit von Restics Umsetzung von AES256-CTR hergeleitet.
Restic zieht für jeden Klartext, der verschlüsselt werden soll, einen zufälligen IV. Dieser IV wird mit einem Zufallszahlengenerator gezogen, der eine gleichverteilte Nonce zwischen aus dem Raum $\{0,1\}^{128} \setminus 0^{128}$ zieht. Dadurch kann es passieren, dass Restic zweimal die gleiche Zahl für einen IV wählt und so eine Nonce-Kollision auslöst. Durch die Gleichverteilung der Nonces und mit der Birthday-Boundary aus Kapitel 2.2.5 ergibt sich die Wahrscheinlichkeit $\frac{\lambda^2}{2^{128+1}}$ für eine Nonce-Kollision der IVs, wobei λ die Anzahl der generierten IVs und damit die Anzahl der verschlüsselten Bytestrings ist. Diese Abschätzung ist allerdings zu schwach für den Counter-Mode, da es auch zu Kollisionen der $IV + i$ für Datenblöcke innerhalb zweier Bytestrings kommen kann. Diese Kollision und das daraus resultierende Problem wurde im vorherigen Abschnitt gezeigt.
Um eine aussagekräftige Abschätzung für die IND$ Sicherheit zu finden, muss also die Kollision aller IVs und $IV + i$ betrachtet werden, die jemals mit dem gleichen Schlüssel k verwendet wurden. Da sich in Restic über die gesamte Lebensdauer eines Repositorys niemals der Schlüssel k ändert, können einfach alle Datenstrukturen betrachtet werden, die jemals von Restic für das gleiche Repository verschlüsselt werden. Betrachtet man einen Bytestring, der aus L Datenblöcken besteht, mit einem dazugehörigen IV können alle für die Verschlüsselung dieses Bytestrings verwendeten $IV + i$ als ein ganzzahliges Intervall $[IV; IV + (L-1)]$ aufgefasst werden. Wenn für zwei IV-Intervalle I_i und I_j gilt $I_i \cap I_j \neq \emptyset$ gilt, kommt mindestens eine Zahl aus I_i ebenfalls in I_j vor und damit kollidieren die beiden Intervalle I_i und I_j. Damit lässt sich die IND$ Sicherheit von Restics AES256-CTR mit der Kollisionswahrscheinlichkeit der IV-Intervalle aller mit dem gleichen Schlüssel verschlüsselten Bytestrings abschätzen. Dazu werden im Folgenden m verschlüsselte Bytestrings betrachtet, die aus genau L Datenblöcken bestehen. Es gibt in Restic zwei Einschränkungen, die zunächst für die Betrachtung der Wahrscheinlichkeit nicht berücksichtigt werden. Die eine Einschränkung ist, dass Restic den IV-Raum $[1; N-1]$ verwendet und die nachfolgende Betrachtung den IV-Raum $[0; N-1]$ betrachtet. Die zweite Einschränkung ergibt sich durch den Counter-Mode, der beim Inkrementieren eines IVs für die einzelnen Daten-

blöcke einen Überlauf durch die Modulo-Operation verarbeitet. Damit wird ein IV, der beim Inkrementieren größer wird als $N-1$ wird, mit Modulo N verrechnet, wodurch das IV-Intervall bei 0 fortfährt. Damit wird der IV-Raum von Restic wie ein Ring behandelt und entspricht bei Hinzunahme der 0 dem Restklassenring $\mathbb{Z}/2^N\mathbb{Z}$. In der folgenden Betrachtung der Intervallkollisionen wird der IV-Raum zunächst nur als linearer Raum betrachtet, bei dem die inkrementierten IVs linear über $N-1$ hinauswachsen können. Am Ende wird ein Korrekturterm eingeführt, der die berechnete Wahrscheinlichkeit für einen ringförmigen IV-Raum anpasst. Im Folgenden wird für jedes Intervall und damit für jeden Bytestring eine Variable Z_i eingeführt und das Problem der Intervallkollision auf ein Ordnungsproblem für die Variablen Z_i reduziert. Für jeden dieser m Bytestrings wird ein IV zufällig aus einem gleich verteilten Nonce-Raum $[0; N-1]$ gezogen. Die verschlüsselten Bytestrings werden nach dem Wert ihrer gewählten IVs aufsteigend geordnet, sodass für Bytestrings D_i und D_j paarweise gilt $IV_i \leq IV_j \Leftrightarrow i < j$. Die verwendeten Indizes der geordneten Bytestrings, IVs und Intervalle sind weiterhin die Zahlen 1 bis m. Die Bytestrings und damit auch die IV-Intervalle sind nach ihren IVs aufsteigend geordnet und jedes IV-Intervall ist ganzzahlig und zusammenhängend durch den Counter-Mode konstruiert. Damit gilt die folgende Implikation, die besagt, dass sobald zwei IV-Intervalle kollidieren, auch alle IV-Intervalle zwischen diesen zwei Intervallen kollidieren.

$$I_i \cap I_{i+j} \neq \emptyset \implies IV_{i+j} \in I_i \implies \forall 0 \leq z \leq j : IV_{i+z} \in I_i \implies \forall 0 \leq z \leq j : I_i \cap I_{i+z} \neq \emptyset$$

Um eine Intervallkollision bei m Intervallen festzustellen, reicht es bei geordneten Intervallen aus benachbarte Intervalle zu betrachten. Zwei IV-Intervalle I_i und I_{i+1} kollidieren genau dann, wenn für den Abstand ihrer IVs gilt $IV_{i+1} - IV_i < L$. Mit dieser Erkenntnisse wird jedem Intervall I_i eine Zahl $Z_i := (IV_i - (i-1) \cdot L) \in \mathbb{Z}$ zugeordnet. Diese Zahl Z_i ist der zugehörige IV IV_i, der um so viele Intervalllängen L nach vorne verschoben wird, wie es kleinere IVs als IV_i gibt. Es gibt vor dem geordneten IV_i von Z_i genau $(i-1)$ kleinere IVs, die jeweils mindestens einen Abstand von L zueinander benötigen, damit keins dieser Intervalle kollidiert. Damit bleibt im Falle, dass es keine Kollision der Intervalle gibt, die Ordnung der IVs auf den konstruierten Z_i erhalten. Diese Aussage lässt sich mathematisch wie folgt beweisen:
Überlappen sich zwei aufeinanderfolgende Intervalle I_i und I_{i+1}, gilt $IV_{i+1} - IV_i < L$ und damit gilt $Z_i > Z_{i+1}$, da $Z_{i+1} - Z_i = (IV_{i+1} - IV_i) - (((i+1)-i) \cdot L) = (IV_{i+1} - IV_i) - L < 0$. Wenn sich zwei aufeinanderfolgende Intervalle I_i und I_{i+1} nicht überlappen, gilt $IV_{i+1} - IV_i \geq L$ und damit gilt $Z_i \leq Z_{i+1}$, da $Z_{i+1} - Z_i = (IV_{i+1} - IV_i) - (((i+1)-i) \cdot L) = (IV_{i+1} - IV_i) - L > 0$. Damit ist eine Ordnung auf den Z_i gegeben und es gilt für aufsteigend geordnete IVs die Äquivalenz 4.2.

$$Z_1 \leq Z_2 \leq \cdots \leq Z_m \iff \forall 1 \leq i \leq m-1 : IV_{i+1} - IV_i \geq L \tag{4.2}$$

Damit kommt es bei m verschlüsselten Bytestrings, die aus L Datenblöcken bestehen, genau dann zu keiner Kollision, wenn die zu den aufsteigend geordneten IVs gehörigen Z_i aufsteigend geordnet sind. m aufeinanderfolgend gezogene IVs werden im Folgenden als IV-Tupel bezeichnet und das Tupel $(Z_1, \ldots, Z_m)$ zu den aufsteigend geordneten IVs wird als

Z-Tupel bezeichnet, wobei Z_i immer zu dem geordneten IV_i gehört. Die Wahrscheinlichkeit für keine Intervallkollision bei m verschlüsselten Bytestrings entspricht also genau der Wahrscheinlichkeit mit der ein Z-Tupel für m gezogene IVs aufsteigend geordnet ist. Daraus kann mit der Gegenwahrscheinlichkeit später ermittelt werden, wie hoch die Wahrscheinlichkeit für mindestens eine Intervallkollision ist.
Dazu wird zunächst bestimmt wie viele Möglichkeiten es für m gezogene IVs gibt, sodass das zugehörige Z-Tupel aufsteigend geordnet ist. Die IVs werden in der Regel nicht aufsteigend geordnet gezogen, daher muss berücksichtigt werden, dass für jedes Z-Tupel genau $m!$ mögliche IV-Permutationen existieren. Damit existieren zu jedem Z-Tupel $m!$ mögliche zugehörige IV-Tupel. Nun muss die Anzahl alle möglichen Z-Tupel bestimmt werden, deren Einträge aufsteigend geordnet sind. Zur Erinnerung, die Einträge in einem Z-Tupel sind in der gleichen Reihenfolge, wie die Einträge des geordneten IV-Tupels und Z_i gehört zu IV_i. Damit sind die Einträge des Z-Tupel nur im Fall keiner Intervallkollision ebenfalls aufsteigend geordnet und damit ist Z_1 der kleinste Eintrag des Z-Tupels. Bei einem solchen geordneten Tupel muss das kleinste Z_1 mindestens den Wert 0 besitzen, da gilt $Z_1 := IV_1 - 0$ und der kleinste IV ebenfalls 0 ist. Alle nachfolgenden Z_i sind gleich oder größer, da das Z-Tupel aufsteigend geordnet ist und das größte Z_m kann maximal den Wert $(N-1) - (m-1) \cdot L$ besitzen. Damit können die Z_i eines geordneten Z-Tupels höchstens $M := N - (m-1) \cdot L$ verschiedene Werte annehmen. Bei Z-Tupeln mit m Einträgen und insgesamt M verschiedenen Werten für die Z_i gibt es genau $\binom{M}{m}$ verschiedene Z-Tupel, die aufsteigend geordnet sind. Damit ergeben sich bei N^m möglichen IV-Tupeln genau $m! \cdot \binom{M}{m}$ IV-Tupel, deren Z-Tupel ebenfalls aufsteigend geordnet sind. Nun kann die Gleichverteilung der IVs genutzt werden, um die Wahrscheinlichkeit $\frac{m! \cdot \binom{M}{m}}{N^m}$ für keine Kollision der IV-Intervalle bei m Verschlüsselungen von Bytestrings bestehend aus L Datenblöcken zu berechnen. Diese Wahrscheinlichkeit lässt sich umformen, wie in Gleichung 4.3 gezeigt.

$$\begin{aligned}
\frac{m! \cdot \binom{M}{m}}{N^m} &= \frac{(M-0) \cdot (M-1) \cdot \dots \cdot (M-(m-1))}{N^m} \\
&= \prod_{i=0}^{m-1} \left(\frac{M-i}{N}\right) \\
&= \prod_{i=0}^{m-1} \left(\frac{N-(m-1)\cdot L - i}{N}\right) \\
&= \prod_{i=0}^{m-1} \left(1 - \frac{(m-1)\cdot L + i}{N}\right)
\end{aligned} \tag{4.3}$$

Um das Produkt in der Gleichung aufzulösen, kann der natürliche Logarithmus und die Exponentialfunktion verwendet werden. Damit die erste Hälfte der Gleichung 4.4 gilt, muss gezeigt werden, dass gilt $\frac{(m-1)\cdot L+i}{N} << 1$. Nur mit $x << 1$ lässt sich die Taylor-Entwicklung $log(1-x) \approx -x$ anwenden, da die eigentliche Taylor-Reihe $log(1-x) = -\sum_{i=1}^{\infty} \frac{x^i}{i}$ ist. Um das zu garantieren und zu zeigen, dass die Summe $\sum_{i=2}^{\infty} \frac{x^i}{i}$ vernachlässigbar in x ist, werden die Parameter betrachtet, die die Summe maximieren. Damit $\frac{(m-1)\cdot L+i}{N}$ maximal wird, muss der

Nenner $(m-1)\cdot L+i$ maximal werden. i wird durch die Summe maximal $m-1$ groß. Da die Wahrscheinlichkeit 4.4 verwendet werden soll, um einen maximalen Wert an verschlüsselten Nachrichten zu bestimmen, gibt es ebenfalls Obergrenzen für die Parameter m und L. Die IVs werden gleich verteilt zufällig gezogen. Daher ergibt sich durch die Birthday-Boundary der Nonce-Kollision auf einem Raum der Größe 2^{128} für die maximale Anzahl m an zu ziehenden IVs der Wert $m < 2^{64}$. Das heißt, die Wahrscheinlichkeit 4.4 wird niemals mit einem größeren Wert als $m = 2^{64}$ verwendet. Tatsächlich wird die Wahrscheinlichkeit auch nur mit Werten $m << 2^{64}$ verwendet, da Werte um 2^{64} eine bereits viel zu hohe Wahrscheinlichkeit für eine Kollision besitzen.
Ebenso ist die größte von Restic verschlüsselte Datenstruktur ein Tree-Blob mit einer Größe von 4 GiByte (siehe Tabelle 2.6). Damit ergibt sich für L ein maximaler Wert von 2^{28} Datenblöcken pro Klartext. Mit diesen Werten ergibt sich $2^{-36} \approx \frac{(2^{64}-1)\cdot 2^{28}+(2^{64}-1)}{2^{128}} < 2^{-35} << 1$. Damit gilt für die Summe $\sum_{i=2}^{\infty} \frac{2^{-35\cdot i}}{i} < \frac{2^{-35\cdot 2}}{2}\cdot 2 = 2^{-70}$ und ist im Vergleich zu dem Term $x \approx 2^{-36}$ in der Taylor-Reihe vernachlässigbar. Somit darf die Abschätzung des Logarithmus in Gleichung 4.4 angewendet werden.
Zum Schluss der Gleichung kann die Summe mit der Gaußsche Summenformel aufgelöst werden.

$$
\begin{aligned}
\exp(\log(\prod_{i=0}^{m-1}(1-\frac{(m-1)\cdot L+i}{N}))) &= \exp(\sum_{i=0}^{m-1}\log(1-\underbrace{\frac{(m-1)\cdot L+i}{N}}_{x})) \\
&\overset{x<<1}{\approx} \exp(-\sum_{i=0}^{m-1}\frac{(m-1)\cdot L+i}{N}) \\
&= \exp(-\sum_{i=0}^{m-1}\frac{(m-1)\cdot L+i}{N}) \\
&= \exp(-\sum_{i=0}^{m-1}\frac{(m-1)\cdot L}{N}-\sum_{i=0}^{m-1}\frac{i}{N}) \\
&= \exp(-\frac{m(m-1)\cdot L}{N}-\frac{m(m-1)}{2\cdot N}) \\
&= \exp(-\frac{m(m-1)(2L+1)}{2N})
\end{aligned}
\tag{4.4}
$$

Damit kann nun eine Abschätzung für die Erfolgswahrscheinlichkeit eines Angreifers auf die IND$ Sicherheit von AES256-CTR formuliert werden, sofern die zwei Einschränkungen von Restic zusätzlich noch betrachtet werden. In Restic ist der IV-Raum $\{0,1\}^{128} \setminus 0^{128}$, wobei in der bisherigen Betrachtung der IV-Raum $\{0,1\}^{128}$ betrachtet wurde. Damit ist der eigentliche IV-Raum minimal kleiner, als für die Abschätzung 4.4 vorausgesetzt. Für die lineare Betrachtung der Intervallkollision, wirkt sich eine Verkleinerung des Nonce-Raums nur auf den Wert N aus, der im Fall von Restic den Wert $N := 2^{128}-1$ annimmt. Der Unterschied zwischen Restics IV-Raum und dem IV-Raum, der die Null enthält, ist jedoch auch so gering, dass man $N := 2^{128}$ wählen könnte, ohne dass sich die Kollisionswahrscheinlichkeit

relevant verändert.
Nun muss die Ring-Eigenschaft des IV-Raums von Restic betrachtet werden. In die bisherige Betrachtung ist eine Kollision an den Grenzen des IV-Raums nicht mit eingeflossen. Bei der linearen Betrachtung besitzen Intervalle an den Grenzen eine minimal geringere Kollisionswahrscheinlichkeit als alle anderen Intervalle. Bei einem ringförmigen IV-Raum hingegen besitzen alle Intervalle die gleiche Kollisionswahrscheinlichkeit. Dazu kann eine einfache konservative Abschätzung zu der Kollisionswahrscheinlichkeit auf einem linearen IV-Raum hinzugefügt werden. Man kann für die konservative Abschätzung die Annahme treffen, dass jedes Intervall kollidiert, sobald dessen $IV \in [N - L + 1; N - 1]$ innerhalb der letzten $L - 1$ IVs des IV-Raums liegt, wodurch es bei dem Inkrementieren zu einem Überlauf kommt. Damit stellt diese Annahme eine Überabschätzung für den Fall dar, dass das Intervall mit einem zweiten Intervall, dessen IV Teil der kleinsten IVs des IV-Raums ist. Ein zweites Intervall wird für die konservative Abschätzung erst gar nicht betrachtet und es wird immer davon ausgegangen, dass ein Intervall an der Grenze des IV-Raums kollidiert. Die Wahrscheinlichkeit, dass ein gezogener IV nicht im Intervall $[N - L + 1; N - 1]$ liegt ist $1 - \frac{L-1}{N}$, da die IVs gleichverteilt gezogen werden. Damit ist die Wahrscheinlichkeit bei m zufällig gleichverteilt gezogenen IVs, dass keiner dieser IVs in den dem Grenzintervall liegt $(1 - \frac{L-1}{N})^m$. Somit folgt für die Wahrscheinlichkeit, dass mindestens einer der gezogenen IVs in dem Grenzintervall liegt und dadurch eine konservativ geschätzte Kollision auf Grund der Ringstruktur verursacht $1 - (1 - \frac{L-1}{N})^m$.
Jetzt müssen alle Kollisionswahrscheinlichkeiten und die ursprüngliche IND$ Sicherheitsgarantie zusammengesetzt werden. Die Erfolgswahrscheinlichkeit eines Angreifers $\mathcal{A}$ ein IND$ Sicherheitsspiel für AES256-CTR zu gewinnen, bei dem es keine Kollisionen bezüglich der IV-Intervalle geben kann, ist in 4.1 dargestellt. In Restic kommt die Wahrscheinlichkeit dazu, dass es bei m zufällig gleichverteilt gezogenen IVs zu einer Kollision der IV-Intervalle kommt, wodurch $\mathcal{A}$ ebenfalls das IND$ Sicherheitsspiel gewinnen würde. Diese zusätzliche Wahrscheinlichkeit ergibt sich aus der Gegenwahrscheinlichkeit der Wahrscheinlichkeit 4.3 für keine Kollision auf einem linearen IV-Raum und der konservativen Abschätzung $1-(1-\frac{L-1}{N})^m$ für eine mögliche Kollision an den IV-Raum Grenzen, wie sie bei einem ringförmigen IV-Raum auftritt. Damit ergibt sich die Abschätzung 4.5 für die Erfolgswahrscheinlichkeit eines Angreifers $\mathcal{A}$ das IND$ Sicherheitsspiel für Restics Implementierung von AES256-CTR zu brechen. m bezeichnet die Anzahl der verschlüsselten Klartexte und L die Anzahl der Datenblöcke aus der ein Klartext besteht.

$$\begin{aligned} Adv^{IND\$}_{CTR[AES256]}(\mathcal{A}) \leq Adv^{PRP}_{AES256}(\mathcal{B}) &+ \frac{(m \cdot L)^2}{2^{128+1} - 2} \\ &+ (1 - \exp(-\frac{m(m-1)(2L+1)}{2^{128+1} - 2})) \\ &+ (1 - (1 - \frac{L-1}{2^{128} - 1})^m) \end{aligned} \tag{4.5}$$

4.3.2. Obergrenze für IND$ Sicherheit

In diesem Kapitel wird eine Grenze für die Erfolgswahrscheinlichkeit eines Angreifers $\mathcal{A}$ auf die IND$ Sicherheit von Restics AES256-CTR aufgestellt. Es wird gezeigt, dass $Adv^{IND\$}_{CTR[AES256]}$ für Restics AES256-CTR dominiert wird durch einen Term für die Kollisionswahrscheinlichkeit der Initialisierungsvektoren. Dementsprechend muss für eine akzeptable Grenze der IND$ Sicherheit ebenfalls eine akzeptable Grenze für die Kollisionswahrscheinlichkeit gefunden werden. So sollte eine akzeptable Kollisionswahrscheinlichkeit der IVs beispielsweise weit unter der Birthday-Boundary liegen.
Der NIST Standard SP-800-38D [3] sagt Folgendes zu einer Blockchiffre im Galois/Counter-Mode (GCM):

"The probability that the authenticated encryption function ever will be invoked with the same IV and the same key on two (or more) distinct sets of input data shall be no greater than 2^{-32}*"*

Dieser NIST Standard [3] sagt, dass für GCM die Wahrscheinlichkeit, dass unterschiedliche Bytestrings mit dem gleichen IV und dem gleichen Schlüssel verschlüsselt und authentifiziert werden, maximal 2^{-32} sein soll. Der GCM ist ein Modus, mit dem gleichzeitig das Chiffrat eines Klartextes erstellt wird, als auch ein MAC berechnet wird. Zur Verschlüsselung benutzt GCM ebenfalls den Counter-Mode mit einer Blockchiffre. Der Unterschied bei der Verschlüsselung mit GCM ist, dass ein 12 Byte großer IV pro Klartext gewählt wird, an den ein 4 Byte großer Zähler konkateniert wird. Zusammen ergibt das für jeden Datenblock eine 16 Byte große Nonce wie beim Counter-Mode. Damit existiert in GCM nicht mehr das Problem, dass es zu einer Kollision der IV-Intervalle verschiedener Klartexte kommen kann, wenn die IVs selbst keine Kollision erzeugt haben. Allerdings erhöht sich bei GCM die Chance auf eine IV-Kollision für die reine Wahl des IVs durch den kleineren IV-Raum von 2^{96} anstelle von $2^{128}-1$ möglichen IVs wie bei Restic. Der NIST Standard bezieht sich explizit auf die Kollisionswahrscheinlichkeit der gezogenen Nonces für GCM, wodurch das XOR Ergebnis beider Klartexte und nicht nur einzelner Datenblöcke für einen Angreifer sichtbar werden würde. Außerdem wird GCM sowohl zur Garantie der Vertraulichkeit als auch der Integrität verwendet, wodurch direkt zwei Sicherheitseigenschaften gebrochen werden könnten, wenn GCMs Sicherheit gebrochen wird. Damit könnte man davon ausgehen, dass der NIST Standard bei der Grenze von 2^{-32} sehr streng ist und eine Obergrenze für Restics AES256-CTR eventuell großzügiger und damit größer als 2^{-32} ausfallen könnte.
Dennoch wird auch in dieser Masterarbeit für Restics AES256-CTR eine akzeptable Obergrenze von 2^{-32} für die Kollisionswahrscheinlichkeit aller IV-Intervalle festgelegt. Das heißt die Wahrscheinlichkeit, dass die verwendeten Nonces für die Verschlüsselung zweier Datenblöcke gleich sind, darf maximal 2^{-32} sein. Nach der bestimmten Erfolgswahrscheinlichkeit 4.5 für Restics AES256-CTR ergibt sich die Beschränkung 4.6 für die Wahrscheinlichkeit einer Intervallkollision.

$$(1-\exp(-\frac{m(m-1)(2L+1)}{2^{129}-2}))+(1-(1-\frac{L-1}{2^{128}-1})^m) \overset{!}{\leq} 2^{-32} \tag{4.6}$$

Damit wurde zwar keine konkrete Grenze für die IND$ Sicherheit von Restics AES256-CTR Umsetzung getroffen, aber mit der Beschränkung 4.6 und den gleichen Parametern m und L in Abschätzung 4.5 kann ebenfalls eine Schranke für die IND$ Sicherheit 4.5 betrachtet werden. Alle Beschränkungen hängen sowohl von der Anzahl m gezogener IVs und der Anzahl L von Datenblöcke in jedem verschlüsselten Klartext ab. Mit welcher Wahl der Parameter wird die Grenze von 2^{-32} für möglichst kleine Parameter erreicht. Die Suche nach möglichst kleinen Parametern ist darin begründet, dass nach einem Szenario gesucht wird, in dem Restic eine minimale Menge Daten verschlüsseln muss, bis die Grenze der Sicherheitsgarantie erreicht ist. Außerdem hängen m und L für eine feste Datenmenge, die durch Restic verschlüsselt wird, antiproportional voneinander ab. Daher muss herausgearbeitet werden, welcher der beiden Parameter maximiert werden soll. In den ersten Term der Intervallkollisionswahrscheinlichkeit 4.6 geht m quadratisch ein und in den zweiten Term exponentiell wachsend, da gilt $0 < 1 - \frac{L-1}{2^{128}-1} \leq 1$. Der Parameter L wirkt bei beiden Termen nur linear und ist außerdem gibt es in Restic nur vereinzelnde Datenstrukturen, wie Tree-Blobs, die sehr groß werden können (2^{28} Datenblöcke). Damit steigt die Kollisionswahrscheinlichkeit 4.6 deutlich stärker durch die Maximierung von m. Für die Abschätzung der IND$ Sicherheit von Restics AES256-CTR 4.5 wirken sowohl m als auch L linear für den dritten Term, der zusätzlich in die IND$ Sicherheit einfließt.
Bei einer Betrachtung mit Beispielwerten bestätigt sich die Erkenntnis. Würde man also einen maximalen Wert für $L = 2^{28}$ betrachten, müssten $m = 2^{34}$ Datenstrukturen der Größe L verschlüsselt werden, um eine Kollisionswahrscheinlichkeit von 2^{-32} zu erreichen. Das wären 64 EiByte an verschlüsselten Daten. Wohingegen bei einer Maximierung von m der Parameter L minimal werden muss, um die Gesamtdatenmenge so gering wie möglich zu halten. Das ist allerdings auch nur möglich, da m nicht nach oben beschränkt ist. Somit für eine Maximierung von m der Wert $L := 1$ betrachtet, wodurch der zweite Term von 4.6 zu null wird und nur noch die Beschränkung $(1 - \exp(-\frac{3m(m-1)}{2^{129}-2})) \overset{!}{\leq} 2^{-32}$ gelten muss. Dass der zweite Term für $L = 1$ verschwindet, ist damit erklärbar, dass durch $L = 1$ ein Szenario betrachtet wird, indem alle verschlüsselten Klartexte nur einen Datenblock groß sind. Damit ist der einzige Term für die Kollisionswahrscheinlichkeit der IV-Intervalle, die Wahrscheinlichkeit für die Kollision der IVs bei deren Ziehung, da jedes Intervall Länge 1 besitzt. Für $L := 1$ wird bereits nach $m = 2^{47,7}$ verschlüsselten Datenblöcken eine Kollisionswahrscheinlichkeit zwischen 2^{-33} und 2^{-32} erreicht. Das entspricht nur einer Datenmenge von ungefähr 3.327 TiByte verschlüsselter Daten.
Setzt man nun die Wert $L := 1$ und $m := 2^{47,7}$ in 4.5 ein, erhält man eine Abschätzung von $2^{-31} + Adv^{PRP}_{AES256}(\mathcal{B})$ für die IND$ Sicherheit von Restics AES256-CTR Implementierung. Damit kann sich aus der Grenze von 2^{-32} für die Kollisionswahrscheinlichkeit der IV-Intervalle von Restics AES256-CTR eine schöne Schranke für die IND$ Sicherheit von Restics AES256-CTR herleiten lassen. Es folgt die Abschätzung 4.7 für die IND$ Sicherheit von Restics AES256-CTR, wobei die Parameter m und L durch einen Parameter q ersetzt wurden, die Anzahl der mit dem gleichen Schlüssel verschlüsselten Datenblöcke darstellt.

$$Adv^{IND\$}_{CTR[AES256]}(\mathcal{A}) \leq Adv^{PRP}_{AES256}(\mathcal{B}) + \frac{q^2}{2^{129}-2} + (1 - \exp(-\frac{3q(q-1)}{2^{129}-2})) \tag{4.7}$$

Damit konnte mit Abschätzung 4.7 eine Abschätzung konstruiert werden, die nur noch von der Anzahl verschlüsselter Datenblöcke abhängt, egal aus welcher Datenstruktur diese Datenblöcke stammen. Das ist wichtig, um die IND$ Sicherheit für Restic abschätzen zu können, da es keine einheitlichen Größen für Restics Datenstrukturen gibt, mit denen man m und L nachvollziehbar festlegen könnte.
Solang gilt $Adv^{IND\$}_{CTR[AES256]}(\mathcal{A}) \overset{!}{\leq} Adv^{PRP}_{AES256}(\mathcal{B}) + 2^{-31}$, wird unter der Annahme, dass AES256 nur mit vernachlässigbarer Wahrscheinlichkeit von einer echten Zufallspermutation unterscheidbar ist, von einer vernachlässigbaren Erfolgswahrscheinlichkeit für Angreifer $\mathcal{A}$ gesprochen. Außerdem wird in dieser Masterarbeit genau dann davon gesprochen, dass AES256-CTR IND$ sicher ist.

4.3.3. Worst-Case-Szenarios für die Verwendung von Restics AES256-CTR

Die Abschätzung 4.7 hängt nur noch von q ab. Damit ist ein Worst-Case-Szenario ein Szenario, bei dem die verschlüsselten Daten möglichst groß sind und möglichst viele Datenblöcke enthalten.
Die Anzahl der Einträge in den Index-Datenstrukturen und den Pack-Headern steigt je mehr Data-Blobs es gibt. Damit steigt auch die Anzahl der verschlüsselten Datenblöcke pro Pack-Header und die Anzahl der verschlüsselten Index-Datenstrukturen. Daher muss in einem Worst-Case-Szenario auch davon ausgegangen werden, dass Restic nur Data-Blobs minimaler Größe erzeugt, um möglichst viele Data-Blobs zu erzeugen. Data-Blobs können für Dateien, die größer 512 KiByte sind, minimal 512 KiByte große werden (siehe Tabelle 2.6). In der Theorie ist es allerdings möglich, dass jeder Datenblock der Benutzerdaten in einem eigenen Data-Blob gespeichert wird, wenn jeder Datenblock zu einer eigenen Datei gehört. Das heißt, in der Theorie ist die minimale Größe für einen Data-Blob 1 Datenblock (16 Byte). In Einsatzszenario 1 werden bei jedem Backup 4 TiByte von Benutzerdaten als Data-Blobs verarbeitet. Wäre jeder Data-Blob nur 16 Byte groß, würden bei jedem Backup 63 TiByte an Index-Datenstrukturen erzeugt werden und die Grenze von 2^{-32} wäre schon nach 46 Tagen erreicht. Die Wahrscheinlichkeit, dass jeder Data-Blob bei jedem Backup nur 16 Byte groß ist, ist bei einer Benutzerdaten-Größenordnung im Gibibyte-Bereich oder mehr nicht mehr realistisches. Diese Masterarbeit betrachtet für die informationstheoretische Sicherheit von AES256-CTR Data-Blobs mit einer minimalen Größe von 16 KiByte. Diese Größe ist realistischer und sie reicht aus, damit die Index-Datenstrukturen keinen einen überproportionalen Einfluss auf die Größe der verschlüsselten Daten haben. So bildet diese minimale Größe jedes Data-Blobs eine Überabschätzung, mit der eine relevante Aussage über die Erfolgswahrscheinlichkeit des Angreifers getroffen werden kann.
Außerdem wird die untere Abschätzung für die Komprimierungsrate verwendet, um möglichst viele Datenblöcke aus möglichst wenig Benutzerdaten zu erhalten.

4.3.4. Analyse der Einsatzszenarien

In diesem Kapitel werden die realistischen Einsatzszenarien aus Kapitel 4.2 für die festgelegte Schranke von 2^{-32} für die Kollisionswahrscheinlichkeit und 2^{-31} für die IND$ Sicherheit von Restics AES256-CTR analysiert. Es wird mit den Worst-Case Parametern aus Kapitel 4.3.3 die minimale Zeitspanne bestimmt, nach der die festgelegten Schranken für die Sicherheit von Restics AES256-CTR erreicht sind. Aus dieser Zeitspanne kann auch eine minimale Grenze für die Benutzerdaten bestimmt werden, von denen Backups erzeugt werden können, bevor die Sicherheit von AES256-CTR im Worst-Case nicht mehr gewährleistet werden kann. Zur Größenbetrachtung der einzelnen Datenstrukturen werden die Werte aus Tabelle 2.6 verwendet.
Damit die Abschätzung 4.6 gilt und damit die IND$ Erfolgswahrscheinlichkeit 4.7 für einen Angreifer $\mathcal{A}$ nicht größer als 2^{-31} wird, dürfen m und q höchstens $2^{47,7}$ groß sein, während $L := 1$ gilt. Das heißt der Angreifer $\mathcal{A}$ darf sich höchstens $2^{47,7}$ Datenblöcke verschlüsseln lassen. $2^{47,7}$ Datenblöcke entspricht bei einer Blockgröße von 16 Byte abgerundet 3.326 TiByte an eventuell komprimierten und verschlüsselten Daten.

Analyse von Einsatzsszenario 1:
Bei Einsatzszenario 1 werden bei jedem *backup* Aufruf 4 TiByte Benutzerdaten verschlüsselt. Das entspricht 2^{28} Data-Blobs mit je 2^{14} Byte. Damit ergibt sich ein Komprimierungsoverhead pro Data-Blob von 2^5 Byte. Ein solcher komprimierter Data-Blob besteht aus $2^{10} + 2^2$ Datenblöcken. Die maximale Größe eines einzelnen Tree-Blobs ist 4 GiByte. Die Größe eines Tree-Blobs steigt durch die Anzahl der Node-Einträge. Ein 4 GiByte großer Tree-Blob könnte bereits Node-Einträge für alle neuen 2^{28} Data-Blobs enthalten. Dieser eine Tree-Blob wäre Teil eines Packs und würde einen Eintrag im Index erhalten. Restic würde jedoch mehr Datenblöcke erzeugen, wenn jeder Data-Blob in einem eigenen Tree-Blob gespeichert wird. Das würde bedeuten, dass jede Datei, von der ein Backup erzeugt wird, nur aus einem Data-Blob besteht und jede Datei an einem einzigartigen Datei-Pfad auf S_{Restic} gespeichert ist. Ein komprimierter Tree-Blob, der nur einen Data-Blob enthält, besitzt eine maximale Größe von 499 Byte. Damit ergeben sich $32 = 2^5$ Datenblöcke pro Tree-Blob. Bei Pack-Datenstrukturen tragen nur die Pack-Header etwas zu der Menge verschlüsselter Daten bei, da die Größe der Blobs bereits in die Gesamtbetrachtung eingeflossen ist. Eine Pack-Datenstruktur kann maximal 128 MiByte an verschlüsselten und authentifizierten Blobs enthalten. Außerdem kann ein Pack-Header maximal 16 MiByte groß werden, wobei jeder Blob des Packs maximal 41 Byte zu dem Pack-Header hinzufügt. Damit enthält eine Pack-Datenstruktur 8.160 Data-Blobs und 252.764 Tree-Blobs. Es werden insgesamt 32.897 Packs für Data-Blobs und 1.063 Packs für Tree-Blobs benötigt. Der komprimierte Pack-Header eines Packs für Data-Blobs wird 334.656 Byte groß. Der unkomprimierte Pack-Header eines Packs für Tree-Blobs wird $252.764 \cdot 41$ Byte groß und besitzt 2.560 Byte Komprimierungsoverhead. Eine Index-Datenstruktur kann maximal Informationen für 50.000 Blobs beinhalten und hat dabei eine maximale unkomprimierte Größe von 12 MiByte. Für alle Data-Blobs und Tree-Blobs zusammen werden 10.738 Index-Datenstrukturen benötigt. Jede komprimierte Index-Datenstruktur hat maximal 786.624 Datenblöcke. Außerdem wird pro *backup* Aufruf genau eine Lock-Datenstruktur mit einer maximalen komprimierten Größe von 262 Byte

Datenstrukturen	Datengröße	Datenblöcke
Data-Blobs	4 TiByte + 1 GiByte	$2^{38} + 2^{26}$
Tree-Blobs	~ 125 GiByte	$\sim 2^{33}$
Pack-Headers	~ 21 GiByte	$\sim 2^{31}$
Indices	~ 126 GiByte	$\sim 2^{33}$
Locks	262 Byte	$\sim 2^{5}$
Snapshots	~ 2.049 MiByte	$\sim 2^{28}$

Table 4.1.: Größe der erzeugten Daten und Datenblöcke pro Tag in Einsatzszenario 1

Datenstrukturen	Datengröße	Datenblöcke
Data-Blobs	~ 385 GiByte	$48 \cdot (2^{29} + 2^{20})$
Tree-Blobs	~ 12 GiByte	$\sim 2^{30}$
Pack-Headers	~ 2 GiByte	$\sim 2^{27}$
Indices	12 GiByte	$\sim 2^{30}$
Locks	~ 13 KiByte	$\sim 2^{10}$
Snapshots	~ 3.073 MiByte	$\sim 2^{28}$

Table 4.2.: Größe der erzeugten Daten und Datenblöcke pro Tag Einsatzszenario 2

und eine Snapshot-Datenstruktur mit einer maximalen komprimierten Größe von 1 KiByte + 2 GiByte + 513 KiByte erstellt. Damit ergibt sich pro Tag in Einsatzszenario 1 die in Tabelle 4.1 aufgelistete aufgerundete Menge von neuen Daten und Datenblöcken.

Nach 777 Tagen wurden zum ersten Mal mehr als $2^{47,7}$ Datenblöcken verschlüsselt. Für Einsatzszenario 1 hat der Angreifer direkt Zugriff auf ein initiales Repository mit Größe 40 TiByte. Damit braucht der Angreifer weniger Tage, um die Grenze von $2^{47,7}$ Datenblöcken zu erreichen. In 13 Tagen werden bereits mehr als 40 TiByte reiner Benutzerdaten verschlüsselt und an den Angreifer geschickt. Daher kann man sagen, dass in Einsatzszenario 1 nach 764 Tagen die kritische Grenze von $2^{47,7}$ verschlüsselten Datenblöcken erreicht ist und die Vertraulichkeit von Restic nicht mehr gewährleistet ist.

Analyse von Einsatzszenario 2:
In Einsatzszenario 2 wird 48-mal pro Tag der *backup* Befehl für jeweils 8 GiByte Benutzerdaten aufgerufen. Die aufgerundete Menge von neuen Daten und Datenblöcken pro Tag, die verschlüsselt werden, befinden sich in Tabelle 4.2

Nach 8.059 Tagen wurden zum ersten Mal mehr als $2^{47,7}$ Datenblöcken verschlüsselt. Für Einsatzszenario 2 hat der Angreifer direkt Zugriff auf ein initiales Repository mit Größe 8 TiByte. Damit braucht der Angreifer weniger Tage, um die Grenze von $2^{47,7}$ Datenblöcken zu erreichen. In 22 Tagen werden bereits mehr als 8 TiByte reiner Benutzerdaten verschlüsselt und an den Angreifer geschickt. Daher kann man sagen, dass in Einsatzszenario 2 nach 8.037 Tagen die kritische Grenze von $2^{47,7}$ verschlüsselten Datenblöcken erreicht ist und die Vertraulichkeit von Restic kann nicht mehr gewährleistet werden.

4.3.5. Fazit

Die Data-Blobs stellen die reinen Benutzerdaten dar, von denen ein Backup erzeugt wird. Alle anderen Datenstrukturen sind von Restic hinzugefügte Daten zur Verwaltung des Repositorys. In Einsatzszenario 1 sieht man, dass die Data-Blobs 93,7% der zu verschlüsselnden Daten ausmachen. Bei Einsatzszenario 2 bestehen die verschlüsselten Daten nur noch aus 93% Data-Blobs. Trotz der Überabschätzung mit einer geringen Data-Blob Größe von 16 KiByte wirken sich die von Restic erzeugten Daten kaum auf die Größe eines Repositorys aus und damit ist ihr Beitrag zur Erhöhung der Angriffswahrscheinlichkeit sehr gering. Die Angriffswahrscheinlichkeit wird also maßgeblich durch die Größe der Benutzerdaten bestimmt, von denen ein Backup erstellt wird. Außerdem verhält sich die Größe der von Restic hinzugefügten Daten antiproportional zu der Größer der betrachteten Data-Blobs. Je mehr Data-Blobs für die gleiche Menge Benutzerdaten erzeugt werden, desto mehr Metadaten speichert Restic, um die Data-Blobs in Zukunft zu verwalten. Wären alle Data-Blobs beispielsweise nur 1 Datenblock (16 Byte) groß, würde Restic dem Repository Daten hinzufügen, die mehr als das 10 Fache der Benutzerdaten sind. Das ist jedoch kein realistisches Szenario für Backupdaten im dreistelligen bis vierstelligen Tibibyte Bereich.
In Einsatzszenario 1 dauert es 764 Tage bis die obere Schranke 4.7 für die IND$ Sicherheit von Restics AES256-CTR erreicht wird. Ein Angreifer müsste in diesem Szenario also unbemerkt mehr als 2 Jahre Zugriff auf S_{Backup} haben. In Einsatzszenario 2 dauert es 8.037 Tage bis die obere Schranke 4.7 erreicht wird. Das entspricht ungefähr 22 Jahren. In beiden Szenarien werden über den jeweiligen Zeitraum Backups von ungefähr 3.000 TiByte Benutzerdaten durchgeführt.
Solange über die Lebensdauer eines realistischen Restic-Repositorys weniger als 3.000 TiByte Benutzerdaten verschlüsselt verwendet, ist das von Restic eingesetzte Verschlüsselungsverfahren AES256-CTR IND$ sicher und die Vertraulichkeit von Restic gewährleistet.
Die bisherige Betrachtung hat sich hauptsächlich auf den *backup* Befehl von Restic bezogen. Allerdings ist der *prune* Befehl ebenfalls ein Befehl, der eine große Menge Daten mit dem Masterkey verschlüsseln kann und zum Repository hinzufügen kann. Im Worst-Case könnte der *prune* Befehl das gesamte Repository neu verschlüsseln. Mit dieser Abschätzung kann der *prune* Befehl in die bisherigen Erkenntnisse integriert werden, wobei $size(R)$ die aufsummierte Größe aller Datenstrukturen des gerade betrachteten Repositorys R bezeichnet. Wird der *prune* Befehl in einem Szenario ausgeführt, wird dieser Ausführung so behandelt, als ob ein *backup* Befehl ausgeführt worden wäre, der dem Repository die Menge $size(R)$ verschlüsselter Daten hinzufügt. Der *restore* Befehl fügt dem Repository keine Daten hinzu und trägt damit auch nichts zur Erfolgswahrscheinlichkeit eines Angreifers auf die IND$ Sicherheit von Restics AES256-CTR bei. Es können in den Einsatzszenarien also beliebig viele *restore* Befehle ausgeführt werden, ohne die Vertraulichkeitsgarantie zu verändern.
Des Weiteren muss erwähnt werden, dass man Szenarien konstruieren kann, in denen eine beliebige Anzahl Tree-Blobs durch einen einzigen *backup* Befehl erzeugt wird. In Restic wird für jedes Verzeichnis, das Teil eines Pfads zwischen einem Target-Pfad und einer Datei in einer Tree-Blob Datenstruktur gespeichert. Damit kann ein System beliebig viele Verzeichnisse auf dem Pfad zu einer Datei einfügen, wodurch die Anzahl der Tree-Blobs linear zur Anzahl hinzugefügter Verzeichnisse steigt. Bisher wurden nur Data-Blobs, also Dateiinhalte,

als Benutzerdaten bezeichnet. Strenggenommen enthalten Tree-Blobs ebenfalls Daten des Benutzer, da sie Verzeichnisnamen und Modifikationszeitpunkte speichern. In der Praxis ist die Größe der Tree-Blobs jedoch nicht ausschlaggebend für die Sicherheitsschranke von $Adv^{IND\$}_{CTR[AES256]}$. Für Einsatzszenario 2 wurden beispielsweise über 500.000 Tree-Blobs und damit auch über 500.000 Verzeichnisse pro *backup* Befehl betrachtet.

4.4. Integrität

Das kryptografische Verfahren, das von Restic eingesetzt wird, um die Sicherheitseigenschaft Integrität zu gewährleisten, ist ein Poly1305-AES128 Algorithmus. Dieses Kapitel betrachtet die Informationstheoretische Sicherheit des von Restic verwendeten Poly1305-AES128 in den beschriebenen Einsatzszenarien. Es wird die Obergrenzen der Datenmenge bestimmt, von denen Restic ein Backup erstellen darf, ohne die Sicherheit von Poly1305-AES128 zu gefährden.
Es wird eine akzeptable Obergrenze für die Erfolgswahrscheinlichkeit eines Angreifers bestimmt.
Außerdem werden Worst-Case-Bedingungen für die Einsatzszenarien in Bezug auf Poly1305-AES128 herausgearbeitet und die vorgestellten Einsatzszenarien aus Kapitel 4.2 untersucht. Damit wird eine minimale obere Schranke der Datenmenge bestimmt, von der Restic Backups erstellen darf, ohne die Sicherheit von Poly1305-AES128 zu gefährden. Außerdem wird bestimmt nach welcher Zeitspanne in den Einsatzszenarien diese Schranke erreicht ist.

4.4.1. EUF-CMA Sicherheit

In diesem Kapitel wird eine Abschätzung für die Erfolgswahrscheinlichkeit eines Angreifers $\mathcal{A}$ auf die EUF-CMA Sicherheit von Restics Poly1305-AES128 Implementierung aufgestellt. [2] kommt zu der Abschätzung 4.8 für die Wahrscheinlichkeit, dass ein Angreifer mindestens einen gültigen gefälschten MAC erzeugen kann. Bei der Abschätzung 4.8 konnte der Angreifer für den gleichen Schlüssel k_{Poly} genau C authentifizierte Nachrichten beobachten und D Fälschungsversuche durchführen. Außerdem ist L bei dieser Abschätzung die Anzahl Datenblöcke, aus der jede authentifizierte oder gefälschte Nachricht besteht und δ ist die Wahrscheinlichkeit $Adv^{PRP}_{AES128}(\mathcal{B})$.

$$\delta + \frac{(1 - C/2^{128})^{-(C+1)/2} 8D \cdot L}{2^{106}} \tag{4.8}$$

Die Abschätzung 4.8 gilt laut [2] nur, solange eine von Poly1305-AES128 verwendete Nonce nicht wiederverwendet wird. Das folgende Zitat aus [2] stellt klar, unter welchen Bedingungen die Abschätzung 4.8 für die Wahrscheinlichkeit gilt, dass ein Angreifer mindestens eine gültige Fälschung erzeugen konnte.

"It is true even if the attacker can see all the authenticated messages sent by the sender. It is true even if the attacker can see whether the receiver accepts a forgery. It is true even if the attacker can influence the sender's choice of messages and unique nonces. (But it is not true if the nonce-uniqueness rule is violated.)"

Die Abschätzung 4.8 gilt für alle Werte D und damit insbesondere auch für $D := 1$. Für $D := 1$ und einen Angreifer, der die Nachrichten wählen darf, zu denen MACs erstellt werden, lässt sich aus der Multi-Challenge Abschätzung 4.8 die Abschätzung 4.9 für die EUF-CMA Sicherheit von Poly1305-AES128 konstruieren.

$$Adv^{EUF-CMA}_{Poly[AES128]} \leq Adv^{PRP}_{AES128}(\mathcal{B}) + \frac{(1 - C/2^{128})^{-(C+1)/2} 8 \cdot L}{2^{106}} \tag{4.9}$$

In Restic wird Poly1305-AES128 nur benutzt, um MACs für Chiffrate zu erzeugen, die vorher mit Restics AES256-CTR erzeugt wurden. Daher besitzt jedes Chiffrat zu dem ein MAC berechnet wird bereits einen zufällig gleichverteilt gezogenen IV aus dem Raum $\{0,1\}^{128} \setminus 0^{128}$. Restic verwendet den IV eines Chiffrats ebenfalls als Nonce für Poly1305-AES128. In Restic wird für AES256-CTR und das AES128 Verfahren von Poly1305 ein anderer Schlüssel verwendet, wodurch die Verwendung von AES mit der gleichen Nonce für den ersten Chiffratblock und den MAC zu keiner Abhängigkeit führt. Allerdings verschlechtert sich die EUF-CMA Sicherheit von Restics Poly1305-AES128, durch die mögliche IV-Kollision bei der Berechnung zweier MACs und muss in die Erfolgswahrscheinlichkeit eines Angreifers berücksichtigt werden.

Das Problem der IV-Kollisionen:
Bevor betrachtet wird, wie die Abschätzung 4.9 an Restics Poly1305-AES128 Umsetzung angepasst wird, wird betrachtet, warum eine Kollision der IVs auch für Poly1305 problematisch wäre. Zur Erinnerung sei gesagt, dass der Poly1305-AES128 Schlüssel aus zwei 16 Byte großen Schlüsseln besteht. Ein Schlüssel k_{Poly} wird für die Funktion $Poly_{k_{Poly}}(C)$ über dem Chiffrat C verwendet und der zweite Schlüssel k_{AES128} wird für $AES_{k_{AES128}}(IV)$ mit dem zu C gehörigen IV verwendet. Der MAC für C ergibt sich dann durch $MAC := (Poly_{k_{Poly}}(C) + AES_{k_{AES128}}(IV)) \bmod 2^{128}$.
Kommt es zu einer IV-Kollision für zwei generierte MACs mit den gleichen Poly1305-AES128 Schlüsseln, ist das eine Verletzung der Sicherheit von Poly1305-AES128. Ein Angreifer kann durch die Differenz der beiden MACs den AES-Term wegkürzen und erhält eine Differenz der beiden Poly-Terme. Da die Funktion $Poly$ ein Polynom berechnet ist auch die Differenz der beiden Poly-Terme ein Polynom, wodurch der geheime Schlüssel k_{Poly} rekonstruiert werden kann. Ein Angreifer kann also mit einer einzigen IV-Kollision den geheimen Poly1305 Schlüssel k_{Poly} berechnen und ist damit in der Lage zu jedem Bytestring D den Wert $Poly_{k_{Poly}}(D)$ zu berechnen (siehe Kapitel 2.2.3). Damit kann der Angreifer den Wert $AES128(N)$ aus einem bereits vorhandenen MAC rekonstruieren und ist in der Lage einen MAC für mit der Nonce N zu fälschen.

Anpassung der Abschätzung 4.9 an Restics Poly1305-AES128:
Zusätzlich zur EUF-CMA Sicherheit 4.9 von Poly1305-AES128 ohne Nonce-Wiederholung zählt bei Restic ebenfalls die IV-Kollisionswahrscheinlichkeit zur Erfolgswahrscheinlichkeit eines Angreifers auf die EUF-CMA Sicherheit von Restics Poly1305-AES128. Die Wahrscheinlichkeit, dass es zu einer IV-Kollision bei m gezogenen IVs kommt, lässt sich durch die Birthday-Boundary für Restics IV-Raum der Größe $2^{128} - 1$ abschätzen mit $\frac{m^2}{2^{128+1}-2}$.
Wie für die Vertraulichkeit sollte für eine akzeptable Erfolgswahrscheinlichkeit eines Angreifers auf die Integrität von Restic die Kollisionswahrscheinlichkeit der IVs weit unterhalb der Birthday-Boundary liegen. Das heißt der Angreifer darf weit weniger als 2^{64} mit dem gleichen Schlüssel erstellte MACs beobachten, wodurch gilt $C \leq 2^{64}$. Das Paper [2] stellt genau für den Fall, dass gilt $C \leq 2^{64}$, die vereinfachte Abschätzung $\frac{(1-C/2^{128})^{-(C+1)/2}8D\cdot L}{2^{106}} \leq \frac{14\cdot D\cdot L}{2^{106}}$ auf.
Somit lässt sich die Abschätzung 4.10 für die Erfolgswahrscheinlichkeit eines Angreifers $\mathcal{A}$ auf die EUF-CMA Sicherheit von Restics Poly1305-AES128 treffen, wobei m die Anzahl der gezogenen IVs ist und L die Anzahl der Datenblöcke aus der jeder Bytestring besteht, zu dem $\mathcal{A}$ einen gültigen MAC sieht oder einen MAC gefälscht hat.

$$Adv^{EUF-CMA}_{Poly[AES128]} \leq Adv^{PRP}_{AES128}(\mathcal{B}) + \frac{14 \cdot L}{2^{106}} + \frac{m^2}{2^{128+1} - 2} \tag{4.10}$$

Der erste Term der Abschätzung 4.10 lässt sich durch einen einfachen Maximalwert abschätzen. L bezeichnet in dem Term $\frac{14\cdot L}{2^{106}}$ die Anzahl der Datenblöcke, aus denen alle Nachrichten bestehen, zu denen der Angreifer $\mathcal{A}$ sich über ein Orakel im EUF-CMA Spiel MACs erstellen lassen kann. Außerdem sollte die Nachricht für die $\mathcal{A}$ in der Challenge-Phase des EUF-CMA Spiels einen MAC fälscht aus L Datenblöcken bestehen. In Restic werden MACs immer für verschlüsselte Datenstrukturen erstellt. Damit kann L nach oben abgeschätzt werden mit der größten Restic-Datenstruktur. Laut Tabelle 2.6 ist die größte Restic-Datenstruktur ein Tree-Blob bestehend aus 2^{28} Datenblöcken. Damit gilt für Restic die Abschätzung $\frac{14\cdot L}{2^{106}} \leq \frac{14\cdot 2^{28}}{2^{106}} \leq \frac{2^{32}}{2^{106}} = 2^{-74}$.
Unter Verwendung der oberen Schranke für L ergibt sich die Abschätzung 4.11 für die Erfolgswahrscheinlichkeit eines Angreifers $\mathcal{A}$ auf die EUF-CMA Sicherheit von Restics Poly1305-AES128 Implementierung.

$$Adv^{EUF-CMA}_{Poly[AES128]}(\mathcal{A}) \leq Adv^{PRP}_{AES128}(\mathcal{B}) + \frac{m^2}{2^{129} - 2} + 2^{-74} \tag{4.11}$$

4.4.2. Obergrenze für EUF-CMA Sicherheit

In diesem Kapitel wird eine Grenze für die Erfolgswahrscheinlichkeit eines Angreifers $\mathcal{A}$ auf die EUF-CMA Sicherheit von Restics Poly1305-AES128 aufgestellt. Dazu kann wie in Kapitel 4.3.2 der NIST Standard SP-800-38D verwendet werden. Der Standard besagt, dass bei der Verwendung von GCM die Wahrscheinlichkeit für eine IV-Kollsion maximal den Wert 2^{-32} annehmen darf. Wie auch bei Poly1305 folgt auch die MAC-Berechnung von GCM der Wegman-Cater-Struktur mit $MAC_C := GHASH(C) \oplus AES(IV)$. Bei GCM kann ein Angreifer bei einer IV-Kollision den GHASH-Teil der MACs isolieren, genauso wie es für

Poly1305-AES128 der Fall ist mit der Funktion *Poly*. Auch in GCM ist es einem Angreifer dadurch möglich den Schlüssel für die Funktion *GHASH* zu rekonstruieren, wodurch der Angreifer in der Lage ist beliebige MACs zu fälschen. Damit ist für GCM eine IV-Kollision genauso gefährlich, wie für Poly1305-AES128.
Also kann die Obergrenze der IV-Kollisionswahrscheinlichkeit von GCM aus NIST Standard SP-800-38D [3] auch auf die IV-Kollisionswahrscheinlichkeit für Restics Poly1305-AES128 übertragen werden. Es wird gefordert, dass die Wahrscheinlichkeit für eine IV-Kollision maximal 2^{-32} ist. Damit ergibt sich die Beschränkung 4.12 für eine akzeptable IV-Kollisionswahrscheinlichkeit bei Restics Poly1305-AES128, wobei m die Anzahl der gezogenen IVs ist. Für 2^{-32} ergibt sich für m ein maximaler Wert zwischen 2^{48} und 2^{49} gezogenen IVs. Aus diesem Grund wird festgelegt, dass Restic mit dem gleichen Schlüssel niemals mehr als 2^{48} MACs erzeugen darf und damit niemals mehr als 2^{48} Datenstrukturen authentifizieren darf, um die EUF-CMA Sicherheit von Restics Poly1305-AES128 zu gewährleisten.

$$\frac{m^2}{2^{129}-2} \overset{!}{\leq} 2^{-32} \tag{4.12}$$

Damit wurde eine obere Schranke 4.12 gefunden, mit der die maximale akzeptable Erfolgswahrscheinlichkeit eines Angreifers $\mathcal{A}$ auf die EUF-CMA Sicherheit von Restics Poly1305-AES128 bestimmt werden kann.
Für eine schönere Betrachtung wird 2^{-74} aus Abschätzung 4.11 ebenfalls mit 2^{-32} abgeschätzt. Damit ergibt sich folgende obere Schranke für die EUF-CMA Sicherheit von Restics EUF-CMA Implementierung. Solang gilt $Adv^{EUF-CMA}_{Poly[AES128]}(\mathcal{A}) \overset{!}{\leq} Adv^{PRP}_{AES128}(\mathcal{B}) + 2^{-31}$, wird unter der Annahme, dass AES128 nur mit vernachlässigbarer Wahrscheinlichkeit von einer echten Zufallspermutation unterscheidbar ist, von einer vernachlässigbaren Erfolgswahrscheinlichkeit für Angreifer $\mathcal{A}$ gesprochen. Insbesondere folgt daraus, dass maximal 2^{48} MACs durch Restic für dasselbe Repository mit dem gleichen Masterkey erzeugt werden dürfen, bevor die EUF-CMA Sicherheit von Restics Poly1305-AES128 verletzt ist. Außerdem wird in dieser Masterarbeit genau dann davon gesprochen, dass Poly1305-AES128 EUF-CMA sicher ist.

Betrachtung von MC-EUF-CMA:
Für die Betrachtung Restics ist die EUF-CMA Sicherheit eventuell zu schwach, da ein Angreifer nur eine einzige Fälschung vornehmen darf und diese Fälschung muss direkt verifiziert werden können, damit der Angreifer das Sicherheitsspiel gewinnt. Bei einem realistischen Einsatzszenario von Restic, ist es einem Angreifer jedoch möglich mehrere Fälschungen verifizieren zu lassen. Daher ist eine Betrachtung der MC-EUF-CMA Sicherheit von Restics Poly1305-AES128 ebenfalls interessant.
In Restic wird für jede vom S_{Backup} erhaltene verschlüsselte und authentifizierte Datenstruktur der MAC dieser Datenstruktur verifiziert. Kann Restic den MAC einer erhaltenen Datenstruktur nicht verifizieren, weil das Chiffrat der Datenstruktur verändert wurde oder der MAC verändert wurde, bricht Restic den Restic-Befehl, der gerade ausgeführt wird ab. Damit kann ein realer Angreifer, der eine Fälschung verifizieren lassen möchte, nur einen seiner gefälschten MACs pro Restic-Befehl verifizieren lassen. Diese Aussage gilt zumindest

so lange, bis es dem Angreifer gelungen ist, einen MAC zu fälschen, dessen Verifikation erfolgreich ist. In diesem Fall bemerkt Restic die Fälschung nicht und führt den Restic-Befehl weiter aus.
Da Restic ein komplexes Protokoll ist mit vielen Möglichkeiten Fehlerquellen, deutet der Abbruch eines Restic-Befehls nicht unbedingt auf einen Angreifer hin, der versucht die Integrität von Restic zu attackieren. Es sei dazu gesagt, dass ein Benutzer, der die Log-Ausgaben von Restic überprüft, herausfindet, warum es zu einem Abbruch des Restic-Befehls kam und somit herausfindet, dass ein MAC nicht erfolgreich verifiziert werden konnte. In einem realistischen Einsatzszenario werden das jedoch die wenigsten Benutzer sein und sofern ein Angreifer nicht jeden Restic-Befehl ausnutzt, um einen gefälschten MAC verifizieren zu lassen, ist es realistisch, dass ein normaler Benutzer nicht misstrauisch wird.
Die MC-EUF-CMA Sicherheit unterscheidet sich zur EUF-CMA Sicherheit nur dadurch, dass ein Angreifer in der Challenge-Phase mehrere Fälschungen durchführen kann und gewinnt, sobald eine Fälschung erfolgreich verifiziert werden kann (siehe Kapitel 2.2.3.1). Für Poly1305-AES128 gibt es für die Wahrscheinlichkeit 4.8, dass mindestens einer von D gefälschten MACs erfolgreich verifiziert werden kann. Das ist exakt die Erfolgswahrscheinlichkeit eines Angreifers auf die MC-EUF-CMA Sicherheit von Poly1305-AES128. Für Restics Umsetzung von Poly1305-AES128 würde wie beim EUF-CMA Sicherheitsspiel ebenfalls ein Angreifer gewinnen, sobald es zu einer IV-Kollision kommt. Daher ergibt sich für das MC-EUF-CMA Sicherheitsspiel von Restics Poly1305-AES128 die Abschätzung 4.13 für die Erfolgswahrscheinlichkeit eines Angreifers $\mathcal{A}$.

$$Adv^{MC-EUF-CMA}_{Poly[AES128]}(\mathcal{A}) \leq Adv^{PRP}_{AES128}(\mathcal{B}) + \frac{14 \cdot D \cdot L}{2^{106}} + \frac{m^2}{2^{129} - 2} \tag{4.13}$$

Auch in diesem Fall kann L mit 2^{28} abgeschätzt werden und aus dem NIST Standard [3] folgt, die Beschränkung 4.12. Es gilt nach Kapitel 2.2.3.1 $Adv^{EUF-CMA}_{Poly[AES128]}(\mathcal{A}) \leq Adv^{MC-EUF-CMA}_{Poly[AES128]}(\mathcal{A})$ und für $Adv^{EUF-CMA}_{Poly[AES128]}(\mathcal{A})$ ist die maximal akzeptable Obergrenze $Adv^{PRP}_{AES128}(\mathcal{B}) + 2^{-31}$. Um eine akzeptable Obergrenze für $Adv^{MC-EUF-CMA}_{Poly[AES128]}$ zu finden, wird zunächst betrachtet, wie groß D maximal sein darf, damit ebenfalls gilt $Adv^{MC-EUF-CMA}_{Poly[AES128]} \leq Adv^{PRP}_{AES128}(\mathcal{B}) + 2^{-31}$. Für $D \leq 2^{43}$ gilt auch für die Erfolgswahrscheinlichkeit eines Angreifers $\mathcal{A}$ auf die MC-EUF-CMA Sicherheit von Restics Poly1305-AES128 die gleiche Sicherheitsschranke $Adv^{PRP}_{AES128}(\mathcal{B}) + 2^{-31}$, wie für die EUF-CMA Sicherheit. In der Realität bedeutet $D = 2^{43}$, dass ein Angreifer 2^{43} Fälschungen durchführen müsste, damit mindestens ein gefälschter MAC mit einer Wahrscheinlichkeit von 2^{-31} erfolgreich verifiziert wird. Das bedeutet es müssten 2^{43} Restic-Befehl ausgeführt werden, die einen von $\mathcal{A}$ gefälschten MAC verifizieren. Das sind auch 2^{43} Restic-Befehle, die mit sehr hoher Wahrscheinlichkeit alle fehlschlagen. Spätestens nach so vielen fehlgeschlagenen Befehlen sollte auch dem naivsten Benutzer auffallen, dass etwas mit dem korrumpierten Repository nicht stimmt.
Für ein realistisches Beispiel, kann das Einsatzszenario 2 herangezogen werden, das den Einsatz von Restic bei einem privaten Benutzer darstellt. Es wird alle 30 Minuten ein Restic-Befehl für das gleiche Repository in Einsatzszenario 2 ausgeführt. Damit wurden nach 22 Jahren, nachdem die Grenze der Vertraulichkeitsgarantie für Restics Verschlüsselungsverfahren erreicht ist, weniger als 2^{19} Restic-Befehle ausgeführt.

Somit lässt sich auch die Schranke der EUF-CMA Sicherheit auf eine akzeptable Erfolgswahrscheinlichkeit für das MC-EUF-CMA Sicherheitsspiel von Restics Poly1305-AES128 übertragen. Solang gilt $Adv^{MC-EUF-CMA}_{Poly[AES128]}(\mathcal{A}) \overset{!}{\leq} Adv^{PRP}_{AES128}(\mathcal{B}) + 2^{-31}$, wird unter der Annahme, dass AES128 nur mit vernachlässigbarer Wahrscheinlichkeit von einer echten Zufallspermutation unterscheidbar ist, von einer vernachlässigbaren Erfolgswahrscheinlichkeit für Angreifer $\mathcal{A}$ gesprochen. Insbesondere folgt daraus, dass maximal 2^{43} MACs durch Restic für dasselbe Repository mit dem gleichen Masterkey erzeugt werden dürfen, bevor die MC-EUF-CMA Sicherheit von Restics Poly1305-AES128 verletzt ist. Außerdem folgt daraus, dass maximal 2^{43} durch den Angreifer $\mathcal{A}$ gefälschte MACs von Restic für dasselbe Repository mit dem gleichen Masterkey angefragt und verifiziert werden dürfen, bevor die MC-EUF-CMA Sicherheit von Restics Poly1305-AES128 verletzt ist. Außerdem wird in dieser Masterarbeit genau dann davon gesprochen, dass Poly1305-AES128 MC-EUF-CMA sicher ist.

4.4.3. Worst-Case-Szenario für die Verwendung von Restics Poly1305-AES128

Bei den Worst-Case-Szenarien geht es darum, Bedingungen für Szenarien zu finden, in denen mit einer minimalen Menge von Daten in einer minimalen Zeitspanne die Erfolgswahrscheinlichkeit eines Angreifers maximiert wird. Für die Abschätzung 4.11 ist ein Worst-Case-Szenario ein Szenario, bei dem möglichst viele MACs erstellt werden. Mit einem *backup* Befehl sollten so viele Restic-Datenstrukturen wie möglich erstellt werden, da jeder erstellte MAC zu einer eindeutigen Datenstruktur gehört. Je kleiner die einzelnen Datenstrukturen sind, desto mehr werden für die gleiche Benutzerdatenmenge erstellt. Es ist möglich, dass jeder Data-Blob nur ein Datenblock (16 Byte) groß ist. In der Theorie ist es auch möglich, dass jede Datei nur 1 Byte enthält und damit jeder Data-Blob nur 1 Byte groß ist. Beide Fäll sind für größere Mengen von Daten absolut unrealistisch. Daher wird die ebenfalls unwahrscheinlichen Worst-Case Abschätzung verwendet, dass jede Data-Blob genau 2^{10} Byte groß ist.
Für die Komprimierungsrate bei Data-Blobs und Tree-Blobs wird die untere Abschätzung verwendet, da bei größeren Blobs die Anzahl der Pack-Datenstrukturen steigt und damit mehr Pack-Header existieren, die ebenfalls alle mit einem MAC versehen werden. Für die restlichen Datenstrukturen ist deren Größe irrelevant für die Anzahl erstellter MACs und damit wird für alle Datenstrukturen außer Blobs die Komprimierungsrate mit 11 abgeschätzt.

4.4.4. Analyse der Einsatzszenarien

In Einsatzszenario 1 werden die Benutzerdaten in 2^{32} Data-Blobs mit einer Größe von 2^{10} Byte aufgeteilt. Jeder der Data-Blobs besitzt einen Komprimierungsoverhead von jeweils 2^5 Byte Für jeden Data-Blob wird ein Tree-Blob erstellt, der die maximale komprimierte Größe 499 Byte hat. Eine Pack-Datenstruktur muss mindestens 4 MiByte an verschlüsselten und authentifizierten Blobs enthalten. Damit enthält eine Pack-Datenstruktur entweder

Datenstrukturen	Datengröße	Neu erzeugte MACs
Data-Blobs	4 TiByte + 128 GiByte	2^{32}
Tree-Blobs	~ 2 TiByte	2^{32}
Pack-Headers	8 TiByte	$\sim 2^{21}$
Indices	~ 256 GiByte	$\sim 2^{18}$
Locks	20 Byte	1
Snapshots	186 MiByte	1

Table 4.3.: Größe der erzeugten Datenstrukturen und MACs pro Tag in Einsatzszenario 1

3.972 Data-Blobs oder 7.899 Tree-Blobs. Eine Index-Datenstruktur kann maximal 50.000 Blobs beinhalten. Die Größe einer Index-Datenstruktur wird sonst nur durch deren Alter beschränkt. Wurde eine Index-Datenstruktur von Restic vor mehr als 10 Minuten angelegt, wird sie verschlüsselt, authentifiziert und eine neue Index-Datenstruktur für die weiteren Einträge angelegt. Wenn Restic nicht gerade auf einem sehr langsamen System läuft, ist diese Grenze von 10 Minuten nur sehr unwahrscheinliche zu erreichen. Daher wird die Bedingung von maximal 50.000 Blobs pro Index-Datenstruktur betrachtet. Für alle Data-Blobs und Tree-Blobs zusammen werden nach oben abgeschätzt 2^{18} Index-Datenstrukturen benötigt, deren komprimierte Größe jeweils 1 MiByte beträgt. Außerdem wird pro *backup* Aufruf genau eine Lock-Datenstruktur mit einer komprimierten Größe von 20 Byte und eine Snapshot-Datenstruktur mit einer komprimierten Größe von 186 MiByte erstellt. Damit ergibt sich pro Tag in Einsatzszenario 1 die in Tabelle 4.1 aufgelistete abgerundete Menge von neuen Daten und die aufgerundete Menge von authentifizierten Datenstrukturen.

Nach 1000 Tagen ergibt sich mit der Abschätzung 4.12 für Einsatzszenario 1 immer noch nur eine Erfolgswahrscheinlichkeit von weniger als 2^{-53}. Also auch mit dieser Abschätzung steigt die Erfolgswahrscheinlichkeit des Angreifers nicht merklich an.

4.4.5. MAC$ Sicherheit von Poly1305-AES128

In diesem Kapitel wird die Erfolgswahrscheinlichkeit eines Angreifers auf die MAC$ Sicherheit aus Kapitel 2.2.3.2 von Poly1305-AES128 bestimmt. Dazu wird die Erfolgswahrscheinlichkeit $Adv^{MAC\$}_{Poly1305-AES128}(\mathcal{A})$ eines Angreifers $\mathcal{A}$ für das MAC$ Sicherheitsspiel von Poly1305-AES128 bestimmt.
Poly1305-AES128 erzeugt mithilfe eines 16 Byte großen IVs über einer Nachricht M beliebiger Länge einen 16 Byte großen MAC. Es seien $k_{Poly} \xleftarrow{\$} \mathcal{IV}_{poly}$ und $k_{AES128} \xleftarrow{\$} \{0,1\}^{128}$ zwei uniforme Schlüssel, die jeweils 16 Byte groß sind. $\mathcal{IV}_{poly}$ bezeichnet hierbei den IV-Raum für die 2^{106} möglichen Poly1305 Schlüssel (siehe Kapitel 2.2.3).
Der durch Poly1305-AES128 erzeugte MAC berechnet sich nach Kapitel 2.2.3 wie folgt:

$$MAC_M := (Poly_{k_{Poly}}(M) + AES_{k_{AES128}}(IV)) \bmod 2^{128}$$

In der Welt $\mathcal{W}_0$ (b=0) wird für einen vom Angreifer gewählten Bytestring M ein MAC MAC_M wie oben angegeben ehrlich durch Poly1305-AES128 berechnet und an den Angreifer zurück-

gegeben In der Welt $\mathcal{W}_1$ (b=1) wird ein uniformer Bytestring $MAC_M \xleftarrow{\$} \{0,1\}^{128}$ an den Angreifer $\mathcal{A}$ zurückgegeben.
Für den Beweis werden zwei Zwischenwelten betrachtet, in denen Stück für Stück Teile der MAC-Berechnung von Poly durch Zufall ersetzt werden. Zunächst wird eine Welt $\mathcal{W}_{0.1}$ betrachtet, in der AES128 durch eine echte Zufallsfunktion $\mathcal{F}$ ersetzt wurde mit $MAC_M := Poly_{k_{Poly}}(M) + \mathcal{F}(IV)$. Die Wahrscheinlichkeit, dass ein Angreifer zwischen der Welt $\mathcal{W}_0$ und $\mathcal{W}_{0.1}$ unterscheiden kann, wird durch die Wahrscheinlichkeit $Adv^{PRF}_{AES128}(\mathcal{B})$ beschränkt. $Adv^{PRF}_{AES128}(\mathcal{B})$ ist die Wahrscheinlichkeit, dass AES128 von einer uniform gezogenen echten Zufallsfunktion unterscheidbar ist.
Nun wird der Übergang zu Welt $\mathcal{W}_{0.2}$ betrachtet, in der die Funktionswerte der Zufallsfunktion durch uniform gezogene Werte ersetzt werden und somit unabhängig von den IVs sind. Dadurch, dass der von $\mathcal{F}$ verwendete IV uniform gezogen wurde, ist der aus $\mathcal{F}$ resultierende Wert ebenfalls uniform. Die Uniformität gilt jedoch nur, wenn jeder Funktionswert $F(IV_i)$ einzeln betrachtet wird. Kommt es zu einer Kollision ($IV_i = IV_j$ mit $i \neq j$) zweier IVs, dann gilt $\mathcal{F}(IV_i) = \mathcal{F}(IV_j)$ und damit kommt es zu einer Abhängigkeit aller verwendeten Funktionswerte von $\mathcal{F}$ in $\mathcal{W}_{0.1}$. Damit jedoch die mit $\mathcal{F}$ berechneten MACs aus $\mathcal{W}_{0.1}$ ununterscheidbar sind von uniform gezogenen Werten, müssen alle verwendeten Funktionswerte von $\mathcal{F}$ unabhängig und echt zufällig sein. Der echte Zufall ist per Definition von $\mathcal{F}$ gegeben und die unabhängig auch, sofern es zu keiner Kollision der verwendeten IVs kommt. Damit ist die Wahrscheinlichkeit, dass ein Angreifer $\mathcal{W}_{0.1}$ von der Welt $\mathcal{W}_{0.2}$, in der die Werte der Zufallsfunktion durch echte uniforme Zufallswerte ersetzt werden, unterscheiden kann, durch die Kollisionswahrscheinlichkeit der IVs begrenzt. Da die IVs uniform sind, ist eine Kollision von zwei gezogenen IVs nicht auszuschließen. Diese Kollisionswahrscheinlichkeit wird durch die Birthday-Boundary nach Kapitel 2.2.5 beschränkt mit $\frac{m^2}{2^{128+1}}$, wobei m die Anzahl gezogener IVs und damit auch die Anzahl berechneter MACs ist. Damit ist die Wahrscheinlichkeit, dass ein Angreifer $\mathcal{W}_{0.1}$ von $\mathcal{W}_{0.2}$ unterscheiden kann, durch die Kollisionswahrscheinlichkeit $\frac{m^2}{2^{129}}$ für die gezogenen IVs beschränkt.
Somit wird in $\mathcal{W}_{0.2}$ ein MAC für eine Nachricht M berechnet durch $MAC_M := (Poly_{k_{Poly}}(M) + UB) \bmod 2^{128}$, wobei UB ein uniformer Bytestring der Größe 16 Byte ist. Nun wird für Poly1305-AES128 der Unterschied zwischen $\mathcal{W}_{0.2}$ und $\mathcal{W}_1$, in der ein MAC als uniformer Bytestring gewählt wird, abgeschätzt. Die beiden Welten wären ununterscheidbar, wenn die Werte von $Poly_{k_{Poly}}$ ununterscheidbar von uniformen Bytestrings wären, da die Summe zweier uniformer Bytestrings wieder ein uniformer Bytestring ist. Da für jeden MAC der gleiche Schlüssel k_{Poly} verwendet wird, könnten die einzelnen Werte der Funktion $Poly_{k_{Poly}}$ Abhängigkeiten zueinander aufweisen und damit besitzen die MACs eine Struktur, die ein Angreifer erkennen kann. Mit dem flogenden Zitat aus Paper [2], lässt sich zeigen, dass die Differenzen für einen zufälligen Schlüssel $r := k_{Poly}$ von $H_r := Poly_{k_{Poly}}$ für zwei unterschiedliche Nachrichten fast uniform verteilt sind. Das Paper [2] sagt, dass für einen uniformen Schlüssel r folgende Aussage gilt:

"The crucial property of H_r is that it has small differential probabilities: if g is a 16-byte string, and m, m' are distinct messages of length at most L, then $H_r(m) = H_r(m') + g$ with probability at most $\frac{8 \cdot \lceil L/16 \rceil}{2^{106}}$"

Dabei ist L in dem Zitat die Länge der Nachrichten in Bytes, zu denen ein MAC erstellt wird. Für den weiteren Verlauf dieses Kapitel wird die Definition dieser Masterarbeit für L benutzt, wobei L die Anzahl Datenblöcke ist, aus der ein Bytestring besteht. Mit dieser Definition von L ergibt sich der Term $\frac{8 \cdot L}{2^{106}}$.
Ein Angreifer sieht von der Berechnung nur die Ergebnisse MAC_1 und MAC_2 für zwei Nachrichten M_1 und M_2. Der Angreifer könnte versuchen durch die Differenz $MAC_1 - MAC_2 = (Poly_{k_{Poly}}(M_1) - Poly_{k_{Poly}}(M_2)) + (UB_1 - UB_2)$ eine Struktur in den MACs zu erkennen. Damit sind die Welten $\mathcal{W}_{0.2}$ und $\mathcal{W}_1$ unterscheidbar, wenn es ein Nachrichtenpaar (M_1, M_2) gibt, sodass für ein bestimmtes g gilt $Poly_{k_{Poly}}(M_1) = Poly_{k_{Poly}}(M_2) + g$.
Es gibt nun zwei Möglichkeiten die Wahrscheinlichkeit abzuschätzen, dass bei m berechneten MACs mindestens zwei MACs eine Abhängigkeit aufweisen durch $Poly_{k_{Poly}}(M_1) = Poly_{k_{Poly}}(M_2) + g$ für ein bestimmtes g. Eine konservative Abschätzung mit der Union-Bound, wodurch die Wahrscheinlichkeit für eine Abhängigkeit bei m berechneten MACs zu folgender Abschätzung führt:

$$\frac{m(m-1)}{2} \cdot \frac{8 \cdot L}{2^{106}}$$

Diese Abschätzung geht davon aus, dass jedes MAC-Paar unabhängig voneinander zur Unterscheidbarkeit beiträgt. Diese Abschätzung wächst quadratisch in m und wird durch den kleinen Schlüsselraum für k_{Poly} mit einer Größe von 2^{106} deutlich zu groß. Mit dieser Abschätzung ergibt sich beispielsweise schon für $m = 2^{40}$ und $L = 1$ eine Unterscheidungswahrscheinlichkeit von 2^{-25}.
Da der Angreifer nach der Beobachtung von m erstellten MACs eine einzige Entscheidung trifft, ist das relevante Ereignis nicht, ob es ein beliebiges MAC-Paar gibt, das unterschieden werden kann, sondern ob $\mathcal{A}$ konkret zwischen $\mathcal{W}_{0.2}$ und $\mathcal{W}_1$ unterscheiden kann. Dazu besitzt eine $\mathcal{A}$ eine Unterscheidungsstrategie, die von $\mathcal{A}$ fordert ein festes g festzulegen. Für ein festes g gibt es nur m mögliche MAC-Paare, die $\mathcal{A}$ unterscheiden kann. Damit ergibt sich aus dieser Begründung die folgende Abschätzung, die nur noch linear in m wächst:

$$m \cdot \frac{8 \cdot L}{2^{106}}$$

Diese Abschätzung wird verwendet, um die Gesamtabschätzung **??** für die Erfolgswahrscheinlichkeit eines Angreifers $\mathcal{A}$ auf die MAC$ Sicherheit von Poly1305-AES128 aufzustellen.

$$Adv^{MAC\$}_{Poly[AES128]}(\mathcal{A}) \leq Adv^{PRF}_{AES128}(\mathcal{B}) + \frac{m^2}{2^{129}} + m \cdot \frac{8 \cdot L}{2^{106}} \tag{4.14}$$

5. Sicherheitsmodell und Sicherheitsbeweis

In diesem Kapitel wird der Sicherheitsbeweis für Restics Vertraulichkeit und Integrität durchgeführt. Dazu wird zuerst die Leakage betrachtet, die durch die Ausführung von Restic-Befehlen auf einem Repository entsteht. Leakage bezeichnet die unvermeidlichen Informationen, die ein Beobachter abseits der Chiffrate, erhält.
Im nächsten Schritt wird die Vertraulichkeit von Restic betrachtet. Dazu wird ein Sicherheitsspiel $Exp^{IND\$}_{Restic}$ aufgestellt, mit dem die Vertraulichkeit von Restic bewiesen werden kann. Zunächst wird ein allgemeiner Reduktionsbeweis bezüglich abstrakten kryptografischen Primitiven geführt. Dann werden konkrete Schranken für Restics verwendete Primitive eingesetzt und somit die Vertraulichkeit von Restic parametrisiert.
Dasselbe wird für die Integrität Restics durchgeführt. Zunächst wird ein Sicherheitsspiel $Exp^{EUF-CMA}_{Restic}$ für Restics Integrität vorgestellt. Für dieses Sicherheitsspiel wird ein allgemeiner Reduktionsbeweis geführt und am Ende werden konkrete Sicherheitsschranken für konkrete kryptografische Primitive eingesetzt.

5.1. Leakage

Dieses Kapitel bestimmt die Informationen, die ein Angreifer $\mathcal{A}$ aus der reinen Beobachtung des Repositorys lernen kann, während ein Restic-Befehl ausgeführt wird. Diese Informationen werden Leakage genannt. Dieses Kapitel stellt eine formale Definition für die Leakage der abstrakten Modellierung der Restic-Befehle als Event-Tupel dar. Diese formale Definition wird Leakage-Funktion genannt.
In den daraufhin vorgestellten Sicherheitsspielen kennt $\mathcal{A}$ die Klartexte (Verzeichnissysteme), mit denen ein *backup* Befehl oder *restore* Befehle ausgeführt wird. Genau genommen, darf der Angreifer diese Verzeichnissysteme sogar selbst wählen. Daher wird ebenfalls die Leakage betrachtet, die sich aus der Leakage-Funktion mit und ohne Kenntnisse über die konkreten Befehlsargumente ableiten lässt.
Am Ende werden reale Konsequenzen durch die Leakage herausgearbeitet. Es wird besprochen, was ein realer Angreifer aus der Leakage herleiten kann und wie sie ausgenutzt werden kann, um weitere Informationen über S_{Restic} und dessen Benutzer herauszufinden.

5.1.1. Leakage-Funktion $\mathcal{L}$

Dieses Kapitel definiert die allgemeine Leakage eines Event-Tupel τ für die Ausführung eines Restic-Befehls. Bevor die Leakage definiert werden kann, müssen einige weitere Definitionen betrachtet werden.
Über die Lebensdauer eines Repositorys R werden in der Regel mehr als nur ein Restic-Befehl ausgeführt. Jeder der ausgeführten Restic-Befehle kann nach der abstrakten Modellierung eindeutig durch ein Event-Tupel beschrieben werden. Alle Event-Tupel für Restic-Befehle, die auf R ausgeführt wurden, werden nummeriert mit τ_t, wobei τ_t für den t-ten Restic-Befehl aus R steht. Das i-te Event des t-ten Restic-Befehls wird bezeichnet mit $e_{t,i} \in \tau_t$. e_{max_t} bezeichnet das letzte Event eines Event-Tupels τ_t. Alle Event-Tupel sind aufsteigend nach ihrer zeitlichen Ausführung geordnet nummeriert. Damit ergibt sich für n ausgeführte Restic-Befehle auf R die Menge $\{\tau_1, \ldots, \tau_n\}$ aller Event-Tupel zu diesen Befehlen, wobei τ_1 der erste jemals ausgeführte Restic-Befehl auf R ist. Wenn von ausgeführten Restic-Befehlen gesprochen wird, sind damit immer nur die drei in dieser Masterarbeit betrachteten Restic-Befehle gemeint.

Definition 5.1.1 *$\mathcal{T}_n^R := \{\tau_1, \ldots, \tau_n\}$ ist die Menge aller Event-Tupel für die ersten n Restic-Befehle, die auf dem Repository R ausgeführt wurden.*
$\mathcal{E}_n^R := \bigcup_{t=1}^{n} \bigcup_{i=1}^{max_t} e_{t,i}$ ist die Menge aller Events der ersten n Restic-Befehle, die auf dem Repository R ausgeführt wurden.
$\mathcal{I}(\mathcal{E}_n^R) := \{(t,i) \in \mathbb{N} \times \mathbb{N} \mid e_{t,i} \in \mathcal{E}_{\setminus}^{\mathcal{R}}\}$ ist die Menge aller Indices aller Events aus $\mathbb{E}_n^R$. Jedes $(t,i) \in \mathcal{I}(\mathcal{E}_n^R)$ kann eindeutig einem Event $e_{t,i} \in \mathcal{E}_n^R$ zugeordnet werden. Ein Eintrag in $\mathcal{I}$ wird Event-Index genannt.

Für den weiteren Verlauf dieses Kapitels wird ein festes Repository R betrachtet, auf dem n Restic-Befehle ausgeführt wurden. Um die Leakage zu bestimmen, ist es wichtig eine Referenzfunktion zu definieren, die eindeutig ein Event referenzieren kann, mit dem eine Datenstruktur zu R hinzugefügt wurde.

Definition 5.1.2 (Referenzfunktion) *Die Funktion $ref : \mathcal{I}(\mathcal{E}_n^R) \rightarrow \mathcal{I}(\mathcal{E}_n^R) \cup \{\perp\}$ bildet ein Event-Index (t,i) eines Events $e_{t,i} \in \mathcal{E}_n^R$ auf einen Event-Index (s,j) eines anderen Events $e_{s,j} \in \mathcal{E}_n^R$ ab. Dabei gilt für $e_{s,j}$ folgendes:*

- $op_{s,j} = send$
- $type_{s,j} = type_{t,i}$
- $data_{s,j} = data_{t,i}$
- $s < t$ *oder* $(s = t$ *und* $delay_{s,j} < delay_{t,i})$
- $\forall e_{r,k} \in \mathcal{E}_n^R$ *mit* $e_{r,k} \neq e_{s,j}$ *und* $e_{r,k}$ *erfüllt die gleichen vier vorherigen Eigenschaften gilt:* $r < s$ *oder* $(r = s$ *und* $delay_{r,k} < delay_{s,j})$

Gibt es kein solches $e_{s,j}$, dann gilt $ref(t,i) = \perp$.
Damit referenziert $ref(t,i)$ das letzte Event vor $e_{t,i}$, das die von $e_{t,i}$ verwendete Datenstruktur $data_{t,i}$ zu R hinzugefügt hat.

Alle Datenstrukturen, die von einem Receive-Event verwendet werden, mussten zu einem früheren Zeitpunkt zum Repository *R* durch ein Send-Event hinzugefügt worden sein. Die einzige Ausnahme bilden die Key-Datenstruktur und die Config-Datenstruktur. Diese beiden Datenstrukturen werden beim Erstellen eines Repositorys zum Repository hinzugefügt, bevor der erste Restic-Befehl betrachtet wird. Daher gilt $ref(t,i) \neq \perp$ für ein Event mit $op_{t,i} = receive$, sofern gilt $type_{t,i} \neq Key, Config$.
Keine der drei betrachteten Arten von Restic-Befehlen fügt neue Datenstrukturen vom Typ Key oder $Config$ zum Repository hinzu. Daher ist auch ohne eine Definition der Referenzfunktion für diese Datenstrukturen eindeutig, wann sie zu *R* hinzugefügt wurden.

Definition 5.1.3 (Leakage-Funktion $\mathcal{L}$) *Die Leakage-Funktion $\mathcal{L}_R(\tau)$ beschreibt alle Informationen, die ein Angreifer durch Beobachtung des Repositorys R erhält, während ein Restic-Befehl ausgeführt wird, der in dem Event-Tupel τ resultiert. Dabei werden nur Informationen betrachtet, die der Angreifer erhält, ohne die Sicherheit des zugrundeliegenden Verschlüsselungsverfahren von Restic zu brechen. Die Leakage-Funktion bildet eine Menge von Events auf eine Menge von Leakage-Tupeln ab. Jedes Event $e_{t,i}$ wird auf ein Leakage-Tupel $leakage(e_{t,i})$ abgebildet.*

$$\mathcal{L}(E_n^R) := \bigcup_{t=1}^{n} \bigcup_{i=1}^{max_t} leakage(e_{t,i})$$

Dazu wird die folgende Funktion zur Bestimmung der Leakage eines einzelnen Events definiert:

$$leakage(e_{t,i}) := \begin{cases} (op_{t,i}, delay_{t,i}, type_{t,i}, size(data_{t,i}), packHeaderSize_{t,i}, ref(t,i)) & type_{t,i} = Pack \\ (op_{t,i}, delay_{t,i}, type_{t,i}, size(data_{t,i}), blobInfo_{t,i}, ref(t,i)) & type_{t,i} = BlobChunk \\ (op_{t,i}, delay_{t,i}, type_{t,i}, size(data_{t,i}), ref(t,i), key_{t,i}) & type_{t,i} = Key \\ (op_{t,i}, delay_{t,i}, type_{t,i}, size(data_{t,i}), ref(t,i)) & sonst \end{cases}$$

$packHeaderSize_{t,i}$ ist die Größe des Pack-Headers einer Pack-Datenstruktur in Bytes (siehe Kapitel 2.3.4.9). $packHeaderSize_{t,i}$ wird unverschlüsselt ans Ende einer Pack-Datenstruktur konkateniert und ist damit ein unverschlüsselter Teil von $data_{t,i}$ und gehört damit auch zur Leakage.
$key_{t,i}$ ist dabei nicht die Key-Datenstruktur, sondern die Leakage, die sich aus einer Key-Datenstruktur ergibt, da diese nicht verschlüsselt im Repository gespeichert ist. Es gilt somit $key_{t,i} := (hostname, username, kdf, n, r, p, time, salt, size(data))$. Für den data Eintrag gilt das gleiche, wie für die $data_{t,i}$ Einträge der Events. Daher ist ebenfalls nur die Länge des Eintrags Teil der Leakage.

Definition 5.1.4 (Leakage-Funktion Notation) *Für die Leakage eines Event-Tupels τ bezüglich Repository R wird $\mathcal{L}_R(\tau)$ geschrieben. Es sei o.B.d.A $\tau = \tau_t$ der t-te Restic-Befehl seit der Erstellung des Repositorys R. Damit ist $\mathcal{L}_R(\tau)$ eigentlich eine Umschreibung für die Leakage $\mathcal{L}_{E_{t-1}^R}(\tau)$*

des Event-Tupels τ unter Berücksichtigung aller früheren Restic-Befehle $\tau_1, \ldots, \tau_{t-1}$ auf R. Diese Klarstellung wird benötigt, um die Korrektheit der Referenzfunktion sicherzustellen, die jedes Event eines früheren Event-Tupels referenzieren kann.

In dem betrachteten Angriffszenario dieser Masterarbeit wird immer ein Angreifer betrachtet, der vollen Lesezugriff auf S_{Backup} und damit Restics Repositorys hat. Die Leakage-Funktion $\mathcal{L}$ beschreibt, welche Informationen diesem Angreifer durch die Beobachtung eines Repositorys R zugänglich sind. Die Masterarbeit betrachtet in der Regel Szenarien, in denen der Angreifer seit Erstellung des Repositorys Zugriff auf das Repository besitzt. Ein Angreifer kann jeden Zustand des Repositorys nach und während der Ausführung eines Restic-Befehls beobachten. Event-Tupel für die abstrakte Modellierung sind nach ihrer Definition aus Kapitel 3.2.5.5 aus der Sicht von S_{Backup} definiert. Daher hat auch ein Angreifer mit Zugriff auf S_{Backup} Zugriff auf alle Informationen der einzelnen Events, außer dem Chiffrat $data_{t,i}$. $\mathcal{L}$ modelliert nur die Informationen, die ein Angreifer erhält, ohne die Sicherheit des zugrundeliegenden Verschlüsselungsverfahrens von Restic zu brechen. Daher bringt $data_{t,i}$ dem Angreifer keine neuen Informationen und ist nicht Teil der Leakage-Funktion. Etwas, das jedoch Teil der Leakage-Funktion ist, ist die Länge der Chiffrate $data_{t,i}$. Konkret sind für jedes Event die Interaktionsart $op_{t,i}$, der Zeitpunkt $delay_{t,i}$ des Events und der Datenstrukturtyp $type_{t,i}$ der verwendeten Datenstruktur Teil der Leakage. Außerdem ist die Länge $size(data_{t,i})$ der verschlüsselten und authentifizierten Datenstruktur Teil der Leakage, da Restic die Länge der Chiffrate nicht verschleiert. Falls $blobInfo_{t,i}$ existiert, ist ebenfalls auch die Blob-Info Teil der Leakage, da S_{Backup} direkt mitgeteilt wird, welche Werte $start$ und end für dieses Event besitzen. Die Referenzfunktion modelliert, dass ein Angreifer Zusammenhänge zwischen Events verschiedener Event-Tupel herstellen kann. Ein Angreifer der ein Repository seit dessen Erstellung beobachtet, kann jede Datenstruktur, die im Repository gespeichert ist, zu einem früheren Event zurückverfolgen, das diese Datenstruktur dem Repository hinzugefügt hatte. Dieses Wissen des Angreifers wird explizit mit dem Wert der Referenzfunktion modelliert.
Die ID der Datenstrukturen ist nicht Teil der Leakage, da die Restic-IDs deterministisch mit den Chiffraten $data_{t,i}$ berechnet werden, und so Informationen über die Chiffrate preisgegeben könnten.

5.1.1.1. Direkt ableitbare Leakage aus der Leakage-Funktion $\mathcal{L}$

Aus den Informationen der Leakage-Funktion $\mathcal{L}(\mathcal{E}_n^R)$ kann ein Angreifer weitere Leakage ableiten. Dazu werden in diesem Kapitel weitere Erkenntnisse vorgestellt, die sich aus der Leakage-Funktion $\mathcal{L}$ ableiten lassen.

Anzahl der Datenstrukturen:
Für jeden Restic Befehl τ_t kann die Anzahl Datenstrukturen eines bestimmten Typs ermittelt werden, die durch diesen Befehl zu R hinzugefügt wurden. Aus $\mathcal{L}(\mathcal{E}_n^R)$ ist also ebenfalls ableitbar $|\{e_{t,i} \mid type_{t,i} = T \wedge op_{t,i} = send\}|$ für ein festes t und für alle $T \in \{Snapshot, Pack, Index, Key, Config\}$.

Das Gleiche kann für die Anzahl der gelöschten Datenstrukturen $op_{t,i} = delete$ und die Anzahl der angeforderten Datenstrukturen $op_{t,i} = receive$ abgeleitet werden.

Zugriffsreihenfolge bei Blobs:
Durch $\mathcal{L}_R(\tau_t)$ kann die Zugriffsreihenfolge für die Packs und damit für die Blobs ermittelt werden, da für jedes Event $e_{t,i}$, das auf einen Blob zugreift, $(delay_{t,i}, blobInfo_{t,i})$ Teil der Leakage ist.

Position der Blobs in den Packs:
Dadurch, dass $blobInfo_{t,i}$ und $ref(t,i)$ Teil der Leakage ist, kann ein Angreifer Blob-Grenzen innerhalb der Packs herausfinden. Das gilt selbstverständlich nur für Blobs oder Blob-Chunks, die auch von einem Restic-Befehl angefordert werden. Daher zählen allgemein die Position aller Blobs in jedem Pack nicht zur Leakage, sondern nur die Grenzen, von Blobs oder Blob-Chunks, die durch einen Restic-Befehl tatsächlich angefordert wurden.

Anzahl der neu erstellten Blobs:
Ein Pack-Header wird in Restic nicht komprimiert, bevor er verschlüsselt und authentifiziert wird (siehe Funktion 27). Außerdem besteht ein Pack-Header nur aus konkatenierten Blob-Headern für jeden in diesem Pack enthaltenen Blob (siehe Kapitel **??**). Die Länge eines Blob-Headers ist für jeden Blob gleich groß und wird mit $size_{BlobHeader}$ bezeichnet. Mit diesen Informationen lässt sich aus dem in der Leakage eines Events für eine Pack-Datenstruktur vorkommenden $packHeaderSize_{t,i}$ die Anzahl der Blob-Header in diesem Pack-Header bestimmen. Mit $\frac{(packHeaderSize_{t,i} - (Overhead_{IV_{t,i}} + Overhead_{MAC_{t,i}}))}{size_{BlobHeader}}$ kann die Anzahl Blob-Header in dem Pack-Header dieses Packs berechnet werden. Da jeder Blob, der in dem Pack gespeichert ist, einen eigenen Blob-Header im Pack-Header besitzt, entspricht die Anzahl Blob-Header auch der Anzahl Blobs, die in diesem Pack gespeichert sind. Damit kann die Gesamtanzahl aller neu erstellten Blobs durch alle Events mit $op_{t,i} = send$ und $type_{t,i} = Pack$ bestimmt werden. Es kann zunächst jedoch nicht unterschieden werden, welche Packs Data-Blobs enthalten und welche Packs Tree-Blobs enthalten.
$Overhead_{IV_{t,i}}$ und $Overhead_{MAC_{t,i}}$ bezeichnen den Speicherbedarf des IVs und MACs des Pack-Headers aus $data_{t,i}$ in Bytes. Denn sowohl ein IV, als auch ein MAC sind Teil des Pack-Headers, da dieser verschlüsselt und authentifiziert wird.

Größe der Konkatenierten Blobs eines Packs:
Ein Pack besteht immer aus konkatenierten Blobs, einem Pack-Header und der Größe des Pack-Header in Form der letzten 4 Bytes einer Pack-Datenstruktur. Damit kann aus der Gesamtgröße der Pack-Datei $size(data_{t,i})$ und $packHeaderSize_{t,i}$ die Größe der konkatenierten Blobs dieses Packs berechnet werden durch die Formel $(size(PackBlobs) := size(data_{t,i}) - (packHeaderSize_{t,i} + 4))$.

Parent-Snapshot:
backup Befehle und *restore* Befehle, können einen Parent-Snapshot verwenden. Dieser kann entweder durch die Option *PARENT* oder *SNAPSHOT_ID* direkt durch den Befehl

festgelegt werden, oder es wird der Latest-Snapshot bestimmt und als Parent-Snapshot verwendet (siehe Funktion 23). Wird der Parent-Snapshot durch die Optionen festgelegt, wird nur ein Snapshot aus dem Repository geladen. Das ist in der Leakage erkennbar durch nur ein Event mit $op_{t,i} = receive$ und $type_{t,i} = Snapshot$ für den t-ten Befehl. In einem solchen Fall ist der verwendete Parent-Snapshot ebenfalls aus der Leakage ableitbar.
Im zweiten Fall werden alle Snapshots des Repositorys angefordert und daraus ein Latest-Snapshot bestimmt. Dadurch kann aus der Leakage zunächst nicht hergeleitet werden, welcher Parent-Snapshot verwendet wurde. Im weiteren Verlauf des *backup* Befehls oder *restore* Befehls werden jedoch weitere Tree-Blobs des Verzeichnisbaums des Parent-Snapshots angefordert. Der erste angeforderte Tree-Blob eines dieser Befehle ist der Root-Tree-Blob des Parent-Snapshots.

Theorem 2 (Root-Tree-Blob Eindeutigkeit) *Jeder Snapshot kann einem Root-Tree-Blob zugeordnet werden.*

Ein Snapshot wird nur durch einen *backup* Befehl angelegt. Fall 1 (neue Daten):
Wurden mit dem *backup* Befehl mindestens ein Tree-Blob oder Data-Blob dem Repository hinzugefügt, heißt das, dass ein Blob mit dieser ID nicht im Repository vorhanden ist. Jeder Knoten des Verzeichnisbaums ist durch einen Tree-Blob repräsentiert, der in Form von Node-Einträgen die IDs aller Tree-Blobs speichert die seinen Konten einen Kindknoten im Verzeichnisbaum repräsentieren. Ebenso werden auch die IDs der Data-Blobs in Node-Einträgen der Tree-Blobs gespeichert. Ein Tree-Blob TB_{new}, der bisher nicht im Repository gespeichert wurde besitzt damit eine ID, die bisher von keinem Tree-Blob verwendet wurde. Damit wurde die ID auch in keinem Node-Eintrag eines Tree-Blob des Repositorys verwendet. Somit ist der Tree-Blob des Elternknotens von TB_{new} bisher ebenfalls nicht im Repository gespeichert. Unter der Annahme, dass es zu keiner Kollision der IDs der Tree-Blobs kommt, muss also auch der Tree-Blob des Elternknotens zu dem Repository hinzugefügt werden. Das gilt analog für die Tree-Blobs alle Elternknoten, bis zum Root-Tree-Blob, der ebenfalls dem Repository hinzugefügt wird. Damit besitzt dieser Snapshot einen eindeutigen Root-Tree-Blob, der ein Send-Event im zugehörigen Event-Tupel des *backup* Befehls besitzt.

Fall 2 (keine neuen Daten):
Speichert der *backup* Befehl B keinen einzigen neuen Tree-Blob oder Data-Blob im Repository, wurde bereits ein Snapshot auf dem exakt gleichen virtuellen Verzeichnissystem ausgeführt. Damit existiert der Root-Tree-Blob des Snapshots, der für diesen *backup* Befehl B erstellt wird, bereits im Repository. □

Umgekehrt gilt dieses Theorem jedoch nicht. Wie in dem Beweis zu sehen ist, kann ein Root-Tree-Blob von mehreren Snapshots verwendet werden. Für die Leakage bedeutet das, dass das Event $e_{t,i}$ eines angeforderten Root-Tree-Blob mit $ref(t,i)$ nicht eindeutig auf das Event-Tupel verweist, das den ausgewählten Parent-Snapshot enthält. Allerdings weist $ref(t,i)$ auf das Event des Event-Tupels, mit dem alle Daten des Parent-Snapshots zu dem Repository hinzugefügt wurden.

5.1.2. Unterscheidbarkeits-Leakage

In den vorgestellten Sicherheitsspielen darf ein Angreifer selbst einen Restic-Befehl und dessen Optionswerte wählen. Damit lässt sich eine weitere Leakage-Darstellung herleiten. Die Leakage, die aus der Differenz zweier ausgeführter Restic-Befehle für einen Angreifer sichtbar ist. Diese Leakage wird $\mathcal{L}_{dif}$ genannt. $\mathcal{L}_{dif}$ stellt jedoch keine weitere Leakage-Quelle dar, sondern ist eine Charakterisierung der Leakage-Funktion $\mathcal{L}$ durch die Betrachtung unterscheidbarer Restic-Befehle.

Definition 5.1.5 (Unterscheidbarkeits-Leakage $\mathcal{L}_{dif}$) *Die Unterscheidbarkeits-Leakage $\mathcal{L}_{dif}$ ist definiert, als die Menge aller Eigenschaften P, durch die zwei Restic-Befehle des gleichen Typs eindeutig unterschieden werden können.*

$$\mathcal{L}_{dif}(\tau_0, \tau_1) := \{P \mid P(\mathcal{L}(\tau_0)) \neq P(\mathcal{L}(\tau_1))\}$$

5.1.2.1. Optionswerte für die Restic-Befehle

Unterschiedliche Optionswerte für die Restic-Befehle führen ebenfalls durch die Leakage unterscheidbar, die $\mathcal{A}$ einsehen kann. Wie sich der Optionswert *PARENT* für den Parent-Snapshot auf die Leakage auswirkt, wurde bereits aus der Leakage-Funktion hergeleitet. Besitzen zwei Event-Tupel für einen *backup* Befehl oder *restore* Befehl jeweils nur ein Event mit $op_{t,i} = receive$ und $type_{t,i} = Snapshot$, wurde die *PARENT* Option oder *SNAPSHOT_ID* Option verwendet. Unterscheidet sich die Referenzfunktion $ref(t, i)$ für diese beiden Events der beiden Event-Tupel, dann kann ein Angreifer, der die Optionswerte kennt, die Event-Tupel unterscheiden und die Event-Tupel den beiden Restic-Befehlen korrekt zuordnen. Nun werden die restlichen Optionswerte für die abstrakte Modellierung der Restic-Befehle betrachtet.

***pack_size* Optionswerte:**
Die *PACK_SIZE* Option gibt an, wie groß die erstellten Pack-Datenstrukturen $\tilde{P}_i$ eines Restic-Befehls minimal sein müssen, damit kein Blob mehr hinzugefügt wird und sie an S_{Backup} zum Speichern geschickt werden. Durch die Leakage-Funktion ist die Größe jedes Packs $size(data_{t,i})$, das dem Repository hinzugefügt wird $op_{t,i} = send$, Teil der Leakage. Zwei Restic-Befehle, die mit einem unterschiedlichen Wert für die *PACK_SIZE* Option ausgeführt werden, lassen sich durch diese Leakage unterscheiden. $\mathcal{A}$ kann anhand der Größe der Pack-Dateien unterscheiden, zu welchem Restic-Befehl welches Event-Tupel gehört.

***connections* Optionswerte:**
Restic erstellt viele parallele Prozesse basierend auf dem Wert der *CONNECTIONS* Option für einen Restic-Befehl. Darunter fällt beispielsweise die Anzahl der Pack-Uploader Prozesse. Kombiniert ein Restic-Befehl einen großen Wert für *CONNECTIONS* mit einem kleinen Wert für *PACK_SIZE* werden in sehr kurzen Abständen viele Pack-Datenstrukturen $\tilde{P}_i$

erzeugt, die an S_{Backup} geschickt werden. Für einen Restic-Befehl mit einem kleineren Wert für $CONNECTIONS$, wie beispielsweise 1, kann immer nur ein $\tilde{P}_i$ gleichzeitig an S_{Backup} geschickt werden. Dieses Verhalten kann dadurch beobachtet werden, dass $dalay_{t,i}$ Teil der Leakage ist.

***overwrite* Optionswerte:**
Die $OVERWRITE$ Option wird für den *restore* Befehl verwendet, um zu entscheiden, ob Dateien und Verzeichnisse, die sowohl auf dem virtuellen Verzeichnissystem vorkommen, als auch wiederherstellt werden sollen, ersetzt, gelöscht oder beibehalten werden sollen (siehe Tabelle 3.4). Die $OVERWRITE$ Option entscheidet damit indirekt, wie viele Data-Blobs aus dem Repository angefordert werden (je mehr überschrieben wird, desto mehr Blobs werden angefordert). Teil der Leakage eines Event-Tupels für einen *restore* Befehl auch immer die Größe $size(data_{t,i})$ der angeforderten $op_{t,i} = send$ Data-Blob-Chunks. Damit kann für zwei Event-Tupel die Gesamtgröße aller angeforderten Daten-Blobs DBC_i berechnet werden. Ein Angreifer, der ein Befehlstupel für den *restore* Befehl wählt, wählt insbesondere auch das virtuelle Verzeichnissystem, auf dem dieser Befehl ausgeführt wird. Wählt der Angreifer also zwei gleiche *restore* Befehlstupel mit unterschiedlichen Werten für die $OVERWRITE$ Option, kann der Angreifer die resultierenden Event-Tupel den beiden Befehlen eindeutig zuordnen.

***unsafe_recover* Optionswerte:**
Wird die $UNSAFE\_RECOVER$ Option für einen *prune* Befehl verwendet, werden zuerst alle Indices aus dem Repository entfernt und am Ende des Befehls werden neue Indices zum Repository hinzugefügt. Dadurch, dass die zeitliche Abfolge $delay_{t,i}$ der Befehle Teil der Leakage eines Event-Tupels ist, kann direkt überprüft werden, ob für die letzten Events eines Event-Tupels gilt $op_{t,i} = send$ und $type_{t,i} = Index$. Ist das der Fall, wurde die $UNSAFE\_RECOVER$ Option verwendet.

***max_unused* und *max_repack_size* Optionswerte:**
Diese beiden Optionen werden für den *prune* Befehl verwendet und können bestimmen indirekt, wie viel Packs aus dem Repository nicht gelöscht werden, sondern unverändert im Repository erhalten bleiben. Ein Angreifer, der den Repository-Zustand vor Ausführung eines *prune* Befehls kennt, kann durch die Referenzfunktion $ref(t, i)$ der einzelnen Events für die Packs des Repositorys rekonstruieren, welche und wie viele Packs unverändert im Repository erhalten geblieben sind. Da der Angreifer in den betrachten Angriffsszenarien, das Repository seit dessen Erstellung beobachtet, kennt der Angreifer den Repository-Zustand vor der Ausführung eines *prune* Befehls und kann so zwei Befehle mit unterschiedlichen Werten für diese Optionen unterscheiden.

5.1.2.2. Anzahl wiederverwendeter Blobs

Werden in einem *backup* Befehl Blobs wiederverwendet, werden diese Blobs nicht erneut verschlüsselt, authentifiziert und zu einem Pack hinzugefügt, das an S_{Backup} geschickt wird.

Die Gesamtanzahl der Blobs, die durch einen *backup* Befehl erzeugt werden, ist Teil der Leakage des Event-Tupels dieses *backup* Befehls. Damit können zwei *backup* Befehle, die unterschiedlich viele Blobs wiederverwenden unterschieden werden, sofern durch die Wahl des Befehlstupels eine ungefähre Anzahl Blobs bekannt ist, die für diese Befehle erstellt werden.

5.1.2.3. Komprimierung

Restics Komprimierungsverfahren findet auf den unverschlüsselten Daten statt und kann vom Angreifer simuliert werden, sofern der Angreifer den Inhalt der Datenstrukturen kennt, die komprimiert und verschlüsselt werden. Beim Verschlüsseln einer Datenstruktur ist die Größe der verschlüsselten Datenstruktur identisch mit der Größe der unverschlüsselten Datenstruktur. Kennt $\mathcal{A}$ also eine unverschlüsselte Datenstruktur oder Data-Blob, kann $\mathcal{A}$ Restics Komprimierung simulieren und so eine exakte Größe der komprimierten Datenstruktur oder des komprimierten Data-Blobs bestimmen. Kennt $\mathcal{A}$ eine unverschlüsselte Datenstruktur oder Data-Blob nur teilweise, kann $\mathcal{A}$ die Größe der komprimierten Datenstruktur oder des komprimierten Data-Blobs nur ungefähr bestimmen oder eine Mindestgröße der komprimierten Daten bestimmen.
Die Größe der komprimierten Datenstrukturen ist Teil der Leakage-Funktion mit $size(data_{t,i}) - (Overhead_{IV_{t,i}} + Overhead_{MAC_{t,i}})$. Somit können zwei Restic-Befehle unterschieden werden, falls Restic eine unterschiedlich starke Komprimierung für die beiden Restic-Befehle verwendet.

5.1.3. Provozierte Leakage

In diesem Kapitel wird auf weitere Leakage eingegangen, die ebenfalls aus der Leakage-Funktion für mehere Event-Tupel hergeleitet werden kann. Diese Leakage wird provozierte Leakage genannt, da sie sich nur für einen speziellen Angreifer ergibt, der diese Leakage mit der Wahl seiner Restic-Befehle provoziert. Diese provozierte Leakage ist jedoch aus der Leakage-Funktion vollständig ableitbar, sofern die Restic-Befehle und deren Optionswerte bekannt sind. Für die Unterscheidbarkeits-Leakage waren die Restic-Befehle und deren Optionswerte ebenfalls bekannt, allerdings lag bei $\mathcal{L}_{dif}$ der Fokus auf der Unterscheidbarkeit zweier Restic-Befehle des gleichen Typs. Die provozierte Leakage braucht keine zweite Ausführung eines anderen Restic-Befehls, um diese Leakage zu erhalten. Bei der provozierten Leakage liegt der Fokus daraus, dass ein Angreifer mit geschickt gewählten Restic-Befehlen Leakage erzeugen kann, die zwar aus der Leakage-Funktion herleitbar ist, aber nicht explizit Teil dieser Leakage-Funktion ist.

5.1.3.1. Provozierte Leakage durch den *backup* Befehl

Ein Angreifer $\mathcal{A}$ kann durch das Beobachten eines Repositorys während einer *backup* Befehlsausführung keine Rückschlüsse auf die einzelnen Tree-Blobs ziehen. $\mathcal{A}$ kennt nur

die Größe der konkatenierten verschlüsselten Tree-Blobs in den Packs $\tilde{P}_i$, die aus diesem *backup* Befehl entstehen. Ebenso kann $\mathcal{A}$ zunächst nicht zuordnen, welche Packs zu Data-Blobs gehören und welche Packs zu Tree-Blobs gehören.
Durch weitere *backup* Befehle mit dem ursprünglichen Backup als Parent-Snapshot, ist es $\mathcal{A}$ jedoch möglich die Grenzen der einzelnen Tree-Blobs des ursprünglichen Backups aus ihren Packs zu extrahieren. Dieses Vorgehen wird gezeigt und es wird die daraus resultierende Leakage erläutert.

Tree-Blobs extrahieren:
Es wird ein *backup* Befehl $B := (VS, O, TP)$ durchgeführt mit $T := (V, E)$ als Verzeichnisbaum für VS und $V := \{v_1, \ldots, v_m, v_{m+1}, \ldots, v_{m+n}\}$, wobei m die Anzahl der Verzeichnisse von VS ist, wie in Kapitel 2.3.1.9 beschrieben. Der aus B resultierende Snapshot wird $SS_{original}$ genannt. Nun wird ein weiterer *backup* Befehl mit (VS', O', TP) ausgeführt, wobei O' als Parent-Snapshot auf $SS_{original}$ verweist und VS' den Verzeichnisbaum $T' := (V', E')$ mit $V' := v_1$ besitzt. Restic fordert vom Repository den Tree-Blob $TB(v_1)$ einzeln an, da $SS_{original}$ als Parent-Snapshot gewählt wurde. Damit sieht $\mathcal{A}$ in welchem Pack TB_1 liegt, wie groß TB_1 ist und an welcher Position er sich in dem Pack befindet. Dieses Vorgehen kann beliebig wiederholt werden, wobei jeweils ein weites Verzeichnis v_i mit $1 \leq i \leq m$ mit zu VS' hinzugenommen wird. Dadurch erhält $\mathcal{A}$ nach und nach diese Informationen für alle TB_i des ursprünglichen *backup* Befehls B. Außerdem findet $\mathcal{A}$ heraus welche durch B erstellten Packs $\tilde{P}_i$ Tree-Blobs enthalten und welche Data-Blobs enthalten, da alle erstellten Packs $\tilde{P}_i$, die keine Tree-Blobs enthalten, Data-Blobs enthalten müssen.

Leakage von Tree-Blobs:
Damit wurde gezeigt, wie durch Kenntnis, der *backup* Befehlstupel für Tree-Blob deren Größe, Packzuordnung und Position in den Packs aus der Leakage-Funktion mehrerer Event-Tupel abgeleitet werden kann. Mit dieser Leakage und ausreichend *backup* Befehlen kann aus der Leakage von $\mathcal{E}_n^R$ ebenfalls die Gesamtanzahl, der durch einen *backup* Befehl erstellten Tree-Blobs hergeleitet werden. Außerdem zählt die Zuordnung der Tree-Blobs zu den Verzeichnissen ebenfalls zur Leakage, sofern $\mathcal{A}$ das Verzeichnissystem VS für den ausgeführten *backup* Befehl kennt.
Ein Nebeneffekt der Leakage über die Anzahl der erstellten Tree-Blob ist, dass dadurch ebenfalls die Anzahl der erstellten Data-Blobs zur Leakage gehört. Durch die Leakage des Event-Tupels τ_t sind alle Send-Events zum Speichern von neuen Packs im Repository bekannt. Kann die Leakage aus mehreren Event-Tupeln abgeleitet werden, welches Pack des ursprünglichen Event-Tupels τ_t Tree-Blobs beinhalten, müssen die anderen Packs zwangsläufig Data-Blobs enthalten. Da die Anzahl der enthaltenen Blobs aus der Leakage eines Send-Events für ein Pack direkt ableitbar ist, kann so auch die Gesamtanzahl neu erstellter Data-Blobs durch τ_t abgeleitet werden.

5.1.4. Provozierte Leakage durch den *restore* Befehl

Für den *restore* Befehl existiert ähnlich zu dem *backup* Befehl eine Möglichkeit, die Data-Blobs statt der Tree-Blobs aus den Packs des Repositorys zu extrahieren. Durch mehrere

restore Befehle mit dem gleichen Backup als Parent-Snapshot, ist es $\mathcal{A}$ möglich die Grenzen der einzelnen Data-Blobs des ursprünglichen Backups aus ihren Packs zu extrahieren. Dieses Vorgehen wird gezeigt und es wird die daraus resultierende Leakage erläutert.

Data-Blobs extrahieren:
Wie bei den Tree-Blobs durch den *backup* Befehl können mit geschickt gewählten *restore* Befehlen nun die Data-Blobs früherer Backups aus den Packs extrahiert werden.
Zunächst wird ein *backup* Befehl $B := (VS, O, TP)$ durchgeführt mit $T := (V, E)$ als Verzeichnisbaum für VS und $V := \{v_1, \ldots, v_m, v_{m+1}, \ldots, v_{m+n}\}$, wobei m die Anzahl der Verzeichnisse von VS ist, wie in Kapitel 2.3.1.9 beschrieben. Der aus B resultierende Snapshot wird $SS_{original}$ genannt. Nun wird ein *restore* Befehl ausgeführt mit $R := (VS', O', TP')$, wobei VS' das gleiche Verzeichnissystem ist wie VS mit der Änderung, dass für die Datei v_{m+1} das erste Byte inkrementiert wurde. R wird mit $SS_{original}$ zur Wiederherstellung ausgeführt. Da sich VS' zu dem ursprünglichen VS nur in einer Datei unterscheidet, fordert Restic nur den Data-Blob von S_{Backup} an, der dieses geänderte Byte enthält. Damit sieht $\mathcal{A}$ in welchem Pack $DB_{m+1,1}$ liegt, wie groß $DB_{m+1,1}$ ist und an welcher Position dieser Data-Blob sich in dem Pack befindet. Dieses Vorgehen kann beliebig wiederholt werden, wobei jeweils ein weiteres Byte aus einem anderen Chunk dieser Datei v_{m+1} inkrementiert wird. Dadurch erhält $\mathcal{A}$ nach und nach diese Informationen für alle Data-Blobs $DB_{m+1,i}$ der Datei zu v_{m+1}. Wird dieses Vorgehen für alle v_i mit $m+1 \leq i \leq m+n$ wiederholt, erhält $\mathcal{A}$ die angegebenen Informationen für alle Data-Blobs aller Dateien, von denen durch B ein Backup erstellt wurde.

Allgemeine Leakage von Data-Blobs:
Für die Data-Blobs ist ebenfalls die Größe der Data-Blobs, Packzuordnung und Position in den Packs aller am Backup B beteiligten Data-Blobs $\{DB(v_{m+1}), \ldots, DB(v_{m+n})\}$ aus der Leakage der Event-Tupel ableitbar. Außerdem ist die Zuordnung der Data-Blobs zu den Packs und die Reihenfolge der Data-Blobs ebenfalls aus der Leakage-Funktion ableitbar, sofern $\mathcal{A}$ das Verzeichnissystem VS für den ausgeführten *backup* Befehl B kennt.
Analog gilt auch hier wieder, dass die Leakage der Data-Blobs automatisch die Leakage über die Anzahl der Tree-Blobs und Tree-Blob-Packs aus dem Event-Tupel für die Ausführung des *backup* Befehls B impliziert.

Leakage über wiederverwendete Data-Blobs:
Etwas, was beim Extrahieren der Tree-Blobs nicht durch $\mathcal{A}$ erkennbar war, war, ob und wie viele Data-Blobs aus vorherigen Backups bereits im Repository vorhanden sind und wiederverwendet wurden. Durch den *restore* Befehl kennt $\mathcal{A}$ die Werte der Referenzfunktion für die angeforderten Data-Blobs. Durch die Beobachtung der Leakage der vorherigen *backup* Befehle weiß $\mathcal{A}$ welche Packs mit welchen früheren *backup* Befehlstupeln zum Repository hinzugefügt wurden. Damit kann aus der Leakage-Funktion ebenfalls abgeleitet werden, welche Data-Blobs von welchen *backup* Befehlen wiederverwendet wurden, wenn entsprechende *restore* Befehle für diese Backups als Parent-Snapshot ausgeführt werden.

Leakage über die Chunkgrenzen:
Ein Angreifer $\mathcal{A}$, der die Größe der komprimierten Data-Blobs einer Benutzerdatei kennt, der die Reihenfolge der Data-Blobs aus denen sich die Datei zusammen setzt kennt und der die unverschlüsselte Benutzerdatei selbst kennt, ist in der Lage die ursprünglichen Chunks, in die die Datei zerlegt wurde zu rekonstruieren. Dazu wird folgendes Verfahren vorgestellt, mit der ein Angreifer aus diesen Erkenntnissen die Chunks einer Benutzerdatei rekonstruieren kann.
Dazu betrachtet der Angreifer immer einen Ausschnitt BS der Klartext-Datei und erweitert diesen Ausschnitt Byte für Byte, bis eine korrekte komprimierte Größe für diesen Ausschnitt gefunden wird. Der Angreifer $\mathcal{A}$ beginnt für den ersten Chunk der Datei für BS mit den ersten k Bytes der Datei, wobei k die Hälfte der Größe des ersten komprimierten Data-Blobs dieser Datei ist. $\mathcal{A}$ führt den Komprimierungsalgorithmus von Restic über diesem Bytestring BS der Länge k durch und vergleicht die komprimierte Größe mit der Größe des ersten Data-Blobs $size(data_{t,i})$. Ist BS kleiner als der erste Data-Blob, wird das nächste Byte der Datei an das Ende von BS hinzugefügt und der Komprimierungsalgorithmus erneut für BS ausgeführt. Das wird so lange wiederholt bis die komprimierte Größe von BS der Größe des ersten Data-Blobs entspricht. Dann wurde wahrscheinlich eine Chunkgrenze gefunden und BS ist der erste Chunk. Dieses Vorgehen kann nun für die nachfolgenden Bytes der Datei und die nächsten Data-Blobs wiederholt werden. Durch die Komprimierung zweier Bytestrings BS_1 und $BS_2 := BS_1||BS_3$, wobei BS_3 ein weiterer Bytestring ist, kann es vorkommen, dass die komprimierten Größen von BS_1 und BS_2 gleich sind. Dadurch gibt es mehrere mögliche Chunkgrenzen, wenn nur die komprimierte Chunkgröße bekannt ist. Wurde eine falsche Chunkgrenze für einen der Chunks gewählt, kann für mindestens einen nachfolgenden Data-Blob kein Chunk gefunden werden, der die gleiche komprimierte Größe besitzt wie der Data-Blob. Dann muss ab dem ersten Chunk überprüft werden, ob durch Hinzunahme weiterer Bytes zu den Bytestrings der Chunks erneut eine mögliche Chunkgrenze gefunden wird. Ist dies der Fall, wird mit dieser neuen Chunkgrenze das Verfahren fortgesetzt.
Damit lassen sich ebenfalls die Chunkgrenzen der Benutzerdateien aus der Leakage-Funktion ableiten, sofern $\mathcal{A}$ die unverschlüsselten Dateien kennt. Die Chunkgrenzen sind normalerweise nicht Teil der Leakage, da die Config-Datenstruktur des Repositorys, die das geheime Polynom speichert, mit dem das Chunkingverfahren Chunkgrenzen berechnet, verschlüsselt ist.

5.1.5. Provozierte Leakage durch den *backup* Befehl

Für den *prune* Befehl gibt es keine provozierte Leakage. Das Befehlstupel eines *prune* Befehls besteht nur aus den Optionswerten und einer Liste von Snapshot-IDs, die zu Beginn der Ausführung des *prune* Befehls aus dem Repository gelöscht werden. Es werden keine Packs sichtbar modifiziert, wodurch eventuelle Leakage auftreten könnte. Alle Packs, die verändert werden, werden von S_{Restic} angefordert und für einen Angreifer nicht einsehbar verändert. Dabei werden alle umgepackten Blobs neu verschlüsselt und es werden Blobs verschiedener Backups nichtdeterministisch in gemeinsame neue Packs gepackt, wodurch die Ausführung eines *prune* Befehls eher hinderlich ist für die zukünftige Leakage weitere

Event-Tupel.
Die Leakage, die aus den Optionswerten für einen *prune* Befehl resultiert, wurde bereits besprochen. Das Aufteilen der Snapshot-IDs in einzelne *prune* Befehle fügt zur Leakage nur hinzu, wie viel Blobs durch das entfernen des jeweiligen Snapshots aus dem Repository entfernt werden. Da die Anzahl Blobs im Repository bereits Teil der Leakage ist und so der Repository-Zustand vor und nach der Ausführung des *prune* Befehls verglichen werden kann.
Es sei SS der Snapshot, der mit einem *backup* Befehl zu dem Repository hinzugefügt wurde. Werden weniger Blobs durch einen *prune* Befehl mit $SSID = \{id(SS)\}$ entfernt, als durch den *backup* Befehl der SS zum Repository hinzugefügt hatte, kann für die Leakage geschlussfolgert werden, dass die nicht entfernten Blobs von Snapshots anderer *backup* Befehle verwendet werden.

5.1.6. Konsequenzen der Leakage

Dieses Kapitel betrachtet die realen Konsequenzen der Leakage. Was ein Angreifer nur durch Beobachten eines Repositorys oder durch das Beobachten des Nachrichtenaustauschs zwischen S_{Restic} und S_{Backup} lernen.

5.1.6.1. Keys

Der größte Teil einer Key-Datenstruktur ist unverschlüsselt im Repository gespeichert. Dadurch sind Einträge, wie *username* und *hostname* direkt einsehbar, was als Verletzung der Sicherheitseigenschaft Privacy angesehen werden kann. Ein Angreifer $\mathcal{A}$ mit Zugriff auf ein Repository, sieht außerdem welche Key-Datenstruktur K bei der Ausführung eines Restic-Befehls angefordert wird. Dadurch erfährt ein Angreifer, welcher Benutzer gerade das Repository benutzt und welche Restic-Befehle er ausführt.

5.1.6.2. Zugriffsmuster

Jeder Restic-Befehl hat einen bestimmten strukturierten Ablauf, anhand dessen man den Befehl identifizieren kann. Dadurch können Verhaltensmuster erkannt werden, die Aufschluss darüber geben, wie häufig jemand ein System wiederherstellt.

5.1.6.3. Metadaten

Alle Dateien im Repository besitzen Metadaten, wie beispielsweise Zeitstempel. Mit den Zeitstempeln der Snapshot-Dateien wird der Backup-Rythmus einsehbar.

5.1.6.4. Deduplikation

Deduplikation ist eine elegante Art redundanten Speicherungen und damit Große Repositorys zu vermeiden. Deduplikation kann jedoch auch sehr viel Informationen zur Leakage beitragen. Schafft es ein Angreifer beispielsweise eine simple Text-Datei in ein Verzeichnissystem des S_{Restic} zu schleusen, von dem ein Backup durchgeführt wird, können mit der Deduplikation Daten aus dem Repository geleakt werden. Angenommen, der *backup* Befehl wird regelmäßig auf dem Verzeichnissystem, in dem sich die Datei des Angreifers befindet, aufgerufen. Dann werden mit jedem Backup wahrscheinlich nur kleine Mengen an Daten oder sogar gar keine neuen Daten zum Repository hinzugefügt.
Ein Angreifer, der Zugriff auf das Repository hat und den Inhalt dieser Datei bestimmen, kann nun erkennen, ob der Inhalt der Datei bereits im Repository liegt oder nicht, je nachdem wie viele neue Packs durch einen *backup* Befehl erstellt wurden. Ein Angreifer kann eine solche Datei beispielsweise als Anhang einer E-Mail, der automatisch heruntergeladen wird, auf S_{Restic} schleusen.

5.1.6.5. Verzeichnisbäume

In Restic wird bereits viel parallelisiert, allerdings erfolgt das Traversieren eines Verzeichnisbaums sequenziell in Form einer Tiefensuche. Dadurch kann die Zugriffsreihenfolge von Tree-Blobs des Parent-Snapshots bei einem *backup* Befehl betrachtet werden. Somit ist es möglich ein Verzeichnishierarchie zu rekonstruieren, ohne die Klartexte zu kennen.

5.1.6.6. Chunking-Fingerabdruck

Restic verwendet ein Content-Defined-Chunking. Das bedeutet der Klartext ist eine direkte Eingabe für die Chunking-Funktion, die die Chunkgrenzen berechnet. Damit sind die Chunkgrenzen abhängig vom Klartext. Angenommen, ein Angreifer kann beispielsweise durch Zugriffsmuster herauszufinden, welche Data-Blobs (Chunks) zu einer Datei gehören. Eventuell gibt es bekannte Dateien, die sich in die exakt gleichen Chunks zerlegen lassen und man ist so in der Lage bestimmte Klartexte zu rekonstruieren. In der Realität geht Restic mit seinem Komprimierungsalgorithmus dagegen sehr gut vor, da dadurch die Klartext-Länge verschleiert wird.

5.1.6.7. Key-Recovery für Restics Chunking

Das Paper [10] stellt einen Key-Recovery Angriff auf Restics Chunkingverfahren vor. Dabei ist der Schlüssel, der durch den Angriff berechnet wird, das Polynom, mit dem Restics Chunkingverfahren arbeitet. Durch solche Angriffe können die Blob-Grenzen innerhalb der Pack-Datenstrukturen berechnet werden. Für das Angreifermodell dieser Masterarbeit, ist das jedoch nicht weiter schlimm, da auch dieser Angreifer durch geschickt gewählte Restic-Befehle diese Leakage aus der Leakage-Funktion ableiten kann.

5.2. Prolog für die Sicherheitsspiele

Dieses Kapitel betrachtet allgemeine Lemma, Parameter und Vorgehensweisen, die für die vorgestellten Sicherheitsspiele verwendet werden. Es wird erklärt, warum für alle Sicherheitsspiele immer zunächst eine abstrakte KDF angenommen wird, satt einer passwortbasierten KDF.
Bisher wurden alle Restic-Befehle relativ nah an der Realität und den in der Realität verwendeten konkreten Primitiven modelliert. Hier werden allgemeine Varianten der Verschlüsselungsfunktion und Entschlüsselungsfunktion mit allgemeinen kryptografischen Primitives betrachtet.

5.2.1. Sicherheitsparameter λ

In den vorgestellten Sicherheitsspielen wird für den Reduktionsbeweis zunächst die asymptotische Sicherheitsbetrachtung herangezogen, um eine allgemeine Schranke für den Vorteil des Angreifers aufzustellen. Danach werden konkrete Parameter für die konkrete Sicherheit von Restics Primitiven betrachtet.
Für die asymptotische Sicherheit, wird jedoch ein Sicherheitsparameter $\lambda \in \mathbb{N}$ eingeführt, der die Stärke der abstrakten kryptografischen Primitive parametrisiert. Alle verwendeten Algorithmen erhalten 1^λ implizit als Eingabe in unärer Darstellung. Ein Algorithmus heißt effizient, wenn seine Laufzeit polynomiell in λ ist. Ein Angreifer wird erfolgreich genannt, wenn sein Vorteil nicht vernachlässigbar in λ ist. Damit ist gemeint, dass es keine Funktion $negl : \mathbb{N} \to \mathbb{R}$ gibt mit $Adv(\mathcal{A}, \lambda) \leq negl(\lambda)$ für alle hinreichend großen λ.
Wie zu Beginn erwähnt, dient λ in der allgemeinen Reduktion als asymptotischer Sicherheitsparameter, der die Sicherheitsaussage unabhängig von konkreten Primitiven macht. Beim Einsetzen von Restics verwendeten Primitives wird λ durch konkrete Sicherheitsabschätzungen ersetzt.

5.2.2. Einführen einer abstrakten KDF

Restic verwendet eine passwortbasierte KDF (pbKDF), die aus einem Benutzerpasswort zusammen mit einem Salt einen Schlüssel herleitet. Die formale Sicherheitsaussage, die in den folgenden Kapiteln getroffen wird, gilt jedoch nur unter der Annahme zufällig gleichverteilt gezogener Schlüssel für die kryptografischen Primitive. In der Realität wählen Benutzer Passwörter mit deutlich niedrigere Entropie, wodurch beispielsweise Wörterbuchangriffe den Vorteil des Angreifers stärker erhöhen, als die formale Schranke aussagt. Solche Angriffe liegen außerhalb des betrachteten Angriffermodells.
Um diese Annahme sauber zu erfassen, wird für beide Sicherheitsspiele ein abstrakte KDF eingeführt. Diese KDF ist PRF-sicher und wählt zufällig gleichverteilt einen Schlüssel aus einem gegebenen Salt, der das Passwort abstrahiert. Dieser gewählte Schlüssel stellt den Userkey der abstrakten Modellierung der Restic-Befehle dar. Anschließend wird in der

Reduktion gezeigt, dass Restics KDF-Primitiv scrypt, die Anforderungen an die abstrakte KDF erfüllt, solange die Annahme eines Passworts mit entsprechend hoher Entropie gilt.

Definition 5.2.1 (Abstakte KDF) *Sei $KDF_k(s)$ eine PRF-sichere KDF, mit Schlüssel $k \in \mathcal{K}(\lambda)$ und Eingabe $s \in \mathcal{S}(\lambda)$, die einen uniformen Schlüssel k ausgibt.*

$$k \xleftarrow{\$} KDF_{k_{KDF}}(1^\lambda, s)$$

Restic benötigt zwei Schlüssel. Einen Schlüssel für das Verschlüsselungsverfahren und einen für das Authentifizierungsverfahren. Dazu wird ein mit $KDF_{k_{KDF}}$ gezogener Schlüssel in zwei Schlüssel aufgeteilt $(k_{enc}, k_{auth}) := k$, für die gilt $k_{enc}||k_{auth} = k$. Aus der Umkehrung von Lemma 1 folgt, dass auch k_{enc} und k_{auth} uniforme Schlüssel sind.

5.2.3. Abstraktion der Sicherheitsprimitive

Die vorgestellten Sicherheitsspiele verwenden die abstrakte Modellierung der drei betrachteten Restic-Befehle. Die abstrakten Modellierungen sind jedoch indirekt weiterhin mit den konkreten Sicherheitsprimitiven von Restic definiert. Damit die Sicherheitsspiel nicht von den konkreten Sicherheitsprimitiven von Restic bestimmt sind, wird in diesem Kapitel die abstrahierung der Sicherheitsprimitive von Restic vorgestellt.

Schlüsselableitungsfunktion:
Die abstrakte Modellierung der Restic-Befehle wurde auf einem bereits initialisierten Repository betrachtet. In der Realität gibt es einen weiteren Restic-Befehl *init*, der für die Initialisierung eines Repositorys zuständig ist. Dieser Befehl wählt einen zufälligen gleichverteilten Masterkey $mk \xleftarrow{\$} \mathcal{K}(\lambda)$, ein zufälliges Polynom für Restics Chunkingverfahren und einen zufällig gleichverteilten Salt $s \xleftarrow{\$} \mathcal{S}(\lambda)$ für eine passwortbasierte KDF (pbKDF). Die Sicherheit dieser von Restic verwendeten pbKDF hängt maßgeblich von einer guten Wahl des Benutzerpassworts ab. Daher wurde in einem vorherigen Abschnitt dieses Prologs bereits beschrieben, wie die pbKDF durch ein abstrakte KDF für die Sicherheitsspiele ersetzt wurde.

Verschlüsselungsverfahren und Authentifizierungsverfahren:
Die abstrakten Modellierungen sind auf IV-basierten Verschlüsselungsverfahren aufgebaut, daher ist eine Verallgemeinerung zu Verschlüsselungsverfahren, die keine Nonces oder IVs benötigen strukturbedingt nicht möglich. Das stellt jedoch keine weiter Einschränkung dar, sofern man symmetrische Verschlüsselungsprimitive für Restic betrachtet, da für fast alle modernen symmetrischen Verschlüsselungsverfahren ein IV oder eine Nonce benötigt wird. Das Authentifizierungsverfahren ist als MAC-Verfahren mit einer festen MAC-Größe definiert. Wird die Konstante $size_{MAC}$ zu einer Funktion $size_{MAC}(C)$ auf dem Chiffrat zu dem der MAC berechnet wurde angepasst, lassen sich ebenfalls MAC-Verfahren mit unterschiedlich großen MACs für unterschiedliche Chiffrate modellieren.

Die einzigen Stellen an denen Restics Verschlüsselungsverfahren und Authentifizierungsverfahren verwendet wird, ist in den Funktionen 16 und 17. Diese Funktionen werden für die vorgestellten Sicherheitsspiel abstrahiert zu den Funktionen 14 und 15. Die Funktion $l(\lambda) : \mathbb{N} \rightarrow \mathbb{N}$ gibt die Bitlänge eines IVs aus dem IV-Raum $\mathcal{IV}(\lambda)$ abhängig von λ an. Damit gilt $size_{IV} := \frac{l(\lambda)}{16}$.

Funktion 14 Restics Verschlüsselung und Authentifizierung für abstrakte Primitive

Require: k_{enc}, k_{auth}
  function $encrypt_\lambda(plaintext)$
    $Byte[size_{IV}]\ iv \xleftarrow{\$} \mathcal{IV}(\lambda)$
    $Bytestring\ ciphertext := Enc^{iv}(plaintext, k_{enc})$
    $Byte[size_{MAC}(ciphertext)]\ mac := Auth(ciphertext, k_{auth}, iv)$
    return $iv||ciphertext||mac$
  end function

Funktion 15 Restics Verfikation und Entschlüsselung für abstrakte Primitive

Require: k_{enc}, k_{auth}
  function $decrypt_\lambda(encryptedData)$
    $Int\ size := size(encryptedData)$
    $Byte[size_{IV}]\ iv := encryptedData[: size - size_{IV}]$
    $Int\ sizeOfMac := ((g)etSizeOfMac(encryptedData))$
    $Bytestring\ ciphertext := encryptedData[size_{IV} : sizeOfMac]$
    $Byte[sizeOfMac]\ mac := Auth(ciphertext, k_{auth}, iv)$
    if $mac == encryptedData[size - sizeOfMac :]$ **then** ▷ Vergleich der MACs
      return $Dec^{iv}(ciphertext, k_{enc})$
    else
      return $error$
    end if
  end function

5.2.4. Uniformitätslemma

Für die IND$ Sicherheit von Restic werden immer wieder uniforme Bytestrings betrachtet. Uniforme Bytestrings sind Bytestrings, die zufällig gleichverteilt gezogen werden. Das folgende Lemma zeigt, dass die Konkatenation uniformer Bytestrings ebenfalls ein uniformer Bytestring ist. Die Aussage des Lemmas gilt auch für die Umkehrung.

Lemma 1 (Uniformitätserhaltung bei Konkatenation) *Seien* $X \xleftarrow{\$} \{0,1\}^n$, $Y \xleftarrow{\$} \{0,1\}^m$ *zufällig gleichverteilt gezogene Bytestrings. Dann ist die Konkatenation* $X||Y$ *ebenfalls gleichverteilt auf* $\{0,1\}^{n+m}$.

Es sei $Z := X||Y$. Gezeigt wird, dass jedes $z \in \{0,1\}^{n+m}$ mit gleicher Wahrscheinlichkeit der Wert von Z ist.
Es seien $x \in \{0,1\}^n$, $y \in \{0,1\}^m$ beliebig und $z := x||y$. Dann gilt $Pr[Z = z] = Pr[X||Y = x||y]$. Da X und Y unabhängig sind gilt $Pr[Z = z] = Pr[X = x] \cdot Pr[Y = y] = \frac{1}{2^n} \cdot \frac{1}{2^m} = \frac{1}{2^{n+m}}$.
Damit gilt, dass auch Z gleichverteil auf $\{0,1\}^{n+m}$ ist. □

5.3. Vertraulichkeit

Dieses Kapitel analysiert die Vertraulichkeit von Restic bezüglich der drei betrachteten Restic-Befehle *backup*, *restore* und *prune*. Dazu wird ein simulationsbasiertes Sicherheitsspiel $Exp^{IND\$}_{Restic}$ für Restic definiert. Für dieses Kapitel werden die Aussagen und Konstrukte aus Kapitel 5.2 vorausgesetzt.
Es wird ein allgemeiner Reduktionsbeweis für $Exp^{IND\$}_{Restic}$ durchgeführt, der als Hybridargument geführt wird, wodurch sich eine Gesamtabschätzung aus Sicherheitsschranken für abstrakte kryptografische Primitive ergeben. Zum Schluss werden wie bei der Vertraulichkeit konkrete Sicherheitsprimitive mit konkreten Sicherheitsschranken in die Gesamtabschätzung der EUF-CMA-NR Sicherheit von Restic eingesetzt.

5.3.1. Sicherheitsspiel

Dieses Kapitel stellt ein simulationsbasiertes Sicherheitsspiel für die Vertraulichkeit von Restic vor. Dabei werden Restics *backup* Befehl, *restore* Befehl und *prune* Befehl betrachtet. Der Ablauf des Sicherheitsspiels für einen beliebigen Restic-Befehl wird beschrieben. Dabei wird die Leakage-Funktion $\mathcal{L}_R$ aus Definition 5.1.3 verwendet. Außerdem wird das Vorgehen zur Generierung eines Event-Tupels τ_1 mit gleicher Leakage, wie ein anderes Event-Tupel τ_0 erklärt.
Als Erstes werden jedoch die zwei Teilnehmer und das Orakel des Sicherheitsspiels vorgestellt.

5.3.1.1. Teilnehmer

Bei dem vorgestellten Sicherheitsspiel gibt es zwei Teilnehmer.
Ein Teilnehmer ist der Challenger $C(1^\lambda)$, der einen Angreifer herausfordert, zu entscheiden, ob das Sicherheitsspiel in der realen Welt oder in der idealen Welt stattfindet. Der andere Teilnehmer ist der Angreifer $\mathcal{A}^{O_{uk,b}}(1^\lambda, c_{mk}, m_{mk}) \rightarrow z$, der versucht das Sicherheitsspiel zu gewinnen, indem er die korrekte Welt errät. Der Angreifer hat dabei Zugriff auf eine Orakel $O_{uk,b}$, das von $\mathcal{A}$ gewählte Restic-Befehle mit dem Userkey uk ausführen kann und dem Angreifer entweder ein Event-Tupel aus der realen oder idealen Welt zurückgibt. Außerdem erhält der Angreifer das Chiffrat $c_{mk} := Enc_{uk_{enc}}(mk)$ des Masterkeys der realen Welt und den MAC $m_{mk} := Auth_{uk_{auth}}(c_{mk})$ des verschlüsselten Masterkeys als explizite Eingabe. Dies ist damit Begründet, dass $\mathcal{A}$ vollen Lesezugriff auf das Repository besitzt und der Masterkey, der einzige Klartext ist, der mit uk verschlüsselt wird. Dadurch, dass $mk \xleftarrow{\$} \mathcal{K}(\lambda)$ zufällig

gleichverteilt gewählt wird, liefert c_{mk} keine Informationen über mk. Somit wird für den *data* Eintrag in der Key-Datenstruktur keine weiter Zwischenwelt für den Reduktionsbeweis benötigt.
Wie in Kapitel 2.3.1.5 beschrieben, kann eine Aufteilung von Restic in ein Resticsystem S_{Restic} und ein Backupsystem S_{Backup} vorgenommen werden. Dieser Angreifer $\mathcal{A}$ aus dem Sicherheitsspiel repräsentiert einen realen nicht manipulierenden Angreifer, der Lesezugriff auf S_{Backup} hat. Damit kann ein solcher realer Angreifer alle ausgetauschten Nachrichten zwischen S_{Restic} und S_{Backup} sehen. In dem vorgestellten Sicherheitsspiel gibt es jedoch keinen Nachrichtenaustausch zwischen zwei Systemen. Stattdessen wird $\mathcal{A}$ mitgeteilt, welche Nachrichten zu welchen Zeitpunkten zwischen S_{Restic} und S_{Backup} bei der Ausführung eines Restic-Befehls ausgetauscht werden würden. Dies geschieht über das Event-Tupel τ eines jeden Restic-Befehls, das äquivalent zur Beobachtung eines Restic-Befehls auf S_{Backup} ist (siehe Theorem 1). Wie in Kapitel 3.2.5.5 beschrieben, ist die Auswirkung jedes Events e_i auf ein Repository eindeutig durch den Typ op_i definiert. Ein Repository wird außer durch die ausgetauschten Nachrichten während der Ausführung eines Restic-Befehls nicht verändert. $\mathcal{A}$ erhält nach jeder vollständigen Ausführung eines Restic-Befehls das zugehörige Event-Tupel und den Repository-Zustand nach der Ausführung dieses Restic-Befehls. $\mathcal{A}$ erhält außerdem zu Beginn des Sicherheitsspiels den initialen Repository-Zustand des Repositorys auf dem die Restic-Befehle ausgeführt werden. Somit ist der betrachtete Angreifer $\mathcal{A}$ des Sicherheitsspiels gleichwertig zu einem realen Angreifer, der Lesezugriff auf S_{Backup} besitzt.
Der Angreifer hat Zugriff auf das Orakel $O_{uk,b}$, das einen vom Angreifer gewählten Restic-Befehl mit dem Userkey uk ausführt. $O_{uk,b}$ erzeugt nach der entsprechenden abstrakten Modellierung aus Kapitel 3 ein Event-Tupel $\tau_{0,t}$ für diesen Restic-Befehl. Damit stellt $O_{uk,b}$ sowohl S_{Restic} und S_{Backup} gleichzeitig dar und $O_{uk,b}$ übermittelt nur noch das Event-Tupel und den Repository-Zustand an $\mathcal{A}$. Dazu besitzt nur $O_{uk,b}$ Zugriff auf das Repository R_0, für das der Restic-Befehl ausgeführt werden soll. Das erzeugte Event-Tupel $\tau_{0,t}$ wird t-tes Event-Tupel der realen Welt genannt.
Außerdem gibt es ein weiteres Orakel O^{Sim}. Dieses Simulations-Orakel O^{Sim} kommuniziert nur mit $O_{uk,b}$ und dem Challenger. O^{Sim} erzeugt aus der Leakage von $\tau_{0,t}$ ein $\tau_{1,t}$ mit der gleichen Leakage, das als Event-Tupel der idealen Welt bezeichnet wird. Dazu besitzt O^{Sim} Zugriff auf einen Zufallszahlengenerator $\mathcal{R}$ mit dem beliebig lange, zufällige, gleichverteilte Bytestrings (uniform) generiert werden können.

5.3.1.2. Prämisse

Es sei daran erinnert, dass die von Restic verwendete pbKDF für das Sicherheitsspiel durch eine abstrakte PRF-sichere KDF ersetzt wurde, um einen Benutzer zu simulieren, der ein Benutzerpasswort mit entsprechend hoher Entropie wählt. Das heißt es wird kein Passwort von einem Benutzer gewählt, sondern zu Beginn vom Challenger ein uniformer Schlüssel für die abstrakte KDF gezogen.

5.3.1.3. Initialisierung

Das vorgestellte Sicherheitsspiel betrachtet ein zu Beginn leeres Repository R_0. Dazu wird durch den Challenger C durch die Ausführung des *init* Befehls von Restic ein neues Repository R_0 erstellt. Der *init* Befehl wählt einen Masterkey mk bestehend aus zwei uniformen Schlüsseln $mk_{enc} \xleftarrow{\$} \mathcal{K}_{enc}(\lambda)$ und $mk_{auth} \xleftarrow{\$} \mathcal{K}_{auth}(\lambda)$ und $mk := mk_{enc}||mk_{auth}$, sowie einen Salt $s \xleftarrow{\$} \mathcal{S}(\lambda)$. mk_{enc} wird für Restics Verschlüsselungsverfahren benutzt und wird aus dem Schlüsselraum für das Verschlüsselungsverfahren mit $\mathcal{K}_{enc}(\lambda)$ uniform gezogen. mk_{auth} wird für Restics Authentifizierungsverfahren benutzt und wird aus dem Schlüsselraum für das Authentifizierungsverfahren mit $\mathcal{K}_{auth}(\lambda)$ uniform gezogen. Außerdem erzeugt der *init* Befehl eine Config-Datei C_0 für R_0. Der *init* Befehl würde mit dem Passwort eines Benutzers und der pbKDF ein Userkey ableiten, mit dem der Masterkey von R_0 verschlüsselt wird. Die pbKDF wurde für die Sicherheitsspiel jedoch durch ein abstrakte KDF ersetzt, wodurch nun der Challenger einen uniformen Schlüssel $k_{KDF} \xleftarrow{\$} \mathcal{K}_{kdf}(\lambda)$ aus dem Schlüsselraum für die KDF wählt. Mit der abstrakten KDF leitet C einen Ersatz für den Userkey $uk := KDF_{k_{KDF}}(1^\lambda, s)$ her, für den gilt $uk = uk_{enc}||uk_{auth}$. Nach Definition der KDF sind die beiden Userkeys uk_{enc} und uk_{auth} ebenfalls uniform. Der Masterkey mk wird mit uk verschlüsselt und authentifiziert und in den *data* Eintrag der Key-Datenstruktur K geschrieben. Die Key-Datei K wird daraufhin ebenfalls dem Repository R_0 hinzugefügt.
C wählt einen Wert $b \xleftarrow{\$} 0, 1$, der bestimmt, ob das Sicherheitsspiel in der realen Welt (b=0) oder der idealen Welt (b=1) stattfindet. R_0 ist das Repository der realen Welt und der Zustand von R_0 nach der Initialisierung durch den Challenger wird initialer Zustand $R_{0,0}$ genannt. Der Challenger benutzt $\mathcal{R}$, um einen uniformen Bytestring $C_1 \xleftarrow{\$} \mathcal{R}(size(C_0))$ zu generieren. Dann erzeugt C basierend auf R_0 das initiale Repository R_1 für die ideale Welt, indem er C_0 durch C_1 ersetzt. Die Key-Datenstruktur bleibt vollkommen identisch mit der Key-Datenstruktur der realen Welt, da der Angreifer explizit den *data* Eintrag der Key-Datenstruktur als Parameter übergeben bekommen hat. Damit ist R_1 der initiale Repository Zustand $R_{1,0}$ der idealen Welt.
C schickt eine Kopie des initialen Repository-Zustands $R_{b,0}$ an $\mathcal{A}$, damit $\mathcal{A}$ den Zustand des Repositorys vor der Ausführung jeglicher Restic-Befehle sieht. Der Challenger C initialisiert das Orakel O^{Sim} mit R_1. Außerdem initialisiert der C das Orakel $O_{uk,b}$ mit uk, b und R_0 und übergibt dieses als Black-Box ebenfalls an den Angreifer $\mathcal{A}$.

5.3.1.4. Ablauf des Sicherheitsspiels

Beim Ablauf des Sicherheitsspiels übernimmt $\mathcal{A}$ die Rolle des Benutzers aus den abstrakten Modellierungen 3.2, 3.4 und 3.6. Damit schickt $\mathcal{A}$ selbst das Tupel B eines Restic-Befehls und dessen Typ $type(B)$, das einen Restic-Befehl repräsentiert, an $O_{uk,b}$ und somit an S_{Restic}. $O_{uk,b}$ führt den Restic-Befehl B bezüglich des Repositorys R_0 aus und ermittelt damit ein Event-Tupel $\tau_{0,t} := events_{uk}(B, type(B), R_0)$, das den gesamten Nachrichtenaustausch der abstrakten Modellierung des jeweiligen Restic-Befehls enthält. Findet das Sicherheitsspiel in der realen Welt statt, schickt $O_{uk,b}$ dieses $\tau_{0,t}$ zusammen mit dem Repository-Zustand

$R_{0,t}$ nach Ausführung von $\tau_{0,t}$ auf R_0 an $\mathcal{A}$. In der idealen Welt muss $O_{uk,b}$ erst die Leakage $\mathcal{L}_{R_0}(\tau_{0,t})$ bezüglich dem Restic-Befehl von $\mathcal{A}$ und dem Repository R_0 bestimmen. Daraufhin berechnet O^{Sim} ein Event-Tupel $\tau_{1,t}$ in der idealen Welt, sowie den Repository-Zustand $R_{1,t}$ in der idealen Welt und es wird $(\tau_{1,t}, R_{1,t})$ statt $(\tau_{0,t}, R_{0,t})$ an $\mathcal{A}$ geschickt. Diesen Ablauf, darf $\mathcal{A}$ für beliebig viele Restic-Befehle wiederholen, bis $\mathcal{A}$ eine Vermutung für den Wert von b an den Challenger C schickt.
Das Sicherheitsspiel für die Vertraulichkeit von Restic wird in Diagramm 5.1 vorgestellt. Eine Durchführung dieses Sicherheitsspiels bezüglich eines Angreifers $\mathcal{A}$ wird mit $Exp^{IND\$}_{Restic}(\mathcal{A}, 1^\lambda)$ bezeichnet.

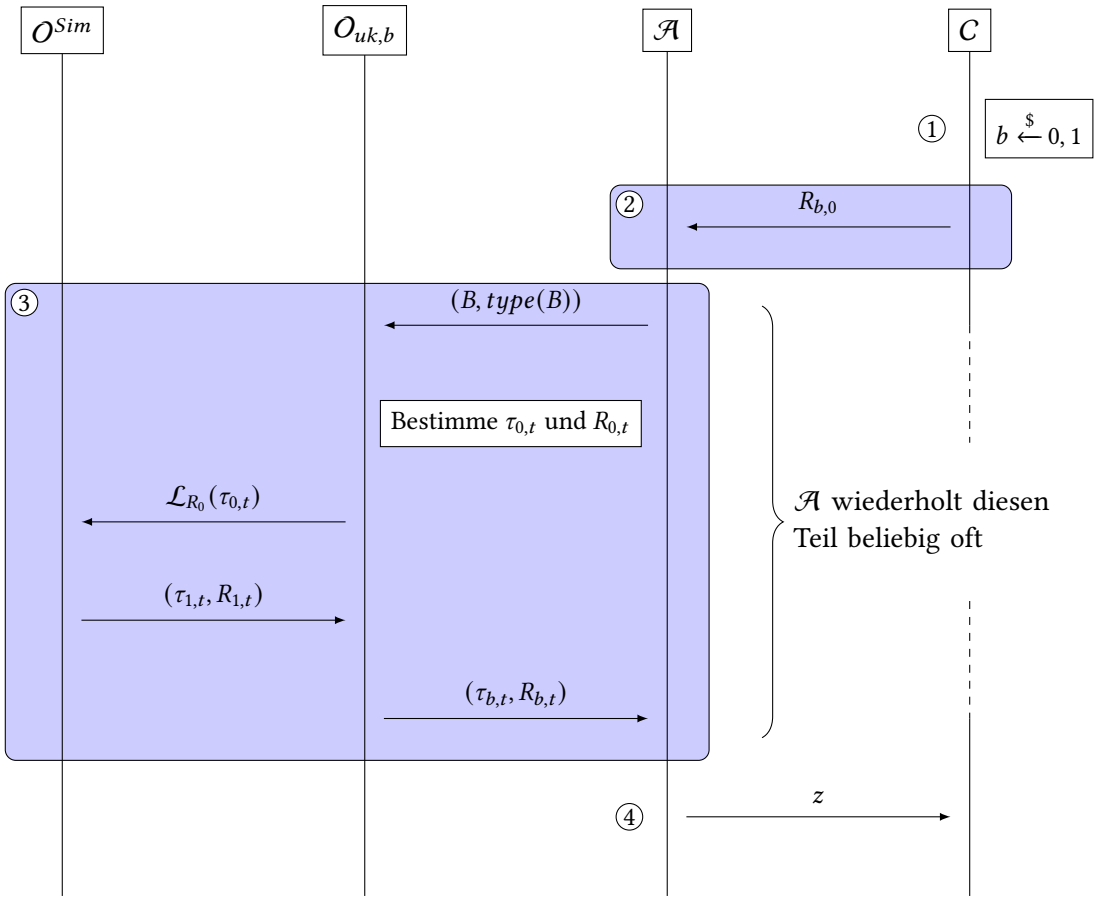

Figure 5.1.: Sicherheitsspiel $Exp^{IND\$}_{Restic}(\mathcal{A}, 1^\lambda)$ für die Vertraulichkeit von Restic bezüglich eines Angreifers $\mathcal{A}$

1. C wählt zufällig ein gleichverteiltes $b \in \{0, 1\}$, das entscheidet, in welcher Welt das Sicherheitsspiel stattfindet.
2. Wie in Kapitel 5.3.1.3 erklärt, initialisiert C das Repository R_0 auf S_{Restic}, das für das gesamte Sicherheitsspiel und alle restlichen Restic-Befehle in der realen Welt verwendet wird. Analog dazu initialisiert C das Repository R_1 für die ideale Welt. C initialisiert das Orakel $O_{uk,b} := Init(1^\lambda, uk, b, R_0)$ mit dem Repository R_0, dem Userkey uk und dem gewählten b. Außerdem initialisiert C das Orakel $O^{Sim} := Init(1^\lambda, R_1, \mathcal{R})$

mit dem Repository R_1 und dem Zufallszahlengenerator $\mathcal{R}$. Da neu initialisierte Repositorys nur eine Config und einen Key enthalten, sind R_1 und R_0 identisch bis auf ihre unterschiedliche Config-Datenstruktur. Basierend auf dem zu Beginn gewählten b wählt C das zugehörige Repository R_b aus und schickt dieses an $\mathcal{A}$.

3. $\mathcal{A}$ darf ein Befehlstupel B für einen beliebigen der drei betrachten Restic-Befehle und dessen Parameter wählen. Dieses Befehlstupel B übergibt $\mathcal{A}$ zusammen mit dem Befehlstyp $type(B)$ an sein Orakel $O_{uk,b}(B, type(B), R_0)$. $O_{uk,b}$ führt den Restic-Befehl B vom Typ $type(B)$ auf dem Repository R_0 aus und erzeugt ein Event-Tupel $\tau_{0,t} := events_{uk}(B, type(B), R_0)$, das den kompletten Nachrichtenaustausch der abstrakten Modellierung des Restic-Befehls beschreibt. Außerdem bestimmt $O_{uk,b}$ den Repository-Zustand $R_{0,t}$ von R_0 nach der Ausführung des Befehls B auf R_0. Daraufhin bestimmt $O_{uk,b}$ für den Befehl B die Leakage $\mathcal{L}_{R_0}(\tau_{0,t})$. $O_{uk,b}$ schickt die bestimmte Leakage $\mathcal{L}_{R_0}(\tau_{0,t})$ an O^{Sim}.
   O^{Sim} generiert aus der Leakage $\mathcal{L}_{R_0}(\tau_{0,t})$ ein Event-Tupel $\tau_{1,t}$ für die ideale Welt und bestimmt den Repository-Zutand $R_{1,t}$ von R_1 für die Ausführung des Event-Tupels $\tau_{1,t}$ auf R_1. O^{Sim} schickt $(\tau_{1,t}, R_{1,t})$ an $O_{uk,b}$ zurück. Basierend auf dem zu Beginn gewählten b wählt $O_{uk,b}$ das zugehörige $(\tau_{b,t}, R_{b,t})$ und gibt dieses an $\mathcal{A}$ zurück.
   $\mathcal{A}$ darf diesen Ablauf für beliebig viele Befehlstupel wiederholen.
4. Nachdem $\mathcal{A}$ diesen Ablauf für beliebig viele Restic-Befehle wiederholt hat, trifft $\mathcal{A}$ eine Entscheidung. $\mathcal{A}$ stellt eine Vermutung für den Wert von b auf und wählt basierend darauf eine Zahl $z \in \{0, 1\}$. $\mathcal{A}$ schickt seine Vermutung $z := \mathcal{A}^{O_{uk,b}}(1^\lambda, c_{mk}, m_{mk})$ an den Challenger C. Entspricht die von $\mathcal{A}$ gewählte Zahle z dem Wert von b, gewinnt $\mathcal{A}$ das Sicherheitspiel. Es gilt $Exp^{IND\$}_{Restic}(\mathcal{A}, 1^\lambda) = 1$ genau dann, wenn $b = z$.
   Andernfalls verliert $\mathcal{A}$ das Sicherheitsspiel und es gilt $Exp^{IND\$}_{Restic}(\mathcal{A}, 1^\lambda) = 0$.

5.3.1.5. Generierung eines Event-Tupels $\tau_{1,t}$ aus $\mathcal{L}_{R_0}(\tau_{0,t})$

O^{Sim} erhält für jedes Event $e_{0,i}$ aus $\tau_{0,t} = (e_{0,1}, \ldots, e_{0,max})$ die Leakage bezüglich $e_{0,i}$ und generiert daraus ein Event $e_{1,i}$ mit $\mathcal{L}_{R_0}(e_{0,i}) = \mathcal{L}_{R_1}(e_{1,i})$. Das Event-Tupel $\tau_{1,t} := (e_{1,1}, \ldots, e_{1,max})$ besteht genau aus diesen generierten Events $e_{1,i}$.
Aus der Leakage eines Events $e_{0,i} = (op_{0,i}, delay_{0,i}, type_{0,i}, data_{0,i}, blobInfo_{0,i})$ wird $e_{1,i}$ wie folgt generiert. Es gilt $op_{1,i} := op_{0,i}$, $delay_{1,i} := delay_{0,i}$ und $type_{1,i} := type_{0,i}$. Gilt $op_{0,i} = send$ und $type_{0,i} \notin \{Pack, Key\}$, dann benutzt O^{Sim} seinen Zufallszahlengenerator $\mathcal{R}$ und generiert einen uniformen Bytestring der Länge $size(data_{0,i})$. Dieser Bytestring wird für $data_{1,i} \xleftarrow{\$} \mathcal{R}(size(data_{0,i}))$ verwendet. Gilt $op_{0,i} = send$ und $type_{0,i} = Pack$, dann benutzt O^{Sim} $\mathcal{R}$ zum Generieren eines uniformen Bytestrings $B2$ der Länge $packHeaderSize_{0,i}$ und eines uniformen Bytestrings $B1$ der Länge $size(data_{0,i}) - (packHeaderSize_{0,i} + 4)$. Damit ergibt sich $data_{1,i} := B1||B2||packHeaderSize_{0,i}$, wobei $packHeaderSize_{0,i}$ genau die letzten vier Bytes von $data_{1,i}$ ist. Diese Unterteilung in zwei generierte Bytestrings wird vorgenommen, damit der IV des Pack-Headers nicht null ist, wodurch $\mathcal{A}$ das Sicherheitsspiel trivial gewinnen könnte (siehe Disclaimer unten).
Die Referenzfunktion $ref(t, i)$, die auf ein früheres Event eines früheren Event-Tupels verweist, ist ebenfalls Teil der Leakage von $e_{0,i}$ aus $\tau_{0,t}$. Damit diese Referenz in die ideale Welt

übertragen werden kann, muss sich $\mathcal{O}^{Sim}$ alle generierten $\tau_{1,t}$ für dieses Sicherheitsspiel merken. Damit $e_{0,i}$ und $e_{1,i}$ die gleiche Leakage besitzen, müssen auch die Werte der Referenzfunktion für beide Events identisch sein. Gilt also $ref(t,i) = (j,s)$, dann muss nach Definition der Referenzfunktion sowohl gelten $data_{0,i} = data_{0,j}$, also auch gelten $data_{1,i} = data_{1,j}$. Dazu muss sich $\mathcal{O}^{Sim}$ alle erstellten Event-Tupel τ_1 merken. Sind $start$ und end Teil der Leakage von $e_{0,i}$, muss in $e_{1,i}$ ebenfalls $blobInfo_{1,i}$ verwendet werden. Dazu werden die Werte von $start$ und end übernommen, sodass gilt $blobInfo_{1,i} := blobInfo_{0,i}$.

Disclaimer bezüglich zufallsgenerierten Chiffraten:
Restic verwendet zur Verschlüsselung AES256 im Counter-Mode. Auch wenn das Verschlüsselungsverfahren von Restic erst zu einem späteren Zeitpunkt betrachtet wird, muss für den Zufallszahlengenerator $\mathcal{R}$ eine Einschränkung getroffen werden. Restic verwendet für Chiffrate als IVs zufällig generierte Nonces, die vor die verschlüsselte Datenstruktur konkateniert werden. Ans Ende jeder verschlüsselten Datenstruktur wird ein MAC konkateniert. Damit besteht eine verschlüsselte und authentifizierte Datenstruktur aus einem zufällig gezogenen IV, dem Chiffrat der Datenstruktur und einem MAC. Der IV in Restic wird zufällig, gleichverteilt aus dem Intervall 1 bis $2^{128} - 1$ gezogen, darf jedoch nicht die Zahl 0 annehmen. Damit ist eine verschlüsselte und authentifizierte Datenstruktur nicht von einem zufälligen Bytestring gleicher Länge unterscheidbar, solange die ersten Bytes dieses Bytestrings, die der Länge eines IVs entsprechen, nicht alle 0 sind und der Angreifer keine Möglichkeit hat den MAC zu verifizieren.

5.3.1.6. Erfolgswahrscheinlichkeit

Die Wahrscheinlichkeit, dass $\mathcal{A}$ das Sicherheitsspiel gewinnt ist $Pr[Exp^{IND\$}_{Restic}(\mathcal{A}, 1^\lambda) = 1] := Pr[z = b]$. Damit ergibt sich für die Erfolgswahrscheinlichkeit des Angreifers die Formel 5.1. Bei der Erfolgswahrscheinlichkeit für dieses Sicherheitsspiel muss $\frac{1}{2}$ abgezogen werden, da der Angreifer durch zufälliges gleichverteiltes Raten der Zahl z eine Wahrscheinlichkeit von $\frac{1}{2}$ besitzt, dass gilt $b = z$.

$$Adv^{IND\$}_{Restic}(\mathcal{A}, \lambda) := |Pr[Exp^{IND\$}_{Restic}(\mathcal{A}, 1^\lambda) = 1] - \frac{1}{2}| \tag{5.1}$$

5.3.2. Teil-Reduktion auf die allgemeine IND$ Sicherheit

Um die Vertraulichkeit von Restic unter der unvermeidlichen Leakage zu beweisen, ist ein Teil des Reduktionsbeweises zu zeigen, dass der Unterscheidungswahrscheinlichkeit zweier Zwischenwelten $\mathcal{W}_2$ und $\mathcal{W}_3$ beschränkt ist durch die IND$ Sicherheit des abstrakten Verschlüsselungsverfahren. Es empfiehlt sich zuerst das Kapitel 5.3.3.1 zu lesen, bis es an die Reduktion von $\mathcal{W}_2$ zu $\mathcal{W}_3$ geht, da dieses Kapitel diese Reduktion modelliert.
Dazu wird eine Abwandlung des IND$ Sicherheitsspiel $Exp^{IND\$}_{Restic}$ auf ein IND$ Sicherheitsspiel bezüglich Restics abstrakten Verschlüsselungsverfahren reduziert. Diese Abwandlung von Restics IND$ Sicherheitsspiels wird $Exp^{IND\$-NoMac}_{Restic}(\mathcal{A}, 1^\lambda)$ genannt. $\mathcal{W}_2$

entspricht der realen Welt von $Exp_{Restic}^{IND\$-NoMac}(\mathcal{A}, 1^\lambda)$ und $\mathcal{W}_3$ entspricht der idealen Welt. Es wird ein Angreifer $\mathcal{A}$ für das abgewandelte Sicherheitsspiel $Exp_{Restic}^{IND\$-NoMac}(\mathcal{A}, 1^\lambda)$ betrachtet, wobei λ der asymptotische Sicherheitsparameter ist. Daraus wird ein Angreifer $\mathcal{B}$ für ein IND$ Sicherheitsspiel $Exp_{Enc}^{IND\$}(\mathcal{B}, 1^\lambda)$ für ein von Restic verwendetes abstraktes Verschlüsselungsverfahren konstruiert. Es wird gezeigt und bewiesen, wie dieser Angreifer $\mathcal{B}$ das IND$ Sicherheitsspiel gewinnt, wenn $\mathcal{A}$ das Sicherheitsspiel $Exp_{Restic}^{IND\$-NoMac}$ gewinnt.

5.3.2.1. Änderungen bei $Exp_{Restic}^{IND\$-NoMac}$

Bei dieser Reduktion simuliert $\mathcal{B}$ nicht das exakte Sicherheitsspiel $Exp_{Restic}^{IND\$}$ für den Angreifer $\mathcal{A}$, sondern eine abgewandelte Form dieses Sicherheitsspiels. Folgende Besonderheiten gelten für das von $\mathcal{B}$ simulierte Sicherheitsspiel $Exp_{Restic}^{IND\$-NoMac}$.

- Der Userkey für ein Repository wird nicht durch eine KDF berechnet, sondern uniform gezogen $uk \xleftarrow{\$} \mathcal{K}_{user}(\lambda)$.
- Änderungen in der realen Welt:
  - IVs werden ebenfalls uniform aus dem kompletten IV-Raum $IV \xleftarrow{\$} \mathcal{IV}(\lambda)$ gezogen (inklusive $0^{l(\lambda)}$).
- Änderungen in der Leakage-Funktion:
  - Die Blob-Grenzen in $data_{0,i}$ für ein Event mit $type_{0,i} = Pack$ sind ebenfalls Teil der Leakage.
  - Für $type_{0,i} \neq Pack$ ist der IV aus $data_{0,i}$ ebenfalls Teil der Leakage.
  - Für $type_{0,i} = Pack$ sind die IVs aller Blobs und Pack-Header aus $data_{0,i}$ ebenfalls Teil der Leakage.
- Änderungen in der idealen Welt:
  - Es wird für jedes Event mit $op_{0,i} = send$ und $type_{0,i} \neq Pack$ ein uniformer Bytestring UB gezogen, der dieselbe Länge, wie das Chiffrat C in $data_{0,i} = IV||C||MAC_C$ besitzt.
  - Aus UB und durch die veränderte Leakage-Funktion wird $data_{1,i} := IV||UB||MAC_{UB}$ konstruiert.
  - Für Events mit $type_{0,i} = Pack$ und $data_{0,i} = IV_1||C_1||MAC_{C_1}|| \ldots ||IV_m||C_m||MAC_{C_m}$ wird $data_{1,i} := IV_1||UB_1||MAC_{UB_1}|| \ldots ||IV_m||UB_m||MAC_{UB_m}$ konstruiert.

Damit stellt $Exp_{Restic}^{IND\$-NoMac}$ ein Sicherheitsspiel dar, in dem in der idealen Welt nur die Chiffrate durch uniforme Bytestrings ersetzt wurden und weiterhin echte MACs über den uniformen Bytestrings berechnet werden.

5.3.2.2. Teilnehmer

Für die Reduktion werden drei Teilnehmer betrachtet. Der erste Teilnehmer ist ein Angreifer $\mathcal{A}$ für das vorgestellte Sicherheitsspiel $Exp^{IND\$-NoMac}_{Restic}(\mathcal{A}, 1^\lambda)$. Der zweite Teilnehmer ist ein Challenger C, der der Challenger für das Sicherheitsspiel $Exp^{IND\$}_{Enc}(\mathcal{B}, 1^\lambda)$ ist, auf das reduziert wird. Außerdem gibt es den Angreifer $\mathcal{B}$, der den Angreifer für das Sicherheitsspiel $Exp^{IND\$}_{Enc}(\mathcal{B}, 1^\lambda)$ darstellt und für $\mathcal{A}$ das Sicherheitsspiel $Exp^{IND\$-NoMac}_{Restic}$ simuliert. Dafür übernimmt $\mathcal{B}$ die meisten Aufgaben von $\mathcal{O}_{uk,b}$ und $\mathcal{O}_{sim}$ aus $Exp^{IND\$-NoMac}_{Restic}$. $\mathcal{B}$ hat außerdem Zugriff auf ein Verschlüsselungsorakel $\mathcal{O}^{Enc}_{k_{enc},b}$ für das betrachtete abstrakte Verschlüsselungsverfahren.
Damit $\mathcal{B}$ die Aufgaben von $\mathcal{O}_{uk,b}$ übernehmen kann, muss $\mathcal{B}$ in der Lage sein die Befehlsausführung eines Restic-Befehls zu emulieren oder $\mathcal{B}$ muss eine Instanz von Restic besitzen, mit der $\mathcal{B}$ sich Restic-Befehle ausführen lassen kann. Dazu ist $\mathcal{B}$ wie auch $\mathcal{O}_{uk,b}$ sowohl S_{Restic}, als auch S_{Backup} in einem. Damit besitzt $\mathcal{B}$ ebenfalls ein Repository R, auf dem alle von $\mathcal{A}$ übergebenen Restic-Befehle ausgeführt werden. Der Unterschied zu dem Repository von $\mathcal{O}_{uk,b}$ ist, dass R hier nicht das Repository R_0 bezüglich der realen Welt ist, sondern nur als Hilfe für $\mathcal{B}$ benutzt wird, um die Event-Tupel τ_0 und τ_1 zu berechnen. Das Repository R besitzt zwar einen Masterkey, aber alle verschlüsselten Datenstrukturen, die $\mathcal{A}$ aus der realen Welt zu sehen bekommt, wurden durch das Orakel $\mathcal{O}^{Enc}_{k_{enc},b}$ verschlüsselt und nicht mit dem Masterkey von R. Der Masterkey von R wird jedoch weiterhin benutzt, um alle Datenstrukturen zu authentifizieren, die $\mathcal{A}$ sieht.
Außerdem besitzt $\mathcal{O}^{Enc}_{k_{enc},b}$ einen Zufallszahlengenerator $\mathcal{R}$, mit dem ein zufälliger, gleichverteilter Bytestring (uniform) beliebiger Länge generiert werden kann.

5.3.2.3. Initialisierung

Der Challenger C wählt einen Wert $b \xleftarrow{\$} \{0, 1\}$, der bestimmt, ob das Sicherheitsspiel $Exp^{IND\$}_{Enc}$ in der realen Welt (b=0) oder der idealen Welt (b=1) stattfindet. C zieht einen uniformen Schlüssel $k_{enc} \xleftarrow{\$} \mathcal{K}_{enc}(\lambda)$ aus dem gleichen Schlüsselraum, wie für den Masterkey mk_{enc}, für das abstrakte Verschlüsselungsverfahren. Der Challenger initialisiert das Verschlüsselungsorakel $\mathcal{O}^{Enc}_{k_{enc},b} := Init(1^\lambda, k_{enc}, b)$ und gibt dieses als Black-Box an $\mathcal{B}$.
$\mathcal{B}$ führt den *init* Befehls von Restic aus, um ein neues Repository R erstellen. Der *init* Befehl wählt einen Masterkey mk bestehend aus zwei uniformen Schlüsseln $mk_{enc} \xleftarrow{\$} \mathcal{K}_{enc}(\lambda)$ und $mk_{auth} \xleftarrow{\$} \mathcal{K}_{auth}(\lambda)$ und $mk := mk_{enc}||mk_{auth}$, sowie einen Salt $s \xleftarrow{\$} \mathcal{S}(\lambda)$. mk_{enc} wird nur von $\mathcal{B}$ verwendet, um aus Restic-Befehlen Event-Tupel zu generieren. Alle Datenstrukturen der Restic-Befehle von $\mathcal{A}$ werden erneut durch das Verschlüsselungsorakel verschlüsselt. Wie auch im IND$ Sicherheitsspiel für Restic wird kein Benutzerpasswort für den *init* Befehl verwendet, sondern $\mathcal{B}$ wählt direkt einen Userkey $uk \xleftarrow{\$} \mathcal{K}_{user}(\lambda)$. Der Masterkey mk wird mit uk verschlüsselt und authentifiziert und in den *data* Eintrag der Key-Datenstruktur K geschrieben. Die Key-Datei K wird daraufhin dem Repository R hinzugefügt. Außerdem erzeugt der *init* Befehl eine Config-Datei C für R. $\mathcal{B}$ verwendet sein Orakel, um den Klartext m_{config} der Config-Datei C zu verschlüsseln und erhält $(iv_{config}, c_{b,config}) \xleftarrow{\$} \mathcal{O}^{Enc}_{k_{enc},b}(m_{config})$.

$\mathcal{B}$ führt das Authentifizierungsverfahren manuell für die Antwort des Orakels durch und erhält einen MAC mac_{config}. Ein initiales Repository ist leer, bis auf eine Key-Datei und eine Config-Datei. Daher erstellt $\mathcal{B}$ ein leeres Repository R' und fügt die Key-Datei K und eine Config-Datei bestehend aus $iv_{config}||c_{b,config}||mac_{config}$ zu R' hinzu.
$\mathcal{B}$ übergibt den Repository-Zustand R' an $\mathcal{A}$.

5.3.2.4. Ablauf der Reduktion

$\mathcal{A}$ darf beliebige Restic-Befehle B wählen und an $\mathcal{B}$ schicken. $\mathcal{B}$ führt den Befehl B auf R aus, um ein Event-Tupel τ_R zu erhalten. Wann immer eine neue Datenstruktur erstellt wird und in einem Send-Event auftaucht $\mathcal{B}$ lässt den Klartext dieser Datenstruktur oder im Falle von Packs, die Klartexte der einzelnen Blobs und Pack-Header durch das Orakel verschlüsseln. Genau, wie für die Config-Datei, ersetzt $\mathcal{B}$ die $data_i$ Einträge aller Send-Events aus τ_R durch die authentifizierten Antworten des Orakels. Jedes Receive-Event und Delete-Event $e_{t,i}$ aus $\tau_{R,t}$ besitzt per Definition eine Referenzfunktion $r(t,i) = (s,j)$, die auf ein früheres Event $e_{s,j}$ verweist. $\mathcal{B}$ ersetzt $data_{t,i}$ eines Receive-Events oder Delete-Events in $\tau_{R',t}$ durch $data_{s,j}$ aus dem Event $e_{s,j} \in \tau_{R',s}$. $\mathcal{B}$ führt das konstruierte Event-Tupel $\tau_{R',t}$ auf dem Repository R' aus und übergibt den Repository-Zustand R'_t nach Ausführung des Event-Tupels zusammen mit dem $\tau_{R',t}$ an $\mathcal{A}$.
Das von $\mathcal{B}$ konstruierte Repository R' ist das Repository R_b aus dem Sicherheitsspiel $Exp^{IND\$-NoMac}_{Restic}$. Genauso konstruiert $\mathcal{B}$ die Event-Tupel $\tau_{b,t} := \tau_{R',t}$, ohne Kenntnis über das eigentliche b zu besitzen. Für $\mathcal{A}$ ist kein Unterschied zwischen der Simulation von $\mathcal{B}$ und dem echten Sicherheitsspiel $Exp^{IND\$-NoMac}_{Restic}$ unter der Verwendung des Masterkeys $k_{enc}||mk_{auth}$ zu sehen.

1. Der Challenger wählt zufällig ein gleichverteiltes $b \in \{0,1\}$, das entscheidet, ob das Orakel $O^{Enc}_{k_{enc},b}$ immer den echten Klartext m_i verschlüsselt, oder einen uniformen Bytestring gleicher Länge anstelle des Chiffrats zurückgibt. Wie in Sicherheitsspiel 5.1 bedeutet $b = 0$, dass $O^{Enc}_{k_{enc},b}$ echte Klartexte verschlüsselt und zurückgibt und $b = 1$, dass alle Chiffrate $c_{b,i}$, die $O^{Enc}_{k_{enc},b}$ zurückgibt, uniforme Bytestrings sind. Gilt $b = 0$, dann ist $c_{b,i}$ das Chiffrat von m_i ohne MAC, wie es auch Restic berechnen würde.

   Abgesehen von der Wahl von b generiert C einen Schlüssel $k_{enc} \xleftarrow{\$} \mathcal{K}_{enc}(\lambda)$ für das verwendete Verschlüsselungsverfahren, mit dem $O^{Enc}_{k_{enc},b}$ alle zukünftigen Klartexte m_i verschlüsseln kann.
2. $\mathcal{B}$ initialisiert das Repository R auf S_{Restic}.
3. $\mathcal{B}$ verwendet R zur Konstruktion des initialen Repositorys R' für den Angreifer $\mathcal{A}$. Dazu übergibt $\mathcal{B}$ den Klartext m_{config} der Config-Datenstruktur von R an $O^{Enc}_{k_{enc},b}$.

   Das Orakel zieht einen uniformen $iv_{config} \xleftarrow{\$} \mathcal{IV}(\lambda)$. Basierend auf dem zu Beginn gewählten b verschlüsselt $O^{Enc}_{k_{enc},b}$ den Klartext m_{config} oder generiert einen uniformen Bytestring mit gleicher Länge wie das eigentlich Chiffrat $c_{0,Config}$. Basierend auf b gibt $O^{Enc}_{k_{enc},b}$ den verschlüsselten Klartext oder den generierten Bytestring an $\mathcal{B}$ zusammen mit iv_{config} zurück. $\mathcal{B}$ generiert mit dem Masterkey von R einen MAC mac_{config} für

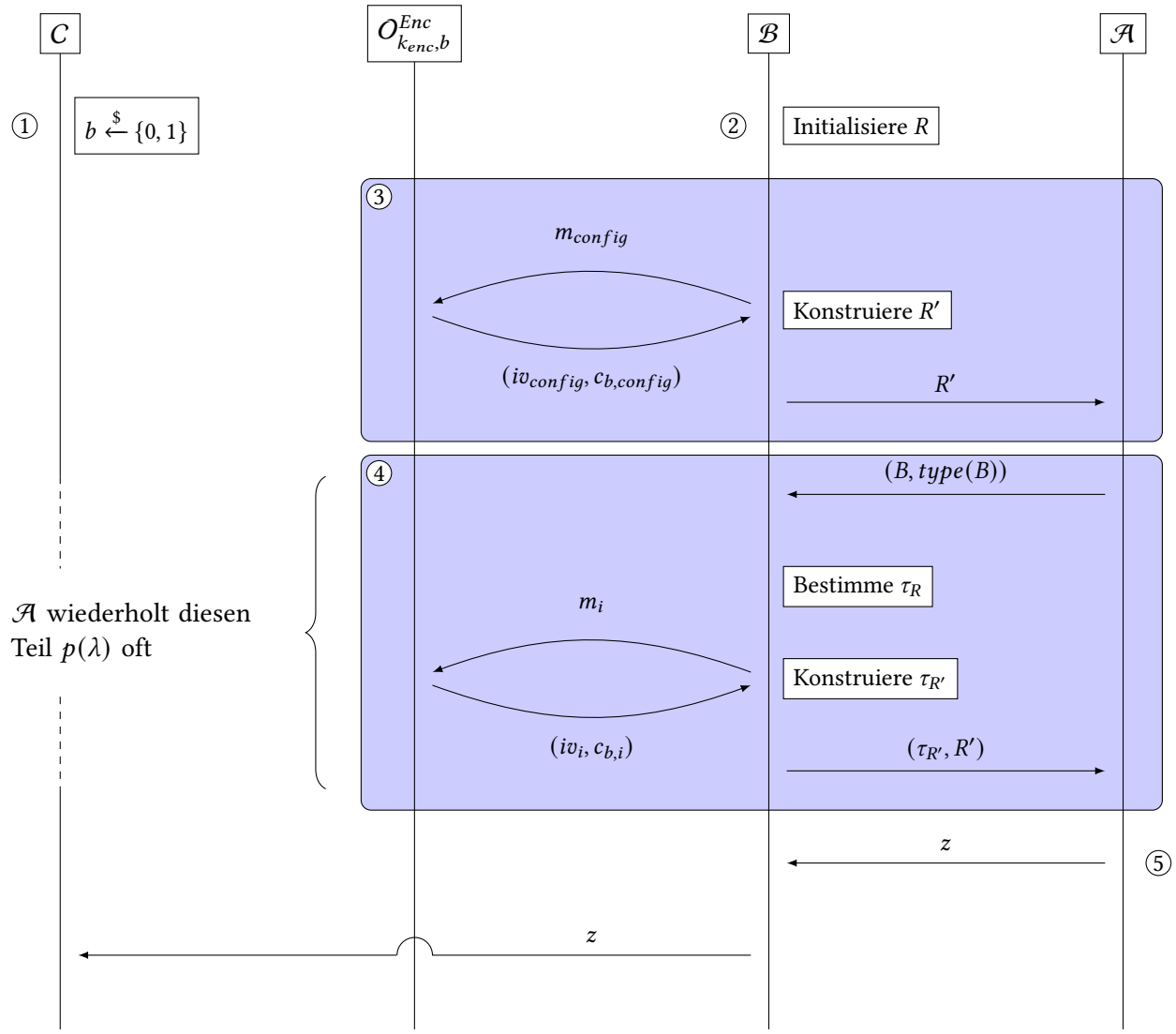

Figure 5.2.: Reduktion eines Angreifers $\mathcal{A}$ für das Sicherheitsspiel $Exp^{IND\$-NoMac}_{Restic}(\mathcal{A})$ auf einen Angreifer $\mathcal{B}$ für die IND$ Sicherheit von Restics Verschlüsselungsverfahren

das erhaltene $c_{b,config}$ und verwendet $iv_{config}||c_{b,config}||mac_{config}$ als Inhalt der Config-Datei von R'. Für die Key-Datenstruktur von R' wird die Key-Datenstruktur von R übernommen. Da ein initiales Repository sonst keine Daten enthält, schickt $\mathcal{B}$ diesen Repository-Zustand R' an $\mathcal{A}$.

4. $\mathcal{A}$ darf ein Befehlstupel B für einen beliebigen der drei betrachten Restic-Befehle und dessen Parameter wählen. Dieses Befehlstupel B schickt $\mathcal{A}$ zusammen mit dem Befehlstyp $type(B)$ an $\mathcal{B}$. $\mathcal{B}$ führt den Befehl B auf R aus, um ein Event-Tupel τ_R zu erhalten. Unter Benutzung des Orakels $O^{Enc}_{k_{enc},b}$ konstruiert $\mathcal{B}$ ein Event-Tupel $\tau_{R'}$. Dieses $\tau_{R'}$ wird von $\mathcal{B}$ auf R' ausgeführt, um den Repository-Zustand R' nach Ausführung von $\tau_{R'}$ zu erhalten. Dann schickt $\mathcal{B}$ $(\tau_{R'}, R')$ an $\mathcal{A}$ zurück.

$\mathcal{A}$ darf diesen Ablauf für $p(\lambda)$ viele Befehlstupel wiederholen, wobei $p(\lambda)$ ein Polynom in λ ist.

5. Nachdem $\mathcal{A}$ diesen Ablauf für $p(\lambda)$ viele Restic-Befehle wiederholt hat, trifft $\mathcal{A}$ eine Entscheidung. $\mathcal{A}$ stellt eine Vermutung für den Wert von b auf und wählt basierend darauf eine Zahl $z \in \{0, 1\}$, die an $\mathcal{B}$ übergeben wird. $\mathcal{B}$ verwendet die gleiche Vermutung z wie $\mathcal{A}$ für b und gibt diese an den Challenger C weiter.
   Entspricht die an C übergebene Zahle z dem Wert von b, hat $\mathcal{B}$ das IND$ Sicherheitsspiel für das von Restic verwendete Verschlüsselungsverfahren gewonnen.

5.3.2.5. Erfolgswahrscheinlichkeit von $Adv_{Enc}^{IND\$}(\mathcal{B}, \lambda)$

$\mathcal{B}$ kann das Sicherheitsspiel $Exp_{Restic}^{IND\$-NoMac}$ mit $k_{enc}||mk_{auth}$ als Masterkey exakt für $\mathcal{A}$ simulieren. Da k_{enc} aus dem gleichen Schlüsselraum $\mathcal{K}_{enc}(\lambda)$ gewählt wird, wie der eigentliche Masterkey-Teil mk_{enc}, kann $\mathcal{B}$ genau $Exp_{Restic}^{IND\$-NoMac}$ simulieren. Damit gewinnt $\mathcal{B}$ ebenfalls sein Sicherheitsspiel, wenn $\mathcal{A}$ das Sicherheitsspiel $Exp_{Restic}^{IND\$-NoMac}$ gewinnt. Daraus folgt:

$$Adv_{Restic}^{IND\$-NoMac}(\mathcal{A}, \lambda) \leq Adv_{Enc}^{IND\$}(\mathcal{B}, \lambda)$$

5.3.2.6. Laufzeit von $\mathcal{B}$

$\mathcal{B}$ führt alle Befehle von $\mathcal{A}$ einmal auf R aus, und einmal mit $O_{k_{enc},b}^{Enc}$ auf R'. Für einen Angreifer $\mathcal{A}$ mit Laufzeit t, ist die Laufzeit t_1 von $\mathcal{B}$ genau das doppelte plus die polynomielle Bedienzeit des Orakels, also $t_1 \leq 2 \cdot t + poly(\lambda)$. Damit ist für alle PPT-Angreifer $\mathcal{A}$ in λ der konstruierte Angreifer $\mathcal{B}$ ebenfalls ein PPT-Angreifer in λ.

5.3.2.7. Fazit

Damit kann ein PPT-Angreifer $\mathcal{A}$ für $Exp_{Restic}^{IND\$-NoMac}(\mathcal{A}, \lambda)$ erfolgreich auf einen PPT-Angreifer $\mathcal{B}$ für $Exp_{Enc}^{IND\$}(\mathcal{B}, \lambda)$ reduziert werden. Somit folgt, die Wahrscheinlichkeit die reale Welt $\mathcal{W}_2$ und ideale Welt $\mathcal{W}_3$ zu unterscheiden, ist durch die Wahrscheinlichkeit beschränkt, die IND$ Sicherheit des verwendeten abstrakten Verschlüsselungsverfahrens zu brechen.

5.3.3. Reduktionsbeweis für die IND$ Sicherheit von Restic

In diesem Kapitel wird die IND$ Sicherheit von Restic auf die Sicherheit der verwendeten kryptografischen Primitive reduziert. Dazu wird gezeigt, dass jeder PPT-Angreifer $\mathcal{A}$ das Sicherheitsspiel $Exp_{Restic}^{IND\$}(\mathcal{A}, 1^{\lambda})$ mit nicht vernachlässigbarer Wahrscheinlichkeit in λ gewinnt, einen Angreifer $\mathcal{B}$ impliziert, der eins der zugrundeliegenden Primitive ebenfalls mit nicht vernachlässigbarer Wahrscheinlichkeit in λ bricht. Der Beweis wird als Hybridargument geführt. Dazu werden drei Zwischenwelten betrachtet ($\mathcal{W}_1$, $\mathcal{W}_2$ und $\mathcal{W}_3$), wobei

$\mathcal{W}_0$ die reale Welt des Sicherheitsspiels $Exp^{IND\$}_{Restic}$ ist und $\mathcal{W}_4$ die ideale Welt von $Exp^{IND\$}_{Restic}$ ist.
Erst wird eine allgemeine Reduktion für abstrakte Sicherheitsprimitive vorgenommen. Danach werden die konkreten Sicherheitsabschätzungen für Restics konkrete Primitive eingesetzt und eine Gesamtabschätzung für Restics IND$ Sicherheit getroffen.

5.3.3.1. Allgemeine Reduktion

In diesem Kapitel werden zunächst nur abstrakte kryptografische Primitive betrachtet. Mithilfe dieser abstrakten Primitive wird eine Gesamtabschätzung in λ für die Sicherheit von Restics IND$ Sicherheit getroffen. Damit kann die IND$ Sicherheit von Restic unabhängig der tatsächlichen kryptografischen Primitive ausgedrückt werden.

Prolog:
Durch die Konstruktion von τ_1 aus τ_0 gilt für alle $e_{b,t,i} \in \tau_{b,t}$ aus allen erzeugten Event-Tupeln $\tau_{b,t}$ folgendes, egal welchen Wert b tatsächlich hat:

- $e_{0,t,i}$ und $e_{1,t,i}$ sind identisch, bis auf $data_{0,t,i}$ und $data_{1,t,i}$
- $size(data_{0,t,i}) = size(data_{1,t,i})$

Dadurch unterscheiden sich zwei Welten nur in den $data_i$ Einträgen aller Events aller erzeugten Event-Tupel.

Reale Welt $\mathcal{W}_0$:
In $\mathcal{W}_0$ besitzen alle $data_i$ für Events mit $type_i \neq Pack$ die Form $IV||Enc^{IV}(M)||MAC$, wobei M der Klartext der Datenstruktur ist. Packs bestehen aus konkatenierten Datenstrukturen (Blobs und Pack-Header) mit der oben beschriebenen Form. Außerdem sind in jeder Welt die letzten vier Bytes von $data_i$ für eine Pack-Datenstruktur unverschlüsselt und identisch zu dem gleichen $data_i$ in jeder anderen Welt. Für eine bessere Übersichtlichkeit werden die letzten vier Bytes bei der Betrachtung von $data_i$ für Events mit $type_i = Pack$ in diesem Reduktionsbeweis und der Struktur von $data_i$ nicht betrachtet. Damit gilt für $type_i = Pack$, dass $data_i$ wie folgt aussieht $data_i := IV_1||C_1||MAC_1|| \ldots ||IV_m||C_m||MAC_m$, wobei dieses Pack $m-1$ Blobs enthält, sowie als letztes Chiffrat den Pack-Header.

Schlüsselableitungsfunktion ($\mathcal{W}_0$ zu $\mathcal{W}_1$):
Schlüsselbasierte kryptografische Verfahren garantieren nur Sicherheit, wenn der verwendete Schlüssel mit hoher Entropie gewählt wurde oder sogar uniform ist. Da in der realen Welt der Userkey durch eine KDF gewählt wird, muss zuerst gezeigt werden, dass die Ausgabe der KDF ununterscheidbar von einem uniform gezogenen Schlüssel ist. Daher findet der erste Schritt von der realen Welt $\mathcal{W}_0$ auf die Welt $\mathcal{W}_1$ statt, in der verwendete KDF durch eine echte Zufallsfunktion ersetzt wurde.

$$\mathcal{W}_0 : uk := KDF_{k_{KDF}}(1^\lambda, s) \qquad \rightarrow \qquad \mathcal{W}_1 : uk \xleftarrow{\$} \mathcal{K}_{user}(\lambda)$$

Die Wahrscheinlichkeit, dass ein Angreifer zwischen $\mathcal{W}_0$ und $\mathcal{W}_1$ unterscheiden kann, beschränkt durch die Wahrscheinlichkeit, dass die verwendete KDF von einer echten Zufallsfunktion unterscheidbar ist. Damit ist die Wahrscheinlichkeit, dass ein Angreifer zwischen $\mathcal{W}_0$ und $\mathcal{W}_1$ unterscheiden kann, beschränkt durch die PRF-Sicherheit $Adv^{PRF}_{KDF}(\mathcal{B}_1, \lambda)$ der verwendeten KDF. Somit ergibt sich die Abschätzung:

$$|Pr[\mathcal{A} \Rightarrow 1 \mid \mathcal{W}_0] - Pr[\mathcal{A} \Rightarrow 1 \mid \mathcal{W}_1]| \leq Adv^{PRF}_{KDF}(\mathcal{B}_1, \lambda)$$

Der Masterkey, der für die Verschlüsselung und Authentifizierung der Datenstrukturen verwendet wird, wird in dem IND$ Sicherheitsspiel und der abstrakten Modellierung von Restic bereits in $\mathcal{W}_0$ zufällig gleichverteilt gezogen.

IV-Raum zur Uniformverteilung erweitern ($\mathcal{W}_1$ zu $\mathcal{W}_2$):
Es sei $\mathcal{IV}(\lambda)$ ein IV-Raum mit $\mathcal{IV}(\lambda) := \{0,1\}^{l(\lambda)}$, wobei $l : \mathbb{N} \rightarrow \mathbb{N}$ eine Funktion abhängig von λ ist.
In der realen Welt und auch in $\mathcal{W}_1$ wird der IV uniform aus dem Raum $\mathcal{IV}(\lambda) \setminus \{0^{l(\lambda)}\}$ gewählt. Im Folgenden wird die Notation $0 := 0^{l(\lambda)}$ verwendet. Damit im nächsten Schritt von $\mathcal{W}_2$ auf die allgemeine IND$ Sicherheit reduziert werden kann, werden in $\mathcal{W}_2$ die IVs uniform aus dem Raum $\mathcal{IV}(\lambda)$ gezogen.

$$\mathcal{W}_0 : IV \xleftarrow{\$} \mathcal{IV}(\lambda) \setminus \{0\} \qquad \rightarrow \qquad \mathcal{W}_1 : IV \xleftarrow{\$} \mathcal{IV}(\lambda)$$

Damit der Schritt von $\mathcal{W}_1$ zu $\mathcal{W}_2$ möglich ist, müssen alle in $\mathcal{W}_2$ verwendeten Sicherheitsprimitive, die die IVs verwenden, auch für $IV = 0^{l(\lambda)}$ definiert sein. Dazu wird eine Fallunterscheidung eingeführt. Im ersten Fall sind sowohl das verwendete Verschlüsselungsverfahren und das Authentifizierungsverfahren für $IV = 0$ definiert, sofern die jeweiligen Verfahren einen IV verwenden. Außerdem wird gefordert, dass die Sicherheitsgarantien der Primitive auf dem IV-Raum $\mathcal{IV}(\lambda)$ mindestens genauso stark sind, wie auf $\mathcal{IV}(\lambda) \setminus \{0\}$. Im zweiten Fall verwendet mindestens ein Primitiv die IVs und ist nicht für $IV = 0$ definiert oder besitzt für den IV-Raum $\mathcal{IV}(\lambda)$ gar keine oder zumindest reduzierte Sicherheitsgarantien. In diesem Fall wird ein für diese Primitive eine Definition für $IV = 0$ hinzugefügt, bei der das resultierende Chiffrat oder der MAC ein Fehlersymbol $\perp$ darstellen. In beiden Fällen ist es einem Angreifer $\mathcal{A}$ nur möglich die Welten $\mathcal{W}_1$ und $\mathcal{W}_2$ zu unterscheiden, wenn in $\mathcal{W}_2$ zu irgendeinem Zeitpunkt der $IV = 0$ gewählt wurde.
Ein IV wird immer dann gezogen, wenn eine Datenstruktur durch Restic verschlüsselt wird. Wenn eine Datenstruktur durch Restic verschlüsselt wird, handelt es sich dabei um eine neue erstellte Datenstruktur, die anschließend auch im Repository gespeichert werden soll. Dementsprechend taucht diese Datenstruktur und auch der verwendete IV in einem Event mit $op_i = send$ auf. $\mathcal{A}$ kann für jede Datenstruktur, für deren Send-Event gilt $type_i \neq Pack$, direkt den verwendeten IV einsehen, da der IV die ersten $l(\lambda)$ Bit von $data_i$ darstellt. Bei einer Pack-Datenstruktur, sieht der Angreifer nur einen konkatenierten Bytestring, aus dem $\mathcal{A}$ nicht ohne weiteres aller IV extrahieren kann. Zum Zeitpunkt des Send-Events für eine Datenstruktur sind die Grenzen der konkatenierten verschlüsselten und authentifizierten Datenstrukturen nicht Teil der Leakage-Funktion $\mathcal{L}$ eines Send-Events für ein Pack. In dem betrachteten Sicherheitsspiel $Exp^{IND\$}_{Restic}$ ist es $\mathcal{A}$ jedoch möglich über weiter Restic-Befehle,

diese Grenzen herauszufinden (siehe Kapitel 5.1.3). Damit besitzt $\mathcal{A}$ in beiden Fällen der Fallunterscheidung die Möglichkeit alle gezogenen IVs zu beobachten und erkennt, ob mindestens ein mal der $IV = 0$ gezogen wurde. Da die beiden Welten bis auf den IV-Raum exakt identisch sind, ist die Wahrscheinlichkeit, mit der ein Angreifer die beiden Welten unterscheiden kann, durch die Wahrscheinlichkeit beschränkt, dass in $\mathcal{W}_2$ der $IV = 0$ mindestens einmal gezogen wird.
Es sei n die Anzahl der Orakelanfragen des Angreifers $\mathcal{A}$ und q_i die Gesamtanzahl der verschlüsselten und authentifizierten Datenstrukturen der i-ten Orakelanfrage. Es sei $q(\lambda) := \sum_{i=1}^{n} q_i$ die Gesamtanzahl der verschlüsselten und authentifizierten Datenstrukturen während einer Ausführung des Sicherheitsspiels $Exp_{Restic}^{IND\$}(\mathcal{A}, 1^{\lambda})$. Damit ist $q(\lambda)$ ebenfalls die Gesamtanzahl aller gezogenen IVs. $\mathcal{A}$ ist polynomiell in λ beschränkt und jedes q_i ist ebenfalls polynomiell beschränkt in λ. Damit gilt, dass auch die Gesamtanzahl $q(\lambda)$ polynomiell in λ beschränkt ist. Damit ist die Wahrscheinlichkeit, dass ein Angreifer zwischen $\mathcal{W}_1$ und $\mathcal{W}_2$ unterscheiden kann, beschränkt durch den Term $\frac{q(\lambda)}{2^{l(\lambda)}}$.

$$|Pr[\mathcal{A} \Rightarrow 1 \mid \mathcal{W}_1] - Pr[\mathcal{A} \Rightarrow 1 \mid \mathcal{W}_2]| \leq \frac{q(\lambda)}{2^{l(\lambda)}} \leq negl(\lambda)$$

Da $q(\lambda)$ polynomiell beschränkt in λ ist, aber $2^{l\lambda}$ superpolynomiell in λ ist, ist der Term $\frac{q(\lambda)}{2^{l(\lambda)}}$ vernachlässigbar in λ.

IND$ Sicherheit des Verschlüsselungsverfahren ($\mathcal{W}_2$ zu $\mathcal{W}_3$):
In $\mathcal{W}_2$ gibt es weiterhin ein Verschlüsselungsverfahren $Enc_{mk_{enc}}^{IV}$, dass mithilfe eines IVs IV und einem geheimen Schlüssel mk_{enc} einen Klartext M zu einem Chiffrat $C := Enc_{mk_{enc}}^{IV}(M)$ verschlüsseln kann. Außerdem gibt es in $\mathcal{W}_2$ ein Authentifizierungsverfahren $Auth_{mk_{auth}}$, das mithilfe eines Schlüssels mk_{auth} zu einem Chiffrat C einen MAC $MAC := Auth_{mk_{auth}}(C)$ generieren kann. In der Welt $\mathcal{W}_3$ wird das Verschlüsselungsverfahren durch eine Funktion $\mathcal{F}_1$ ersetzt, die uniforme Bytestrings $UB_1 \xleftarrow{\$} \mathcal{F}_1(size(C))$ zieht, die die gleiche Länge besitzen wie das eigentliche Chiffrat C des Klartextes M in $\mathcal{W}_2$. Der MAC wird in $\mathcal{W}_3$ über dem uniformen Bytestring UB_1 berechnet, der das Chiffrat ersetzt. Damit besitzt eine verschlüsselte und authentifizierte Datenstruktur in $\mathcal{W}_3$ die Form:

$$data_{3,i} := IV||UB_1||Auth_{mk_{auth}}(UB_1)$$

Die Initialisierungsvektoren werden in beiden Welten uniform aus dem Raum $\mathcal{IV}(\lambda)$ gewählt. Damit unterscheiden sich $\mathcal{W}_2$ und $\mathcal{W}_3$ nur dadurch, dass $\mathcal{W}_2$ echte Chiffrate C für Klartexte K verwendet und $\mathcal{W}_3$ verwendet uniforme Bytestrings mit der gleichen Länge $size$, wie die echten Chiffrate aus $\mathcal{W}_2$.

$$\mathcal{W}_2 : C := Enc_{k_{enc}}^{IV}(K) \qquad \rightarrow \qquad \mathcal{W}_3 : C = UB_1 \xleftarrow{\$} \mathcal{F}_1(size)$$

Ein Angreifer $\mathcal{A}$, der zwischen diesen beiden Welten unterscheiden kann, kann also echte Chiffrate von uniformen Bytestrings unterscheiden. Damit kann ein Angreifer, der zwischen $\mathcal{W}_2$ und $\mathcal{W}_3$ unterscheiden kann, die IND$ Sicherheit des von $\mathcal{W}_2$ verwendeten Verschlüsselungsverfahren brechen.

Wie ein Angreifer $\mathcal{B}_2$ aus einem Angreifer $\mathcal{A}$ konstruiert werden kann, der die IND$ Sicherheit von Restics Verschlüsselungsverfahren bricht ist in Kapitel 5.3.2 dargestellt. Damit ist die Wahrscheinlichkeit, dass $\mathcal{A}$ zwischen $\mathcal{W}_2$ und $\mathcal{W}_3$ unterscheiden kann, durch die Wahrscheinlichkeit $Adv_{Enc}^{IND\$}(\mathcal{B}_\in, \lambda)$ beschränkt, dass der konstruierte Angreifer $\mathcal{B}_\in$ das IND$ Sicherheitsspiel für Restics Verschlüsselungsverfahren gewinnt.

$$|Pr[\mathcal{A} \Rightarrow 1 \mid \mathcal{W}_2] - Pr[\mathcal{A} \Rightarrow 1 \mid \mathcal{W}_3]| \leq Adv_{Enc}^{IND\$}(\mathcal{B}_\in, \lambda)$$

MAC$ Sicherheit des MAC-Verfahrens ($\mathcal{W}_3$ zu $\mathcal{W}_4$):
In $\mathcal{W}_4$ wird die Funktion $Auth_{k_{auth}}$ zum Berechnen von MACs durch eine Funktion $\mathcal{F}_2$ ersetzt, die uniforme Bytestrings $UB_2 \xleftarrow{\$} \mathcal{F}_2(size(M))$ zieht, die die gleiche Länge besitzen wie der eigentliche MAC M für einen Bytestring UB_1. Damit besitzen verschlüsselte und authentifizierter Datenstrukturen in $\mathcal{W}_4$ die Form $IV||UB_1||UB_2$. Somit unterscheiden sich $\mathcal{W}_3$ und $\mathcal{W}_4$ nur dadurch, dass in $\mathcal{W}_3$ echte MACs berechnet werden und in $\mathcal{W}_4$ uniforme Bytestrings gezogen werden, deren Länge der Länge *size* der echten MACs aus $\mathcal{W}_3$ entspricht.

$$\mathcal{W}_3 : M := Auth_{k_{auth}}(UB_1) \qquad \rightarrow \qquad \mathcal{W}_4 : M \xleftarrow{\$} \mathcal{F}_2(size)$$

Ein Angreifer $\mathcal{A}$, der zwischen diesen beiden Welten unterscheiden kann, kann also echte MACs von uniformen Bytestrings unterscheiden. Diese Eigenschaft für MAC-Verfahren wird als Sicherheitsspiel in Kapitel 2.2.3.2 vorgestellt und wird MAC$ Sicherheit genannt. Die Reduktion eines Angreifers $\mathcal{A}$ auf einen Angreifer $\mathcal{B}_3$ für die MAC$ Sicherheit des Authentifizierungsverfahrens ist analog zu der Reduktion für IND$ Sicherheit aus Kapitel 5.3.2. Statt eines Verschlüsselungsorakel besitzt $\mathcal{B}_3$ das Orakel $O_{k_{auth},b}^{Auth}$ des MAC$ Sicherheitsspiels aus Kapitel 2.2.3.2. $\mathcal{B}_3$ besitzt ebenfalls eine Instanz von Restic und ein Repository R, auf dem alle Befehle B des Angreifers ausgeführt werden und Event-Tupel τ_R ergeben. Ebenso konstruiert $\mathcal{B}_3$ aus τ_R Event-Tupel $\tau_{R'}$ bezüglich eines künstlich konstruierten Repositorys R'. $\mathcal{B}_3$ ersetzt die Chiffrate der Send-Events durch uniforme Bytestrings gleicher Länge. Anstelle die Klartexte der Send-Events an das Orakel zu übergeben, gibt $\mathcal{B}_3$ die uniformen Bytestrings an $O_{k_{auth},b}^{Auth}$ und lässt sich einen MAC generieren, der je nach b auch ein uniformer Bytestring sein kann. Damit kann $\mathcal{B}_3$ ebenfalls ein Sicherheitsspiel für $\mathcal{A}$ simulieren, bei dem $\mathcal{W}_3$ die reale Welt und $\mathcal{W}_4$ die ideale Welt ist. Die Laufzeit t_3 von $\mathcal{B}_3$ ist, wie für den konstruierten Angreifer $\mathcal{B}_2$ ebenfalls polynomiell beschränkt in λ, wenn die Laufzeit von $\mathcal{A}$ polynomiell beschränkt ist in λ. Und $\mathcal{B}_3$ gewinnt das MAC$ Sicherheitsspiel, wenn $\mathcal{A}$ zwischen $\mathcal{W}_3$ und $\mathcal{W}_4$ unterscheiden kann. Damit ist die Wahrscheinlichkeit, dass $\mathcal{A}$ zwischen $\mathcal{W}_3$ und $\mathcal{W}_4$ unterscheiden kann, durch die Wahrscheinlichkeit $Adv_{Auth}^{MAC\$}(\mathcal{B}_3, \lambda)$ für das verwendete Authentifizierungsverfahren $Auth$ beschränkt.

$$|Pr[\mathcal{A} \Rightarrow 1 \mid \mathcal{W}_3] - Pr[\mathcal{A} \Rightarrow 1 \mid \mathcal{W}_4]| \leq Adv_{Auth}^{MAC\$}(\mathcal{B}_\ni, \lambda)$$

Von der realen Welt zur idealen Welt:
Per Definition stellt $\mathcal{W}_0$ die reale Welt aus dem Sicherheitsspiel $Exp_{Restic}^{IND\$}$ dar. Die ideale

Welt aus $Exp^{IND\$}_{Restic}$ unterscheidet sich zu $\mathcal{W}_0$ nur dadurch, dass der Inhalt alle $data_{t,i}$ aller erzeugten Event-Tupel τ_t durch uniforme Bytestrings derselben Länge ersetzt wurde. In $\mathcal{W}_4$ besitzen alle $data_{t,i}$ die kein Pack repräsentieren die Form $IV||UB_1||UB_2$. Packs besitzen die Form $IV_1||UB_{1,1}||UB_{2,1}||\ldots||IV_m||UB_{1,m}||UB_{2,m}$. Die IVs werden uniform gewählt und damit besteht jedes $data_{t,i}$ in $\mathcal{W}_4$ aus einer Konkatenation von uniformen Bytestrings. Die Konkatenation von uniformen Bytestrings ist wieder ein uniformer Bytestring nach Lemma 1. Somit gibt es keinen Unterschied, zwischen der idealen Welt aus $Exp^{IND\$}_{Restic}$ und der Welt $\mathcal{W}_4$. Damit ist die Reduktion vollständig. Für die Wahrscheinlichkeit, dass ein Angreifer $\mathcal{A}$ das Sicherheitsspiel $Exp^{IND\$}_{Restic}(\mathcal{A}, 1^\lambda)$ gewinnt, gilt die Gesamtabschätzung 5.2.

$$Adv^{IND\$}_{Restic}(\mathcal{A}, \lambda) \leq \frac{q(\lambda)}{2^{l(\lambda)}} + Adv^{PRF}_{KDF}(\mathcal{B}_1, \lambda) + Adv^{IND\$}_{Enc}(\mathcal{B}_2, \lambda) + Adv^{MAC\$}_{Auth}(\mathcal{B}_3, \lambda) \tag{5.2}$$

Wobei $q(\lambda)$ die Gesamtanzahl aller durch Orakelanfragen von $\mathcal{A}$ gezogenen IVs ist und $l(\lambda)$ die Länge eines IVs in Bits angibt. Außerdem ist wie oben gezeigt der Term $\frac{q(\lambda)}{2^{l(\lambda)}}$ vernachlässigbar in λ.

Fazit der allgemeinen Reduktion:
Da die endliche Summe vernachlässigbarer Funktionen ebenfalls vernachlässigbar ist, folgt folgende Aussage aus Abschätzung 5.2. Für einen PPT-Angreifer $\mathcal{A}$ gilt $Adv^{IND\$}_{Restic}(\mathcal{A}, \lambda) \leq negl(\lambda)$, sofern $Adv^{PRF}_{KDF}(\mathcal{B}_1, \lambda)$, $Adv^{IND\$}_{Enc}(\mathcal{B}_\in, \lambda)$ und $Adv^{MAC\$}_{Auth}(\mathcal{B}_\ni, \lambda)$ vernachlässigbar in λ sind.

Restic ist damit IND$-sicher, wenn die verwendete KDF PRF-sicher ist, das verwendete Verschlüsselungsverfahren IND$-sicher ist und das verwendete Authentifizierungsverfahren MAC$-sicher ist.

Laufzeitbeschränkung der Reduktionsangreifer:
Die Reduktionsangreifer $\mathcal{B}_1$, $\mathcal{B}_2$ und $\mathcal{B}_3$ führen den Angreifer $\mathcal{A}$ als Subroutine aus und erzeugen dabei nur einen polynomiellen Overhead. Dieser Overhead ist abhängig von λ. Da $\mathcal{A}$ polynomiell beschränkt in λ ist, sind auch die Reduktionsangreifer $\mathcal{B}_i$ polynomiell beschränkt in λ. Damit sind alle Reduktionsangreifer $\mathcal{B}_i$ effiziente Angreifer gegen die jeweiligen Primitive.

5.3.3.2. Betrachtung der Gesamtabschätzung bezüglich Restics Primitiven

In diesem Kapitel werden die abstrakten Primitive der allgemeinen Reduktion durch die konkreten von Restic verwendeten Primitive ersetzt. Damit wird die allgemeine Sicherheitsaussage auf Sicherheitseigenschaften der konkreten Primitive scrypt, AES256-CTR und Poly1305-AES128 reduziert. Die konkreten Schlüssellängen und Parametrisierungen der

konkreten Primitive sind festgelegt. Daher wird der Sicherheitsparameter λ der asymptotischen Sicherheit durch eine konkrete Parametrisierungen der einzelnen Primitive ersetzt. Dadurch wird die asymptotische Vernachlässigbarkeitsaussage in λ durch konkrete Sicherheitsschranken ersetzt, die von der Laufzeit t des Angreifers $\mathcal{A}$, der Anzahl verschlüsselter Datenblöcke und der Anzahl der verschlüsselten und authentifizierten Datenstrukturen abhängt.
Beim Einsetzen der Primitive werden die abstrakten Schlüsselräume durch folgende konkrete Räume ersetzt:

- $\mathcal{K}_{enc}(\lambda)$ wird zu $\{0,1\}^{256}$ für den Schlüsselraum von AES256-CTR.
- $\mathcal{K}_{auth}(\lambda)$ wird zu zwei einzelnen Schlüsselräumen für Poly1305-AES128. Ein Schlüsselraum $\{0,1\}^{128}$ ist für den Schlüssel von AES128 und der andere Schlüsselraum ist für den 128 Bit langen Schlüssel von Poly1305. Wichtig hierbei zu erwähnen ist, dass der Schlüsselraum für Poly1305 wie in Kapitel 2.2.3 erwähnt nur eine Größe von 2^{106} besitzt.
- $\{0,1\}^{l(\lambda)}$ wird zu $\{0,1\}^{128} \setminus 0^{128}$ für den IV-Raum von AES256-CTR und den Nonce-Raum für Poly1305-AES128.

Die Sicherheitsaussage 5.3 gilt für alle Angreifer $\mathcal{A}$, die die folgenden Voraussetzung erfüllen:

- Die Gesamtanzahl q_{enc} der verschlüsselten Datenblöcke überschreitet die Schranke q^max_{enc} nicht.
- Die Gesamtanzahl q_{auth} generierten MACs überschreitet die Schranke q^max_{auth} nicht.
- Die Laufzeit t des Angreifers $\mathcal{A}$ liegt unterhalb der Brute-Force-Grenze von 2^{106}. Die Brute-Force-Grenze wird durch das schwächste verwendete Primitiv bestimmt. Ein Angreifer mit $t \geq 2^{106}$ könnte durch systematisches Durchprobieren aller IVs die Sicherheit trivial brechen. Solche Angreifer liegen außerhalb des betrachteten Bedrohungsmodells dieser Masterarbeit. Diese Laufzeiteinschränkung muss ebenfalls für alle aus $\mathcal{A}$ reduzierten Angreifer $\mathcal{B}_i$ gelten. Daher wird vorausgesetzt, dass gilt $t + max_{i=1,...}(c_i) \cdot max(q^{max}_{enc}, q^{max}_{auth}) < 2^{106}$, wobei c_i der Overheadfaktor für den Reduktionsangreifer B_i ist.

Abstrakte KDF $\rightsquigarrow$ scrypt:
Die abstrakte als PRF-sicher angenommene KDF aus Kapitel 5.2.2 wird in Restic durch das Primitiv scrypt ersetzt. Der abstrakte Schlüsselraum der $KDF_{k_{kdf}}$ für den Userkey wird konkret zu einem Schlüsselraum der Größe $2^{256} \cdot 2^{106} \cdot 2^{128} = 2^{490}$. Restic verwendet scrypt genau einmal pro Erstellung einer neuen Key-Datei. Da durch die drei betrachteten Restic-Befehle keine weitere Key-Datei, abseits der initialen Key-Datei, erstellt wird, wird scrypt genau ein mal verwendet. Damit wird der abstrakte Vorteil der abstrakten KDF zu folgendem konkreten Vorteil für scrypt, wobei $q_{kdf} = 1$ die Anzahl der scrypt Aufrufe ist:

$$Adv^{PRF}_{KDF}(\mathcal{B}_1, \lambda) \rightsquigarrow Adv^{PRF}_{scrypt}(\mathcal{B}_1, t_1, q_{kdf})$$

Für die Laufzeit gilt $t_1 \leq t + c_1$, wobei c_1 ein konstanter Overhead ist, da gilt $q_{kdf} = 1$. Scrypt ist ein speicherhartes Verfahren, das allgemein als sicher gegenüber Wörterbuchangriffen gilt. Ein konkreter Sicherheitsbeweis für die PRF-Eigenschaft konnte nicht gefunden werden. Dadurch, dass scrypt speicherhart ist, ist $Adv^{PRF}_{scrypt}(\mathcal{B}_1, t_1, 1)$ vernachlässigbar für einen Angreifer mit beschränkten Speicher- und Zeitressourcen. Die konkrete Schranke dieses Vorteils hängt dabei von den scrypt Parametern (n, r, p) ab, die Restic in jeder Key-Datei speichert. In Restic sind diese Parameter $(n = 2^{16}, r = 8, p = 1)$.

Abstraktes Verschlüsselungsverfahren $\rightsquigarrow AES256 - CTR$:
Das abstrakte IND$-sichere Verschlüsselungsverfahren wird durch Restics AES256-CTR ersetzt. Der abstrakte Schlüsselraum $\mathcal{K}_{enc}(\lambda)$ wird konkret zu $\{0,1\}^{256}$ und der abstrakte IV-Raum $\mathcal{IV}(\lambda) \setminus \{0^{l(\lambda)}\}$ wird zu $\{0,1\}^{128} \setminus 0^{128}$. Die in Kapitel 4.3.2 hergeleitete obere Schranke von $2^{47,7}$ gibt die maximale Anzahl verschlüsselter Datenblöcke an, für die IND$ Sicherheit von Restics AES256-CTR gilt. Damit gilt nach Kapitel 4.3.2 für $q_{enc} \leq q^{max}_{enc} := 2^{47,7}$ und $\epsilon_{IND\$} := Adv^{PRP}_{AES256}(\mathcal{B}_4, t_4, q_{enc}) + 2^{-31}$:

$$Adv^{IND\$}_{Enc}(\mathcal{B}_2, \lambda) \rightsquigarrow Adv^{IND\$}_{CTR[AES256]}(\mathcal{B}_2, t_2, q_{enc}) \leq \epsilon_{IND\$}$$

Da für jede verschlüsselte Datenstruktur in Restic ein IV gewählt, ist die Anzahl q_{enc} verschlüsselter Datenblöcke eine obere Schranke für die Anzahl gezogener IVs. Damit gilt außerdem für die Konkretisierung des IV-Raums:

$$\frac{q(\lambda)}{2^{l(\lambda)}} \rightsquigarrow \frac{q_{enc}}{2^{128}} \leq 2^{-80}$$

Abstraktes Authentifizierungsverfahren $\rightsquigarrow Poly1305 - AES128$:
Das abstrakte MAC$-sichere Authentifizierungsverfahren wird durch Restics Poly1305-AES128 ersetzt. Der abstrakte Schlüsselraum $\mathcal{K}_{auth}(\lambda)$ wird konkret durch die zwei Schlüsselräume der Größe 2^{106} für k_{Poly} und 2^{128} für k_{AES128} ersetzt.

$$Adv^{MAC\$}_{Auth}(\mathcal{B}_3, \lambda) \rightsquigarrow Adv^{MAC\$}_{Poly[AES128]}(\mathcal{B}_3, t_3, q_{enc}, q_{auth}) \leq \epsilon_{MAC\$}$$

$\epsilon_{MAC\$}$ ist dabei folgender Term aus Abschätzung 4.14, wobei L die Menge aller Datenblöcke pro Nachricht ist:

$$Adv^{PRF}_{AES128}(\mathcal{B}_5, t_5, q_{auth}) + \frac{m^2}{2^{129}} + m \cdot \frac{8 \cdot L}{2^{106}}$$

Es kann ausgenutzt werden, dass genau für jede verschlüsselte Nachricht auch ein MAC berechnet wird und damit gilt $m \cdot L = q_{enc}$.
In Kapitel 4.4.2 wurde eine obere Schranke von 2^{48} für die maximale Anzahl berechneter MACs berechnet, unter der die EUF-CMA Sicherheit von Restics Poly1305-AES128 gilt. Da bereits durch die IND$ Sicherheit von AES256-CTR $q^{max}_{enc} = 2^{47,7}$ festgelegt wurde, kann q_{auth} nicht größer werden als $2^{47,7}$, da jeder verschlüsselte Nachrichtenblock zu maximal einer Nachricht gehörten kann, für die ein MAC berechnet wird. Denn damit gilt immer

$q_{auth} \leq q_{enc} \leq q_{enc}^{max}$. Damit gilt nach Abschätzung 4.14 für $q_{enc} \leq q_{auth}^{max}$ und $q_{auth} \leq q_{auth}^{max} := 2^{47,7}$ Schranke $\epsilon_{MAC\$} \leq Adv_{AES128}^{PRF}(\mathcal{B}_5, t_5, q_{auth}) + 2^{-33}$

Gesamtabschätzung:
Durch das Einsetzen der konkreten Sicherheitsabschätzung der konkreten Primitive in die allgemeine Reduktion aus Gleichung 5.2 ergibt sich die konkrete Gesamtabschätzung 5.3 für die Erfolgswahrscheinlichkeit eines Angreifers $\mathcal{A}$ gegen die IND$ Sicherheit von Restic. Dabei muss der Angreife $\mathcal{A}$ allerdings die folgenden Beschränkungen erfüllen:

- Die Gesamtanzahl q_{enc} der verschlüsselten Datenblöcke überschreitet die Schranke q^max_{enc} nicht.
- Die Gesamtanzahl q_{auth} generierten MACs überschreitet die Schranke q^max_{auth} nicht.
- Für die Laufzeit t von $\mathcal{A}$ gilt $t + max_{i=1,\dots}(c_i) \cdot max(q_{enc}^{max}, q_{auth}^{max}) < 2^{106}$, wobei c_i der Overheadfaktor für den Reduktionsangreifer B_i ist.

$$\begin{aligned} Adv_{Restic}^{IND\$}(\mathcal{A}, t, q_{enc}, q_{auth}) \leq\ & Adv_{scrypt}^{PRF}(\mathcal{B}_1, t_1, 1) \\ & + Adv_{AES256}^{PRP}(\mathcal{B}_4, t_4, q_{enc}) + 2^{-31} + 2^{-80} \\ & + Adv_{AES128}^{PRF}(\mathcal{B}_5, t_5, q_{auth}) + 2^{-33} \end{aligned} \tag{5.3}$$

Die Gesamtschranke 5.3 ist eine Summe aus sechs Termen, die jeweils die Sicherheit eines einzelnen Primitivs beschreiben oder eine statische Schranke sind. Alle Terme sind für die betrachteten Parameterbereiche vernachlässigbar klein. Damit gilt, dass auch $Adv_{Restic}^{IND\$}(\mathcal{A}, t, q_{enc}, q_{auth})$ vernachlässigbar ist.

Theorem 3 (IND$ Sicherheit von Restic) *Sei $\mathcal{A}$ ein Angreifer gegen die IND$ Sicherheit von Restic mit Laufzeit t mit $t + max_{i=1,\dots}(c_i) \cdot max(q_{enc}^{max}, q_{auth}^{max}) < 2^{106}$, $q_{enc} \leq q_{enc}^{max}$ und $q_{auth} \leq q_{auth}^{max}$. Es sei die Menge $\{c_1, \dots, c_5\}$ der Menge aller Overheadfaktoren für die aus $\mathcal{A}$ konstruierten Reduktionsangreifer B_i.*
Unter den Annahmen:

- *scrypt ist PRF-sicher*
- *AES256 ist PRP-sicher*
- *AES128 ist PRF-sicher*

gilt Restic ist IND$-sicher. Konkreter gilt die Abschätzung 5.3.

5.3.3.3. Einordnung

Das Theorem 3 zeigt, dass die IND$ Sicherheit von Restic vollständig auf die Sicherheit der verwendeten Primitive zurückgeführt werden kann. Die Sicherheit wird dabei durch das schwächste Primitiv bestimmt. Im aktuellen Fall von Restic ist das Poly1305-AES128 mit einem Schlüsselraum der Größe 2^{106} für einen seiner Schlüssel, sowie die statische Schranke für die IV-Kollisionen und die MAC$ Eigenschaft von Poly1305. Solange allerdings die Schranken $q_{enc}^{max} = 2^{47,7}$ und $q_{auth}^{max} = 2^{47,7}$ eingehalten werden, ist die Gesamtschranke 5.3 für alle relevanten Angreifer in der Praxis vernachlässigbar klein.

Nachwort:
Beim Einsetzen der konkreten Sicherheitsschranken der Primitive ist aufgefallen, dass Poly1305 mit seinem Schlüsselraum der Größe 2^{106} der limitierende Faktor ist. Die Verwendung des Authentifizierungsverfahren für den Reduktionsbeweis der Vertraulichkeit geht nur darauf zurück, dass sich dazu entschieden wurde die Leakage-Funktion so exakt wie möglich zu modellieren. Es wird von der Leakage-Funktion im Allgemeinen nicht vorausgesetzt, dass die Grenze der Konkatenierten Blobs eines Packs Teil der Leakage ist. Es wurde zwar gezeigt, dass mit einer polynomiell beschränkten Anzahl weiterer Restic-Befehle diese Leakage aus der Leakage-Funktion abgeleitet werden kann, dennoch Zählen die Grenzen erstmal nicht zur fest definierten Leakage eines Event-Tupels. Wie in der provozierten Leakage betrachtet, kann man durch mehrere gezielte *backup* Befehle, die Grenze der Tree-Blobs in den Packs herausfinden und durch *restore* Befehle die Grenze der Data-Blobs in den Packs. Wenn man großzügig ist, könnte man die Grenzen aller Blobs in den Packs einfach zur Leakage-Funktion dazu definieren.
Unter dieser Annahme kann man die ideale Welt des vorgestellten Sicherheitsspiel $Exp_{Restic}^{IND\$}$ auf $\mathcal{W}_3$ statt $\mathcal{W}_4$ setzen, da es nun durch die Leakage-Funktion möglich ist für jeden Blob eines Packs einzeln einen uniformen Bytestring zu ziehen und damit auch in der idealen Welt einen echten MAC für jeden Blob zu berechnen. Somit würde die MAC$ Sicherheitsschranke nicht mehr in der Reduktion auftauchen. Diese Annahme ermöglicht auch eine breitere Wahl für Authentifizierungsprimitive, da diese nun nicht mehr MAC$-sicher sein müssen, damit die Vertraulichkeit von Restic gewährleistet ist.
Alle abstrakten Modellierungen der Restic-Befehle setzen ein Repository mit einer Key-Datenstruktur voraus. Somit wurden der Reduktionsbeweis und die konkreten Sicherheitsschranken nur für einen Aufruf der KDF betrachtet. Es sei dazu gesagt, dass die IND$ Sicherheit für Restic nicht bricht, wenn man weitere Key-Datenstrukturen zum Repository hinzufügt, solange die Anzahl in einem vertretbaren Rahmen bleibt. In der Realität werden nicht hunderte von Key-Datenstrukturen für ein Repository erstellt.

5.4. Integrität

Dieses Kapitel analysiert die Integrität von Restic bezüglich der drei betrachteten Restic-Befehle *backup*, *restore* und *prune*. Dazu wird ein EUF-CMA-artiges Sicherheitsspiel $Exp_{Restic}^{EUF-CMA}$ für Restic definiert. Auch für dieses Kapitel werden die Aussagen und Konstrukte aus Kapitel

5.2 vorausgesetzt.
Es werden triviale Angriffe für dieses Sicherheitsspiel betrachtet. Daraufhin wird $Exp_{Restic}^{EUF-CMA}$ zu $Exp_{Restic}^{EUF-CMA-NR}$ erweitert, wodurch es keine trivialen Angriffe mehr für dieses Sicherheitsspiel gibt.
Es wird ein allgemeiner Reduktionsbeweis für $Exp_{Restic}^{EUF-CMA-NR}$ durchgeführt, in dem auf die EUF-CMA Sicherheit eines abstrakten Authentifizierungsverfahrens von Restic reduziert wird. Zum Schluss werden wie bei der Vertraulichkeit konkrete Sicherheitsprimitive mit konkreten Sicherheitsschranken in die Gesamtabschätzung der EUF-CMA-NR Sicherheit von Restic eingesetzt.

5.4.1. Sicherheitsspiel

Dieses Kapitel stellt ein EUF-CMA-artiges Sicherheitsspiel für die Integrität von Restic vor. Dabei werden Restics *backup* Befehl, *restore* Befehl und *prune* Befehl betrachtet. Der Ablauf des Sicherheitsspiels für einen beliebigen Restic-Befehl wird beschrieben. Wie für das Sicherheitsspiel für Restics IND$ Sicherheit, wird auch für dieses Sicherheitsspiel die pbKDF von Restic durch eine abstrakte PRF-sichere KDF ersetzt (siehe Kapitel 5.2.2).
Als Erstes werden jedoch die zwei Teilnehmer und das Orakel des Sicherheitsspiels vorgestellt.

5.4.1.1. Teilnehmer

Bei dem vorgestellten Sicherheitsspiel gibt es zwei Teilnehmer.
Ein Teilnehmer ist der Challenger $\mathcal{C}(1^\lambda)$, der einen Angreifer herausfordert ein modifiziertes Repository und einen Restic-Befehl zu finden, der vollständig auf dem modifizierten Repository ausgeführt werden kann. Der zweite Teilnehmer ist der Angreifer $\mathcal{A}^{O_{uk}}(1^\lambda, mk_{enc}, c_{mk}, m_{mk}) \rightarrow (B', type(B'), R_{mod}, r')$, der versucht das Sicherheitsspiel zu gewinnen, indem er ein Restic-Befehl B und ein modifiziertes Repository R_{mod} findet, sodass B fehlerfrei auf R_{mod} ausgeführt werden kann. Dabei hat $\mathcal{A}$ Zugriff auf ein Orakel O_{uk}, das einen vom Angreifer gewählten Restic-Befehl B mit dem Userkey uk ausführen kann und dem Angreifer das Event-Tupel für B zurückgibt. O_{uk} hat dafür Zugriff auf ein Repository R mit dem Masterkey mk. Außerdem erhält der Angreifer das Chiffrat $c_{mk} := Enc_{uk_{enc}}(mk)$ des Masterkeys von R und den MAC $m_{mk} := Auth_{uk_{auth}}(c_{mk})$ des verschlüsselten Masterkeys als explizite Eingabe. Dies ist damit begründet, dass $\mathcal{A}$ vollen Lesezugriff auf das Repository besitzt und der Masterkey, der einzige Klartext ist, der mit uk verschlüsselt wird. O_{uk} stellt sowohl S_{Restic} und S_{Backup} gleichzeitig dar und O_{uk} übermittelt nur noch das Event-Tupel τ an $\mathcal{A}$. Dadurch, dass $mk \xleftarrow{\$} \mathcal{K}(\lambda)$ zufällig gleichverteilt gewählt wird, liefert c_{mk} keine Informationen über mk.
Wie auch für die Vertraulichkeit von Restic, wird auch für die Integrität von Restic eine Aufteilung in S_{Restic} und S_{Backup} betrachtet (siehe Kapitel 2.3.1.5). Der Angreifer $\mathcal{A}$ repräsentieren einen realen Angreifer mit Zugriff auf S_{Backup}, der in der Lage ist Daten auf dem Backupsystem zu lesen und zu manipulieren. Damit kann ein solcher realer Angreifer alle ausgetauschten Nachrichten zwischen S_{Restic} und S_{Backup} sehen und manipulieren. In dem vorgestellten Sicherheitsspiel gibt es jedoch keinen Nachrichtenaustausch zwischen zwei

Systemen. Stattdessen wird $\mathcal{A}$ mitgeteilt, welche Nachrichten zu welchen Zeitpunkten zwischen S_{Restic} und S_{Backup} bei der Ausführung eines Restic-Befehls ausgetauscht werden würden. Dies geschieht über das Event-Tupel τ eines jeden Restic-Befehls, das äquivalent zur Beobachtung eines Restic-Befehls auf S_{Backup} ist (siehe Theorem 1). Wie in Kapitel 3.2.5.5 beschrieben, ist die Auswirkung jedes Events e_i auf ein Repository eindeutig durch den Typ op_i definiert. Dieses Sicherheitsspiel stellt ein Szenario dar, in dem ein realer Angreifer zunächst ein Repository beobachtet, auf dem Restic-Befehle ausgeführt werden. Nach einem gewissen Zeitraum führt dieser Angreifer eine Manipulation des Repositorys durch, die nicht von Restic bemerkt werden sollte. Dazu erhält $\mathcal{A}$ nach jeder vollständigen Ausführung eines Restic-Befehls das zugehörige Event-Tupel und den Repository-Zustand nach der Ausführung dieses Restic-Befehls. $\mathcal{A}$ erhält außerdem zu Beginn des Sicherheitsspiels den initialen Repository-Zustand des Repositorys, auf dem die Restic-Befehle ausgeführt werden.

5.4.1.2. Prämisse

Es sei daran erinnert, dass die von Restic verwendete pbKDF für das Sicherheitsspiel durch eine abstrakte PRF-sichere KDF ersetzt wurde, um einen Benutzer zu simulieren, der ein Benutzerpasswort mit entsprechend hoher Entropie wählt. Das heißt es wird kein Passwort von einem Benutzer gewählt, sondern zu Beginn vom Challenger ein uniformer Schlüssel für die abstrakte KDF gezogen.
Damit $\mathcal{A}$ volle Kontrolle über das Repository mit Ausnahme der Integrität besitzt, kennt $\mathcal{A}$ den Schlüssel mk_{enc} von Restics Verschlüsselungsverfahren, der Teil des Masterkeys mk des Repositorys R ist.

5.4.1.3. Initialisierung

Das vorgestellte Sicherheitsspiel betrachtet ein zu Beginn leeres Repository R. Dazu wird durch den Challenger C durch die Ausführung des *init* Befehls von Restic ein neues Repository R erstellt. Der *init* Befehl wählt einen Masterkey mk bestehend aus zwei uniformen Schlüsseln $mk_{enc} \xleftarrow{\$} \mathcal{K}_{enc}(\lambda)$ und $mk_{auth} \xleftarrow{\$} \mathcal{K}_{auth}(\lambda)$ und $mk := mk_{enc}||mk_{auth}$, sowie einen Salt $s \xleftarrow{\$} \mathcal{S}(\lambda)$. mk_{enc} wird für Restics Verschlüsselungsverfahren benutzt und wird aus dem Schlüsselraum für das Verschlüsselungsverfahren mit $\mathcal{K}_{enc}(\lambda)$ uniform gezogen. mk_{auth} wird für Restics Authentifizierungsverfahren benutzt und wird aus dem Schlüsselraum für das Authentifizierungsverfahren mit $\mathcal{K}_{auth}(\lambda)$ uniform gezogen. Außerdem erzeugt der *init* Befehl eine Config-Datei C für R. Der *init* Befehl würde mit dem Passwort eines Benutzers und der pbKDF ein Userkey ableiten, mit dem der Masterkey von R verschlüsselt wird. Die pbKDF wurde für dieses Sicherheitsspiel jedoch durch ein abstrakte KDF ersetzt, wodurch nun der Challenger einen uniformen Schlüssel $k_{KDF} \xleftarrow{\$} \mathcal{K}_{kdf}(\lambda)$ aus dem Schlüsselraum für die KDF wählt. Mit der abstrakten KDF leitet C einen Ersatz für den Userkey $uk := KDF_{k_{KDF}}(1^\lambda, s)$ her, für den gilt $uk = uk_{enc}||uk_{auth}$. Nach Definition der abstrakten KDF (siehe Kapitel 5.2.2) sind die beiden Userkeys uk_{enc} und uk_{auth} ebenfalls

uniform. Der Masterkey *mk* wird mit *uk* verschlüsselt und authentifiziert und in den *data* Eintrag der Key-Datenstruktur K geschrieben. Die Key-Datei K wird daraufhin ebenfalls dem Repository R hinzugefügt.
Der Zustand von R nach der Initialisierung durch den Challenger wird initialer Zustand R_0 genannt. C schickt eine Kopie des initialen Repository-Zustands R_0 an $\mathcal{A}$, damit $\mathcal{A}$ den Zustand des Repositorys vor der Ausführung jeglicher Restic-Befehle sieht. Außerdem schickt C den mk_{enc} Teil des Masterkeys von R an $\mathcal{A}$. Damit kann $\mathcal{A}$ alle Daten eines Repository-Zustands von R und die $data_{t,i}$ aller Events $e_{t,i}$ aller Event-Tupel τ_t, die für Restic-Befehle auf R erstellt wurden, entschlüsseln. Das sorgt dafür, dass das Sicherheitsspiel und damit der Beweis für Integrität von Restic völlig unabhängig von dem verwendeten Verschlüsselungsverfahren werden.
Außerdem initialisiert der C das Orakel O_{uk} mit *uk* und R_0 und übergibt dieses als Black-Box ebenfalls an den Angreifer $\mathcal{A}$. Das Repository R stellt dabei bei diesem Sicherheitsspiel eine Ressource dar, auf die sowohl O_{uk}, als auch der Challenger C, die ganze Zeit während des Sicherheitsspiels zugreifen können. Dabei kann der Challenger C kann während des gesamten Sicherheitsspiels nur R legen, während O_{uk} das Repository durch die ehrliche Ausführung von Befehlen B des Angreifers verändern darf.

5.4.1.4. Ablauf des Sicherheitsspiels

Das Sicherheitsspiel $Exp^{EUF-CMA}_{Restic}(\mathcal{A}, 1^\lambda)$ ist ähnlich, wie ein EUF-CMA Sicherheitsspiel für MAC-Verfahren aufgebaut. Es gibt eine Orakel-Phase, in der der Angreifer $\mathcal{A}$ ein Repository und darauf ausgeführte Restic-Befehle beobachten kann, ohne aktiv in die Ausführung einzugreifen. Daraufhin folgt die Challenge-Phase, bei der $\mathcal{A}$ ein Repository beliebig verändert und das veränderte Repository R_{mod} zusammen mit einem Restic-Befehl an den Challenger übergibt. Konnte der Restic-Befehl vollständig ohne Fehler auf dem von $\mathcal{A}$ manipulierten Repository R_{mod} ausgeführt werden, gewinnt $\mathcal{A}$ das Sicherheitsspiel $Exp^{EUF-CMA}_{Restic}(\mathcal{A}, 1^\lambda)$.
Damit das vorgestellte Sicherheitsspiel für Restics Integrität aussagekräftige Ergebnisse liefert, müssen weitere Einschränkungen für das durch $\mathcal{A}$ manipulierte Repository getroffen werden. Ein Angreifer könnte sonst beispielsweise eine Datenstruktur verändern, die von dem Restic-Befehl nicht benutzt wird und würde so das Spiel gewinnen. Welche Einschränkungen genau für die von $\mathcal{A}$ gewählten Repositorys und Restic-Befehle gelten wird im späteren Verlauf dieses Kapitels näher erläutert.
Beim Ablauf des Sicherheitsspiels $Exp^{EUF-CMA}_{Restic}(\mathcal{A}, 1^\lambda)$ übernimmt $\mathcal{A}$ die Rolle des Benutzers aus den abstrakten Modellierungen 3.2, 3.4 und 3.6. Damit schickt $\mathcal{A}$ in der Orakel-Phase selbst das Tupel B, das einen Restic-Befehl repräsentiert, an O_{uk} und somit an S_{Restic}. Das Repository R, auf dem durch O_{uk} Restic-Befehle ausgeführt werden, ist nicht nur ein lineares Repository, auf dem Befehl für Befehl hintereinander ausgeführt wird. $\mathcal{A}$ kann zusammen mit B eine Referenz r an O_{uk} übergeben, die auf einen früheren Repository-Zustand R_r des Repositorys R verweist. Das Repository, das $\mathcal{A}$ referenziert, ist basiert immer auf dem von C zu Beginn des Spiels erstellten Repository R. Jeder Befehl B, den der Angreifer an O_{uk} zur Ausführung übergibt, wird mit einem fortlaufendem Index nummeriert. So ist B_t der t-te Befehl, den $\mathcal{A}$ zum Ausführen an O_{uk} übergibt, wobei gilt $1 \leq t$. Der Repository, das entsteht, nachdem der Restic-Befehl B_t vollständig auf R ausgeführt wurde, wird R_t

genannt. R_0 bezeichnet, das zu Beginn des Sicherheitsspiels von C erstellte Repository R. Der Angreifer übergibt in der Orakel-Phase immer einen Restic-Befehl B und eine Referenz r an O_{uk}. Das Orakel O_{uk} ersetzt R durch den früheren Repository-Zustand R_r und führt auf diesem den Befehl des Angreifers $\mathcal{A}$ aus und gibt $\mathcal{A}$ das zugehörige Event-Tupel τ zurück. Ein Angreifer kann per Definition von R_t für die s-te Anfrage an das Orakel nur Werte für r zwischen 0 und $s-1$ wählen. Für den ersten Befehl des Angreifers gilt damit $r = 0$, da es keinen anderen Repository-Zustand bisher gab.
Über diesen zusätzlichen Parameter r ist der Angreifer in der Lage, jeden beliebigen Repository-Zustand innerhalb desselben Spiels mit demselben Masterkey darzustellen. Sehr wichtig ist, das $\mathcal{A}$ damit nicht aktiv das Repository R verändert und somit nicht schon in der Orakel-Phase gefälschte MACs zu R hinzufügt. $\mathcal{A}$ hat mit r nur die Möglichkeit die "Zeit zurückzudrehen" und ausgeführte Restic-Befehle rückgängig zu machen. Somit wird die Erfolgswahrscheinlichkeit eines Angreifers nicht dadurch beschränkt, dass $\mathcal{A}$ während dem Sicherheitsspiel auf ein lineares Repository beschränkt ist, das mit *prune* Befehlen zurückgesetzt werden müsste bei Bedarf und mit einer erneuten Menge an Befehlen wieder auf einen Zustand gebracht wird, der dem Angreifer einen Vorteil verschafft. Wie gerade angedeutet, ist es $\mathcal{A}$ mit dem *prune* Befehl begrenzt möglich das umzusetzen, was mit der expliziten Übergabe durch eine Referenz r auf einen früheren Repository-Zustand R_r möglich ist.
In der Challenge-Phase wird das Repository auf dem der Befehl B' ausgeführt wird nicht durch eine Referenz r bestimmt, sondern $\mathcal{A}$ bestimmt ein frei gewähltes und beliebig manipuliertes Repository R_{mod}. Trotzdem wird auch in der Challenge-Phase eine Referenz r' von $\mathcal{A}$ zusammen mit dem Befehl B' und R_{mod} übergeben. Diese Referenz wird vom Challenger benutzt, um einen früheren Repository-Zustand R_{basis} zu bestimmen, auf dem R_{mod} basiert. Was es mit R_{basis} auf sich hat wird im späteren Kapitel 5.4.1.5 erklärt. In der Challenge-Phase übergibt $\mathcal{A}$ nun ein Befehlstupel B', das Repository R_{mod} und die Referenz r' an den Challenger C Damit $\mathcal{A}$ das Sicherheitsspiel $Exp^{EUF-CMA}_{Restic}(\mathcal{A}, 1^\lambda)$ nicht trivial gewinnt, unterliegt R_{mod} Einschränkungen, die im späteren Verlauf dieses Kapitels erläutert werden.

1. Wie in Kapitel 5.4.1.3 erklärt, initialisiert C ein Repository R auf S_{Restic}, das für das restliche Sicherheitsspiel für alle weiteren Restic-Befehle verwendet wird. C initialisiert das Orakel $O_{uk} := Init(1^\lambda, uk, R_0)$ mit dem Repository-Zustand R_0 und dem Userkey uk. Außerdem schickt C den initialen Repository-Zustand R_0 an $\mathcal{A}$ und C schickt den Schlüssel mk_{enc} für Restics Verschlüsselungsverfahren, der Teil des Masterkeys von R ist, an $\mathcal{A}$.
2. In der Orakel-Phase darf $\mathcal{A}$ ein Befehlstupel B für einen beliebigen der drei betrachten Restic-Befehle und dessen Parameter wählen. Dieses Befehlstupel B übergibt $\mathcal{A}$ zusammen mit dem Befehlstyp $type(B)$ und einer Referenz r auf einen früheren Repository-Zustand von R an O_{uk}. O_{uk} setzt R auf den Repository Zustand R_r zurück und führt den Restic-Befehl B vom Typ $type(B)$ auf dem Repository R aus. Dabei wird ein Event-Tupel τ_t erzeugt, das den kompletten Nachrichtenaustausch der abstrakten Modellierung des Restic-Befehls B beschreibt. O_{uk} gibt das bestimmte τ_t und den Repository-Zustand R_t nach Ausführung des Befehls B an $\mathcal{A}$ zurück $((\tau_t, R_t) := O_{uk}(B, type(B), r))$. $\mathcal{A}$ darf diesen Ablauf für beliebig viele Befehlstupel wiederholen.

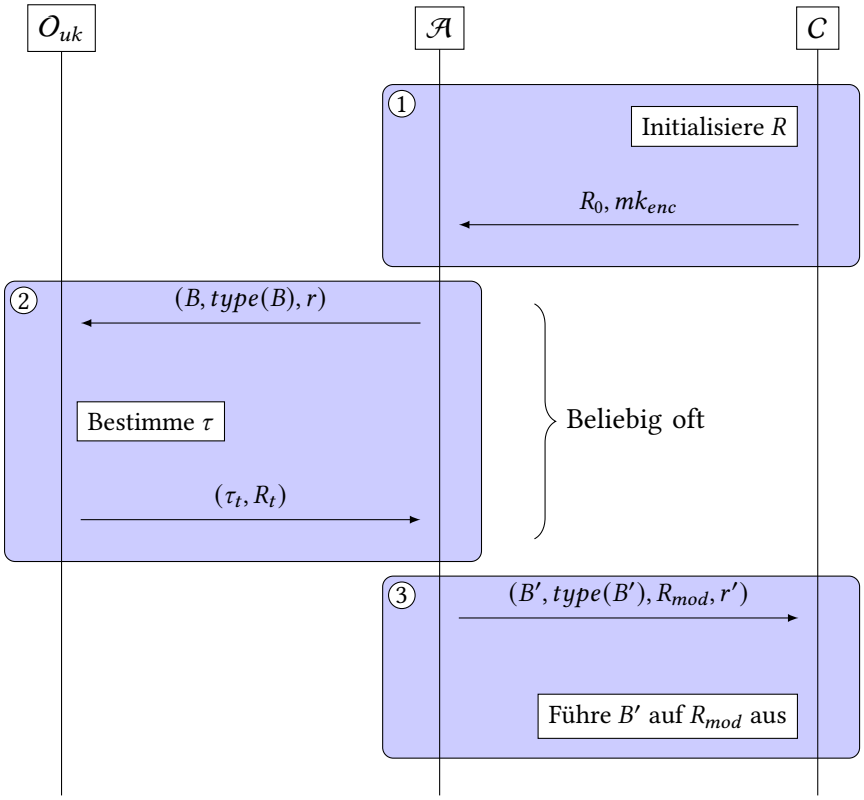

Figure 5.3.: Sicherheitsspiel $Exp^{EUF-CMA}_{Restic}(\mathcal{A}, 1^\lambda)$ für die Integrität von Restic bezüglich eines Angreifers $\mathcal{A}$

3. Nachdem $\mathcal{A}$ beliebig viele Restic-Befehle von O_{uk} hatte ausführen lassen, geht das Sicherheitsspiel in die Challenge-Phase über. $\mathcal{A}$ konstruiert aus einem früheren Repository-Zustand $R_{basis} := R_{r'}$ ein manipuliertes Repository R_{mod}. Dann übergibt $\mathcal{A}$ seine Wahl $(B', type(B'), R_{mod}, r') := \mathcal{A}^{O_{uk}}(1^\lambda, mk_{enc}, c_{mk}, m_{mk})$ an den Challenger C. Der Challenger führt den Befehl B' auf R_{mod} aus. Konnte der Befehl vollständig ausgeführt werden, ohne dass Restic einen Fehler erzeugt hat, gewinnt $\mathcal{A}$ das Spiel. Es gilt $Exp^{EUF-CMA}_{Restic}(\mathcal{A}, 1^\lambda) := 1$ genau dann, wenn B' vollständig auf R_{mod} ausgeführt werden konnte, ohne dass ein Fehler bei der Ausführung aufgetreten ist und $(B', type(B'), R_{mod}, r')$ die nachfolgenden Einschränkungen erfüllt.
   Andernfalls gilt $Exp^{EUF-CMA}_{Restic}(\mathcal{A}, 1^\lambda) := 0$.

5.4.1.5. Einschränkung für das von $\mathcal{A}$ gewählte Tupel $(B', type(B')R_{mod}, r')$

Damit mit dem vorgestellten Sicherheitsspiel $Exp^{EUF-CMA}_{Restic}(\mathcal{A}, 1^\lambda)$ aussagekräftige Ergebnisse für Restics Integrität getroffen werden können, müssen Einschränkungen für die Antwort des Angreifers in der Challenge-Phase getroffen werden. Integrität wird in dieser Masterarbeit als die Fähigkeit beschrieben, zu verhindern, dass eine unbefugte Person Daten verändern oder löschen kann. Im Falle von $Exp^{EUF-CMA}_{Restic}(\mathcal{A}, 1^\lambda)$ ist diese unbefugte Person

der Angreifer $\mathcal{A}$ und die Daten, die dieser nicht verändern oder löschen darf, werden durch das Repository R dargestellt. Der Angreifer, der für dieses Sicherheitsspiel betrachtet wird, besitzt in der Realität vollen Zugriff auf S_{Backup} und kann damit ein Repository beliebig verändern. Da in der Realität S_{Backup} in der Regel nicht Teil von S_{Restic} und damit nicht Teil von Restic ist, kann Restic keine Integritätsgarantie für S_{Backup} aussprechen. Damit kann Restic insbesondere auch nicht die Integrität eines Repositorys auf S_{Backup} garantieren, solange dieses Repository nicht von Restic verwendet wird. Sobald ein Repository von Restic durch einen Restic-Befehl verwendet wird, kann Restic kryptografische Primitive verwenden, um zu erkennen, ob der Teil des Repositorys, den dieser Restic-Befehl verwendet, unbefugt verändert wurde. Unbefugt verändert bedeutet in diesem Fall, dass eine Person oder Angreifer das Repository verändert hat, der keine Kenntnis über die Schlüssel der Primitive besitzt, die zur Garantie der Integrität verwendet werden. In Restic steht vor allem ein Primitiv im Vordergrund, das die Integrität von Restic sicherstellen soll und das Primitiv ist ein Authentifizierungsverfahren, mit dem zu jeder verschlüsselten Datenstruktur ebenfalls ein MAC berechnet wird und zusammen mit dem Chiffrat der Datenstruktur im Repository gespeichert wird. Restic kann also nicht verhindern, dass ein Repository verändert wird, aber Restic kann Veränderungen erkennen, den Benutzer darüber informieren und die weitere Ausführung eines Restic-Befehls verhindern, um Schaden am Resticsystem zu verhindern. Genau auf diesem Versprechen von Restic basiert die Gewinnbedingung eines Angreifers $\mathcal{A}$ für das Sicherheitsspiel $Exp^{EUF-CMA}_{Restic}(\mathcal{A}, 1^{\lambda})$. Kann ein Angreifer $\mathcal{A}$ einen Restic-Befehl B' und ein von $\mathcal{A}$ manipuliertes Repository R_{mod} finden, sodass B' vollständig auf R_{mod} ausgeführt wird, ohne dass Restic die Veränderungen in R_{mod} bemerkt, bricht $\mathcal{A}$ die Integrität von Restic. Die Gewinnbedingung für $\mathcal{A}$ ist jedoch sehr grob formuliert, weswegen weiter Einschränkungen für die Wahl des Tupels $(B', type(B'), R_{mod}, r')$ getroffen werden müssen, um triviale Angriffe zu verhindern. Gibt $\mathcal{A}$ dem Challenger ein B', R_{mod} und r', die nicht alle der nachfolgenden Bedingungen erfüllen, verwirft der Challenger die Wahl von $\mathcal{A}$ und $\mathcal{A}$ verliert das Sicherheitsspiel $Exp^{EUF-CMA}_{Restic}$. Alle geforderten Einschränkung für die Ausgabe eines Angreifers $\mathcal{A}$ in der Challenge-Phase sind am Ende dieses Kapitels durch die Menge $constraints$ zusammengefasst.

R_{mod} muss auf einem vorherigen R_i basieren:
Um in den weiteren Einschränkungen für B' und R_{mod} die Bedingungen sauber definieren zu können, wird eine Basis für das Repository R_{mod} benötigt. Dazu wird gefordert, dass es einen Repository-Zustand R_{basis} aus der Orakel-Phase gibt, der als Basis für R_{mod} dient. Der Repository-Zustand $R_{basis} := R_{r'}$ ist also der Zustand von R nach der Ausführung des r'-ten Befehls in der Orakel-Phase. Mit dem Begriff Basis ist gemeint, dass R_{mod} aus beliebigen Veränderungen von R_{basis} entsteht. Das kann auch dazu führen, dass ein Angreifer jede Datei aus R_{basis} verändert oder ersetzt und damit ein komplett selbsterstelltes Repository für R_{mod} benutzt. R_{basis} wird nur für die Definitionen weitere Einschränkungen benötigt und schränkt den Angreifer selbst nicht ein.
Abgesehen von der Begründung warum R_{basis} benötigt wird, entspricht diese Definition für R_{mod} ebenfalls dem realen Angriffsszenario, in dem ein Angreifer ein bestehendes Repository $R_{r'}$ manipuliert und so die Integrität versucht zu brechen.

Löschen von Datenstrukturen aus dem Repository:
Ein manipulierender Angreifer, der Zugriff auf ein Repository hat, kann natürlich auch Datenstrukturen aus diesem Repository löschen. Je nach Datenstruktur und Restic-Befehl der ausgeführt wird, kann das Fehlen einer Datenstruktur im Repository durch Restic bemerkt werden oder nicht. Zu Beginn der drei betrachteten Restic-Befehle werden beispielsweise durch alle drei Befehle alle Index-Datenstrukturen aus dem Repository angefordert. Restic bemerkt durch eine solche Sammelanforderung nicht, wenn eine Index-Datenstruktur fehlt. Das Fehlen dieser Datenstruktur wird allerhöchstens später bemerkt, wenn die Index-Datenstruktur für die Ausführung des Befehls benötigt wird und nicht vorhanden ist. Genauso könnte ein Angreifer alle Datenstrukturen löschen, die durch die Ausführung eines einzigen *backup* Befehls dem Repository hinzugefügt wurden. Die Ausführung weiterer Restic-Befehl würde dadurch niemals zu Fehlern führen, es sei denn sie verweisen speziell mit ihren Optionswerten, wie *PARENT* oder *SNAPSHOT_ID* auf den gelöschten Snapshot. Der Angreifer stellt nämlich durch das Löschen aller Datenstrukturen, die durch denselben *backup* Befehl zu R hinzugefügt wurden nur den Repository-Zustand vor der Ausführung dieses *backup* Befehls wieder her. Damit befindet sich das Repository wieder in einem früheren durch den Angreifer unmanipulierten Zustand.
Löscht ein Angreifer aber beispielsweise eine Datenstruktur, die Restic explizit über die ID dieser Datenstruktur aus dem Repository anfordert und nicht als Sammelanforderung wie bei den Indices, löst das bei Restic einen Fehler aus.
Das Löschen von Daten aus dem Repository ist weiterhin eine schädliche Aktion eines manipulierenden Angreifers, allerdings nichts was Restic verhindern könnte und in manchen Situationen auch nichts was Restic direkt erkennen würde. Außerdem ist ein Angreifer durch das Löschen in der Lage, Daten aus dem Repository dauerhaft zu entfernen und für Datenverlust zu sorgen. Das ist allerdings nichts, was Restic verhindern könnte und stellt eher das Schutzziel Verfügbarkeit dar, als die Integrität Restics. Des Weiteren ist das Detektieren von fehlenden Dateien keine besondere Eigenschaft und sagt nichts über die kryptografische Stärke von Restics Integrität aus. Daher darf ein Angreifer $\mathcal{A}$ in dem Sicherheitsspiel $Exp^{EUF-CMA}_{Restic}$ zwar weiterhin Datenstrukturen aus R_{basis} löschen, allerdings darf sich R_{mod} nicht nur dadurch von R_{basis} unterscheiden, dass Datenstrukturen aus R_{basis} gelöscht wurden.

Definition 5.4.1 *Es sei R ein Repository und R' ein Repository, das aus R durch das Löschen einer Datenstruktur entsteht.*
Dann wird geschrieben:

$$R \xrightarrow{del} R'$$

Entsteht R' aus R durch das reine Löschen mehrerer Datenstrukturen, wird geschrieben:

$$R \xrightarrow{del*} R'$$

Einschränkungen für B':
Um Einschränkungen für das vom Angreifer gewählte Befehlstupel B' treffen zu können,

wird eine weitere Definition benötigt. $\tau_{B',R_{mod}}$ entspricht dem Event-Tupel, das durch die Ausführung des Befehls B' auf R_{mod} entsteht und $\tau_{B',R_{basis}}$ ist analog das Event-Tupel für B' und R_{basis}. Kommt es bei der Ausführung von B' auf einem der Repositorys zu einem Fehler, endet das jeweilige Event-Tupel direkt, nachdem es bei Restic zu einem Fehler kommt. Das entspricht auch dem realen Verhalten von Restic, da Restic die Ausführung eines Restic-Befehls abbricht, sobald es zu einem Fehler kommt und damit werden keine weiteren Datenstrukturen zwischen S_{Restic} und S_{Backup} ausgetauscht.

Definition 5.4.2 (Fehlerfreie Ausführung) *Sei B' ein Restic-Befehl und R_{mod} ein Repository und $\tau_{B',R_{mod}}$ das Event-Tupel, das durch die Ausführung von B' auf R_{mod} entsteht.*
B' wurde vollständig auf R_{mod} ohne Fehler ausgeführt genau dann, wenn gilt:

$$detect_{err}(\tau_{B',R_{mod}}) := detect_{err}(B', R_{mod}) = 0$$

Kam es zu einem Fehler bei der Ausführung von B' auf R_{mod}, dann gilt $detect_{err}(\tau_{B',R_{mod}}) = 1$.

Damit ein Angreifer $\mathcal{A}$ die Integrität von Restic bricht, muss ein Restic-Befehl auf einem vom Angreifer manipulierten Repository erfolgreich ausgeführt werden können. Damit der Angreifer nicht nur irgendwelche Datenstrukturen verändert, die von einem bestimmten Restic-Befehl nicht verwendet werden, müssen folgende Bedingungen für B' in Kombination mit R_{mod} gelten.
$\tau_{B',R_{mod}}$ muss mindestens ein Event $e_{mod} = (op_{mod}, delay_{mod}, type_{mod}, data_{mod}, blobInfo_{mod})$ enthalten, für das gilt, $op_{mod} := receive$ und $data_{mod}$ ist eine Datenstruktur, die entweder in R_{basis} nicht existiert oder in R_{basis} an einer anderen strukturellen Position wie in R_{mod} existiert. Der Begriff strukturelle Position ist für Data-Blobs und Tree-Blobs ($type_{mod} = BlobChunk$) anders definiert, als für Restics restliche Datenstrukturen. In Restic werden alle Datenstrukturen des Repositorys, die keine Blobs sind, als Datei mit ihrer Restic-ID als Dateiname im Repository gespeichert (siehe Kapitel 2.3.3.2). Dieser Dateiname wird verwendet, falls Restic eine Datenstruktur über ihre ID anfordert. Wird eine Datenstruktur über ihre ID angefordert, schickt S_{Backup} den Inhalt der Datei vom Typ der angeforderten Datenstruktur mit der angeforderten ID als Dateiname zurück.
Die strukturelle Position einer Datenstruktur, die kein Blob ist, ist bestimmt durch ihren Dateinamen und dem Typ dieser Datei im Repository. Der Typ der Datei wird durch das Verzeichnis, in dem sie im Repository gespeichert ist definiert (siehe Kapitel 2.3.3.3). Die strukturelle Position eines Blobs oder eines Blob-Chunks ist bestimmt durch das Pack, in dem diese Blobs vorkommen und deren Konkatenationsreihenfolge. Ein Blob oder eine Menge von Blobs, die eine andere strukturelle Position besitzen, sind Teil einer anderen Pack-Datenstruktur oder kommen in einer unterschiedlichen Reihenfolge in der gleichen Pack-Datenstruktur vor.
Die strukturelle Position einer Datenstruktur eines Events e_i bezüglich dem Repository R ist abstrakt definiert als die Funktion $structure_R(e_i)$.

Einschränkungen für die Veränderung einer Key-Datenstruktur:
Eine Key-Datenstruktur ist im Repository immer unverschlüsselt gespeichert (siehe Kapitel

2.3.4.3). Dementsprechend berechnet Restic für die Key-Datenstruktur auch keinen MAC. Ein Angreifer könnte den *username* Eintrag oder den *hostname* Eintrag oder den *time* Eintrag verändern, ohne dass Restic diese Änderung detektieren kann. Da diese Einträge für die Ausführung der drei betrachteten Restic-Befehle nicht relevant sind, würde ein Angreifer mit dieser Änderung das Sicherheitsspiel $Exp^{EUF-CMA}_{Restic}$ trivial gewinnen. Verändert ein Angreifer die restlichen Einträge der Key-Datenstruktur, führt das zur Berechnung eines anderen Userkeys aus dem gleichen Benutzerpasswort des Challengers. Das würde bei Restic in der Realität zu einem Fehler führen, da mit dem berechneten Userkey nicht mehr der *data* Eintrag der Key-Datenstruktur authentifiziert werden kann.
In der Modellierung des Sicherheitsspiels $Exp^{EUF-CMA}_{Restic}$ wird die KDF gezielt übersprungen und es wird immer mit einem bekannten Userkey gearbeitet. Daher würden für dieses Sicherheitsspiel Restic-Befehle nicht abbrechen, wenn die KDF fehlschlägt, da nicht für jeden Restic-Befehl erneut die KDF verwendet wird, um den Userkey herzuleiten. Um diese Verlierkriterien des Angreifers wieder mit in Betrachtung aufzunehmen und um die trivialen Angriffe auszuschließen, wird die Einschränkung getroffen, dass $\mathcal{A}$ die Key-Datenstruktur von R niemals verändern darf.
Damit gilt für die Key-Datenstruktur $K_{R_{mod}}$ aus R_{mod} und die Key-Datenstruktur $K_{R_{basis}}$ aus R_{basis} folgendes:

$$K_{R_{mod}} \stackrel{!}{=} K_{R_{basis}}$$

Zulässige Ausgabe des Angreifers in der Challenge-Phase:
Dieses Kapitel fasst die Einschränkungen für die zulässigen Antworten des Angreifers in der Challenge-Phase zusammen.

Definition 5.4.3 (constraints) *Alle trivialen Einschränkung für das Ausgabe-Tupel eines Angreifers $\mathcal{A}$ in der Challenge-Phase des Sicherheitsspiels $Exp^{EUF-CMA}_{Restic}(\mathcal{A}, 1^\lambda)$ sind durch die Menge constraints definiert:*
Es sei $(B', type(B'), R_{mod}, r') := \mathcal{A}^{O_{uk}}(1^\lambda, mk_{enc}, c_{mk}, m_{mk})$ die Ausgabe eines Angreifers $\mathcal{A}$.
Dann ist die Enthält die Menge $constraints(B', type(B'), R_{mod}, r')$ folgende Einschränkungen:

- $R_{r'} \overset{del*}{\not\longrightarrow} R_{mod}$
- $K_{R_{mod}}$=$K_{R_{r'}}$
- $\exists e_i \in \tau_{B', R_{mod}}$ *mit* $op_i = receive \wedge data_i \in R_{mod} : data_i \notin R_{r'}$ ***oder*** $structure_{R_{mod}}(e_i) \neq structure_{R_{r'}}(e_i)$

Es gilt für die Funktion $check(B', type(B'), R_{mod}, r') = 1$ genau dann, wenn alle Bedingungen der Menge $constraints(B', type(B'), R_{mod}, r')$ erfüllt sind, ansonsten gilt $check(B', type(B'), R_{mod}, r') = 0$.

Definition 5.4.4 (Gewinnbedingung für $Exp^{EUF-CMA}_{Restic}$) *Für die Ausgabe $(B', type(B'), R_{mod}, r') := \mathcal{A}^{O_{uk}}(1^\lambda, mk_{enc}, c_{mk}, m_{mk})$ eines Angreifers $\mathcal{A}$ gilt:*

$$Exp^{EUF-CMA}_{Restic}(\mathcal{A}, 1^\lambda) := \begin{cases} 1 & check(B', type(B'), R_{mod}, r') = 1 \wedge detect_{err}(\tau_{B', R_{mod}}) = 0 \\ 0 & sonst \end{cases}$$

5.4.1.6. Erfolgswahrscheinlichkeit

Die Wahrscheinlichkeit, dass $\mathcal{A}$ mit der Ausgabe $(B', type(B'), R_{mod}, r')$ das Sicherheitsspiel $Exp^{EUF-CMA}_{Restic}$ gewinnt, ist durch die Formel 5.4 definiert.

$$Adv^{EUF-CMA}_{Restic}(\mathcal{A}, \lambda) := Pr[check(B', type(B'), R_{mod}, r') = 1 \wedge detect_{err}(\tau_{B', R_{mod}}) = 0] \quad (5.4)$$

5.4.2. Replay-Angriffe auf $Exp^{EUF-CMA}_{Restic}$

Mit den Einschränkungen aus der Menge *constraints* für die möglichen Antworten eines Angreifers $\mathcal{A}$ für das Sicherheitsspiel $Exp^{EUF-CMA}_{Restic}$ sind triviale Angriffe ausgeschlossen worden. Es gibt jedoch weiterhin Möglichkeiten für Replay-Angriffe, mit denen ein Angreifer das Sicherheitsspiel $Exp^{EUF-CMA}_{Restic}$ gewinnt. Bei einem Replay-Angriff auf $Exp^{EUF-CMA}_{Restic}$, verwendet der Angreifer eine oder mehrere Datenstrukturen eines oder mehrerer früherer Repository-Zustände R_i in dem Repository R_{mod} wieder, ohne das Restic dies bemerkt. Diese Art von Replay-Angriff funktioniert nur, da Restic im gesamten Sicherheitsspiel den gleichen Masterkey zum Verschlüsseln und Authentifizieren von Datenstrukturen benutzt. Somit kann der Angreifer eine Datenstruktur zu R_{mod} hinzufügen, die zu einem früheren Zeitpunkt mit dem Masterkey des Repositorys authentifiziert wurde, ohne dass der Angreifer diesen Masterkey kennt.
Replay-Angriffe sind zwar für das $Exp^{EUF-CMA}_{Restic}$ ebenfalls triviale Möglichkeiten, das Spiel zu gewinnen, aber Replay-Angriffe sind realitätsnah, da sie kaum Aufwand vonseiten eines Angreifers benötigen. Daher werden Replay-Angriffe zusammen mit ihren Auswirkungen in diesem Kapitel vorgestellt. Replay-Angriffe sind außerdem die einzigen Angriffe, mit denen ein Angreifer das Sicherheitsspiel $Exp^{EUF-CMA}_{Restic}$ gewinnen kann, sofern die verwendeten kryptografischen Primitive sicher sind.
Die Replay-Angriffe werden nach den drei betrachteten Restic-Befehlen gruppiert.

Grundlagen für Replay-Angriffe auf Restic:
Außerdem gibt es für die Key-Datenstruktur und die Config-Datenstruktur keinen Replay-Angriff bezüglich dem Sicherheitsspiel $Exp^{EUF-CMA}_{Restic}$. Warum das so ist und wie genau Restic-IDs sich auf Replay-Angriffe auswirkt, wird in diesem Kapitel beschrieben. Diese Grundlagen sind wichtig, um relevante Replay-Angriffe für die drei betrachteten Restic-Befehle herausarbeiten zu können.
Für die gesamte Lebensdauer eines Repositorys wird in Restic der gleiche Masterkey für dieses Repository verwendet. Damit bleiben auch die Key-Datenstrukturen eines Repositorys, unverändert, sofern keine neuen erstellt werden oder alte gelöscht werden. Im Falle des Sicherheitsspiels $Exp^{EUF-CMA}_{Restic}$ gibt es nur einen Benutzer, der sein Benutzerpasswort nicht ändert, wodurch während des ganzen Sicherheitsspiels dieselbe Key-Datenstruktur im Repository gespeichert ist. Ein Replay-Angriff bezüglich der Key-Datenstruktur ist nicht möglich, da die Key-Datenstruktur im Laufe des Sicherheitsspiels nicht verändert wird und auch keine andere Key-Datenstruktur erstellt wird. Dasselbe gilt ebenfalls für die

Config-Datenstruktur des Repositorys R.
Eine weitere Einschränkung bilden die Restic-IDs. Jede Datenstruktur besitzt eine Restic-ID, die in Restic wie in Tabelle 2.5 beschrieben berechnet wird. Für alle Datenstrukturen außer Blobs verwendet Restic die ID der Datenstruktur, um diese aus dem Repository anzufordern. Da im Repository alle Datenstrukturen als Dateien gespeichert sind und die ID der Datenstruktur der jeweilige Dateiname ist, wird bei einer solchen Anforderung der Inhalt der Datei, deren Name der angeforderten ID entspricht, an Restic zurückgeschickt. Für das Anfragen von einem Blob oder einem Blob-Chunk, verwendet Restic die ID des Packs, das diese Blobs enthält und die Startposition und Endposition dieser Blobs im Pack. Als Blob-Chunks werden immer nur zusammenhängende Bytestrings, die aus mehreren konkatenierten Blobs bestehen aus einem Pack angefordert. Daher reicht immer eine Startposition und eine Endposition für einen Blob-Chunk.
Die Informationen über die Blobs bezieht Restic immer über die Index-Datenstrukturen. Dazu muss Restic allerdings ebenfalls die IDs der angefragten Blobs kennen, um die Informationen der Blobs aus den Index-Datenstrukturen zu erhalten. Immer, wenn Restic eine Datenstruktur aus dem Repository anfordert, berechnet Restic nach Erhalt dieser Datenstrukturen die deterministische ID der erhaltenen Datenstruktur und vergleicht die berechnete ID mit der ID der Datenstruktur, die Restic aus dem Repository angefordert hatte. Restic beendet einen Restic-Befehl immer mit einem Fehler, sobald die beiden IDs nicht übereinstimmen. Damit ist es bei einem Replay-Angriff nicht möglich eine Datenstruktur durch irgendeine Datenstruktur aus einem früheren Repository-Zustand zu ersetzen. Für einen erfolgreichen Replay-Angriff müssen die IDs der angeforderten Datenstrukturen mit den IDs der aus R_{mod} erhaltenen Datenstrukturen übereinstimmen. Wenn Restic mehrere Blobs aus einem Pack gleichzeitig anfordern möchte, geschieht das durch einen Blob-Chunk (siehe Funktion 29). Dabei kann es aus Effizienzgründen vorkommen, dass der angeforderte Blob-Chunk Blobs enthält, die Restic gar nicht benötigt. Restic verifiziert nur die IDs der Blobs aus einem angeforderten Blob-Chunk, die Restic wirklich verwendet.
Ein Angreifer kann das Sicherheitsspiel $Exp_{Restic}^{EUF-CMA}$ also nicht einfach gewinnen, wenn der Dateiname einer angeforderten Datenstruktur nicht der ID dieser Datenstruktur entspricht, selbst wenn die Datenstruktur korrekt authentifiziert wurde.

5.4.2.1. Allgemeine triviale Replay-Angriffe

Es gibt triviale Replay-Angriffe, mit denen der Angreifer für alle drei Arten von Restic-Befehlen für B' das Sicherheitsspiel $Exp_{Restic}^{EUF-CMA}$ gewinnt. Triviale Replay-Angriffe bezeichnen Replay-Angriffe, die Datenstrukturen zu R_{mod} hinzufügen, die durch kategorische Anforderungen von Restic angefordert werden, aber keinen nennenswerten Einfluss auf die Befehlsausführung haben. Diese Angriffe sind das Pendant zum Löschen einer Datenstruktur, die kategorisch durch eine Sammelanforderung angefordert wurde. Diese Replay-Angriffe lassen sich aufgrund der Protokoll-Definition von Restic nur sehr kompliziert ausschließen. Daher werden diese Replay-Angriffe als erstes betrachtet, um sie nicht für jeden Restic-Befehl einzeln zu betrachten.
Es gibt für die abstrakte Modellierung jedes Restic-Befehls mindestens eine Situation, in der alle Datenstrukturen von mindestens einem Typ kategorisch angefordert werden.

Bei einem *backup* Befehl oder *restore* Befehl werden alle Index-Datenstrukturen, alle Lock-Datenstrukturen und bei der Bestimmung eines Parent-Snapshots alle Snapshot-Datenstrukturen angefordert. Bei einem *prune* Befehl werden alle Index-Datenstrukturen angefordert. Bei allen diesen kategorischen Anforderungen ist es einem Angreifer möglich eine Datenstruktur des gleichen Typs aus einem früheren Repository-Zustand zu R_{mod} hinzuzufügen und das Sicherheitsspiel trivial zu gewinnen. Dadurch, dass alle diese Anforderungen kategorisch stattfinden, wird zwangsläufig auch die vom Angreifer hinzugefügte Datenstruktur in dem Event-Tupel $\tau_{B',R_{mod}}$ angefordert. Für die Index-Anforderungen kann der Angreifer eine Index-Datenstruktur zu R_{mod} hinzufügen, die Informationen über Blobs speichert, die nicht in R_{mod} vorhanden sind. Dieser Index wird korrekt verifiziert und verändert die Befehlsausführung nicht, weil er niemals von Restic verwendet wird, da alle Blobs, über die dieser Index Informationen speichert nicht Teil von R_{mod} sind. Für die Lock-Anforderungen kann der Angreifer ein nicht exklusives Lock zu R_{mod} hinzufügen. Dieses Lock wird korrekt verifiziert und verändert die Befehlsausführung nicht, da *backup* Befehle und *restore* Befehle nicht exklusive Locks ignorieren und ohne Fehler den Befehl vollständig ausführen. Für die Snapshot-Anforderungen kann der Angreifer eine Snapshot-Datenstruktur zu R_{mod} hinzufügen, die für B' nicht als Parent-Snapshot gewählt wird. Dieser Snapshot wird korrekt verifiziert und beeinflusst die Befehlsausführung nicht, da er nicht als Parent-Snapshot bei der Bestimmung des Latest-Snapshot gewählt wird.
Mit allen diesen Replay-Angriffen würde $\mathcal{A}$ das Sicherheitsspiel $Exp^{EUF-CMA}_{Restic}$ gewinnen. Allerdings beeinflusst keine der Änderungen des Angreifers die Befehlsausführung bis auf das Anfordern einer zusätzlichen Datenstruktur. Damit werden diese Replay-Angriffe als trivial bezeichnet, aber sie müssen für die Vollständigkeit erwähnt werden.

5.4.2.2. Replay-Angriffe für den *backup* Befehl

Für $type(B') = backup$ können nur Datenstrukturen vom Typ Snapshot, Index oder Tree-Blob für nicht triviale Replay-Angriffe verwendet werden. Indices werden nur verwendet, um zu vermeiden, das Duplikate von Data-Blobs oder Tree-Blobs gespeichert werden oder um Informationen über anzufordernde Tree-Blobs zu erhalten. Ein Angreifer kann Indices zu R_{mod} hinzufügen, die Informationen über durch Befehl B' erzeugt Blobs besitzen, die jedoch gar nicht in R_{mod} gespeichert sind. Dadurch würde Restic diese Blobs nicht mehr an S_{Backup} schicken, um diese Blobs zu speichern.
Tree-Blobs werden durch einen *backup* Befehl nur angefordert, wenn sie zu dem Verzeichnisbaum des Parent-Snapshots gehören und dieser gerade traversiert wird. Ein Verzeichnisbaum ist in Restic so dargestellt, dass jeder Tree-Blob einen Knoten im Verzeichnisbaum repräsentiert und die ID aller Tree-Blobs speichert, die Kindknoten dieses Knotens sind. Ein Angreifer, der also einen Tree-Blob in R_{mod} durch einen anderen Tree-Blob ersetzt, muss sicherstellen, dass entweder beide Tree-Blobs die gleiche ID besitzen oder dass auch der Tree-Blob des Elternknotens durch einen Tree-Blob ersetzt wurde, der die neue ID des eingesetzten Tree-Blobs speichert. Damit ergibt sich im zweiten Fall eine Abhängigkeit, die bis zum Root-Tree-Blob des Verzeichnisbaums des Parent-Snapshots reicht. Da der Parent-Snapshot ebenfalls die ID des Root-Tree-Blobs speichert, müsste auch der Parent-Snapshot durch einen Snapshot ersetzt werden, der die neue ID des neuen Root-Tree-Blobs enthält.

Ein Angreifer $\mathcal{A}$ der einen Tree-Blob austauschen möchte, müsste also alle Tree-Blobs bis zum Root-Tree-Blob austauschen und damit auch den Snapshot, der auf diesen Root-Tree-Blob verweist. $\mathcal{A}$ könnte zu R_{mod} alle Backupdaten eines früheren *backup* Befehls hinzufügen und den Snapshot dieser Backupdaten als Parent-Snapshot verwenden. Das wäre ebenfalls ein korrekter Replay-Angriff. Ein Angreifer hätte damit die Kontrolle über die Data-Blobs, die dem Repository hinzugefügt werden, da die Node-Einträge der Tree-Blobs zur Änderungsdetektion genutzt werden und somit entschieden wird, welche Daten zum Repository hinzugefügt werden.
In jedem Fall würde ein Replay-Angriff für einen *backup* Befehl, dem Angreifer nur die Möglichkeit geben zu entscheiden, welche Datenstrukturen durch den Befehl dem Repository hinzugefügt werden sollen. Diese Macht besitzt jedoch auch ein löschender Angreifer. Damit bieten Replay-Angriffe einem löschenden Angreifer in der Realität keinen Vorteil.

5.4.2.3. Replay-Angriffe für den *restore* Befehl

Für $type(B') = restore$ können nur Datenstrukturen vom Typ Snapshot, Index, Tree-Blob oder Data-Blobs für nicht triviale Replay-Angriffe verwendet werden. Genau, wie bei dem *backup* Befehl werden für den *restore* Befehl nur Tree-Blobs des Parent-Snapshots angefordert. Außerdem sind die angeforderten Data-Blobs ebenfalls Teil des Parent-Snapshots und die IDs der Data-Blobs werden in den angeforderten Tree-Blobs gespeichert. Dadurch ergibt sich genauso, wie beim *backup* Befehl, eine Abhängigkeit, durch die ein Angreifer $\mathcal{A}$ alle Backupdaten des Parent-Snapshot zu R_{mod} hinzufügen muss, sobald der Angreifer einen der angeforderten Tree-Blob oder Data-Blobs ersetzen möchte. $\mathcal{A}$ kann also alle Backupdaten eines früheren *backup* Befehls zu R_{mod} hinzufügen und den Snapshot dieser Backupdaten als Parent-Snapshot verwenden. Damit kann $\mathcal{A}$ ungewollte Zustände auf dem Verzeichnissystem auf S_{Restic} hervorrufen, da falsche Backupdaten wiederhergestellt werden. Der Angreifer $\mathcal{A}$ kann allerdings nur Daten wiederherstellen, von denen er Benutzer des Repositorys selber ein Backup durchgeführt hatte.
Befanden sich zu irgendeinem Zeitpunkt in dem Repository schädliche Daten, könnte ein Angreifer, diesen Replay-Angriff ausnutzen, um diese Daten auf S_{Restic} beim nächsten *restore* Befehl wiederherzustellen. Schädliche Daten könnten beispielsweise veraltete Versionen von Dateien des Betriebssystems oder anderen Programmen sein, die Sicherheitslücken aufweisen, die durch diesen Angriff wiederhergestellt werden könnten. Im schlimmsten Fall befanden sich zu irgendeinem Zeitpunkt Backups von Schadware im Repository, die $\mathcal{A}$ dem gleichen Repository wieder hinzufügen kann und so den nächsten *restore* Befehl dazu bringen kann, diese Backupdaten auf S_{Restic} wiederherzustellen. Natürlich gilt das nur unter der Voraussetzung, dass ein Angreifer weiß, welche Backupdaten Schadware enthielten und dass der Snapshot dieser Backupdaten zu dem Target-Pfad des *restore* Befehls passt.

5.4.2.4. Replay-Angriffe für den *prune* Befehl

Für $type(B') = prune$ können nur Datenstrukturen vom Typ Snapshot, Index, Tree-Blob oder Data-Blobs für nicht triviale Replay-Angriffe verwendet werden. Der *prune* Befehl traversiert

beginnend bei dem Root-Tree-Blob jedes Snapshots deren kompletten Verzeichnisbaum und fordert alle Tree-Blobs und Data-Blobs an, die auf dem Weg passiert werden. Das sind auch die einzigen Blobs, die durch den *prune* Befehl angefordert werden. Damit kann ein Replay-Angriff auch für den *prune* Befehl nicht durch das hinzufügen oder ersetzen eines einzelnen Tree-Blobs oder Data-Blobs durchgeführt werden. Ein Angreifer müsste für den *prune* Befehl ebenfalls alle Backupdaten eines früheren *backup* Befehls zu R_{mod} hinzufügen. Dadurch könnte der Angreifer durch einen Replay-Angriff auf den *prune* Befehl Snapshots, Tree-Blobs und Data-Blobs, die aus dem Repository entfernt wurden, wieder zum Repository hinzufügen.
Ein manipulierender Angreifer kann jedoch frühere Indices, Packs und Snapshots jederzeit zu einem Repository hinzufügen. Daher erhält ein manipulierender Angreifer durch einen Replay-Angriff auf einen *prune* Befehl keinen Vorteil.

5.4.2.5. Restic-Injection Replay

Ein realer Angreifer, der die Möglichkeit besitzt, zu bestimmen, von welchen Daten ein Backup durchgeführt wird, kann folgenden Angriff durchführen. Der Angreifer könnte eine Datei in S_{Restic} einschleusen, deren Inhalt das JSON-Format einer anderen Datenstruktur von Restic ist, die der Angreifer gerne im Repository hätte. Sofern diese Datei nicht durch das Chunking-Verfahren in mehrere Blobs zerlegt wird, wird diese Datei als ein verschlüsselter und authentifizierter Data-Blob zum Repository hinzufügt. Dieser Data-Blob entspricht jedoch exakt der verschlüsselten und authentifizierten Restic-Datenstruktur, die der Angreifer in die ursprüngliche Datei geschrieben hatte. Der Angreifer kann nun den Data-Blob in das zu dem Typ der Datenstruktur passende Verzeichnis des Repositorys schieben und hat damit eine Art Replay-Angriff erschaffen, in der Restic ahnungslos glaubte einen Data-Blob zu verschlüsselt und zu authentifizieren, aber in Wirklichkeit eine beliebige durch den Angreifer gewählte Datenstruktur verschlüsselt und authentifiziert hat. Dadurch kann der Angreifer beliebige Datenstruktur von allen möglichen Typen, außer Key-Datenstrukturen in das Repository schleusen, ohne Kenntnisse über den Masterkey des Repositorys zu besitzen.
Ein Angreifer könnte beispielsweise eine Datei im Anhang einer E-Mail an S_{Restic} schicken, die in einem Verzeichnis gespeichert wird, auf dem ein *backup* Befehl aufgerufen wird.

5.4.2.6. Replay-Angriffe durch ID-Kollision

Es ist ebenfalls ein Replay-Angriff möglich, sobald es zu einer Kollision der IDs zweier Restic-Datenstrukturen kommt. Das konkrete Primitiv in Restic, das die Restic-ID berechnet ist $SHA256$ über einem möglichen Raum von 2^{256} möglichen IDs. Die Wahrscheinlichkeit, dass so eine Kollision auftritt, ist sehr gering. Gerade wenn man bedenkt, dass andere konkrete Schranken von Restics konkreten Primitiven fordern, dass maximal $2^{47,7}$ Datenblöcke verschlüsselt werden und dadurch auch deutlich weniger als $2^{47,7}$ Datenstrukturen existieren.
Ein solcher Angriff könnte dazu führen, dass ein Angreifer einen korrumpierten Data-Blob oder Tree-Blob mit gleicher ID in ein Pack einschleusen könnte. Allerdings müssten diese

Blobs weiterhin mit dem Masterkey des Repositorys durch Restic erstellt worden sein, was einen Replay-Angriff durch eine ID-Kollision sehr unwahrscheinlich macht. Eher bricht die Vertraulichkeit von Restic, als dass ein effektiver Replay-Angriff durch eine ID-Kollision möglich ist.

5.4.3. Sicherheitsspiel ohne erlaubte Replay-Angriffe

Interessant ist ebenfalls, wie sich Restics Integrität verhält, wenn man Replay-Angriffe ausschließt. Dazu wird das Sicherheitsspiel $Exp_{Restic}^{EUF-CMA-NR}$ definiert. Dieses Sicherheitsspiel ist eine Abwandlung von $Exp_{Restic}^{EUF-CMA}$, bei der der Angreifer keine Replay-Angriffe durchführen darf. Dazu wird lediglich für $Exp_{Restic}^{EUF-CMA-NR}$ eine weitere Einschränkung zu der Menge $constraints(B', type(B'), R_{mod}, r')$ hinzufügt.
Damit ist die Menge $constraints$ für $Exp_{Restic}^{EUF-CMA-NR}$ definiert als:

- $R_{r'} \overset{del*}{\not\longrightarrow} R_{mod}$
- $K_{R_{mod}}$=$K_{R_{r'}}$
- $\exists e_i \in \tau_{B',R_{mod}}$ mit $op_i = receive \wedge data_i \in R_{mod} : data_i \notin R_{r'}$ **oder** $structure_{R_{mod}}(e_i) \neq structure_{R_{r'}}(e_i)$
- $\exists e_i \in \tau_{B',R_{mod}}$ mit $op_i = receive \wedge data_i \in R_{mod} : \forall R_t : \exists c \in relC(data_i) : c \notin R_t$

Die neue Einschränkung verwendet eine Hilfsfunktion $relC(data_i)$. Da ein Event-Tupel aus Sicht von S_{Backup} konstruiert ist, kann aus einem Event-Tupel nicht abgeleitet werden, welche angeforderten Blobs aus einem Event e_i mit $type_i = BlobChunk$, wirklich von Restic verwendet werden. An dieser Stelle kommt die Funktion $relC(data_i)$ ins Spiel, die alle von Restic verwendeten Datenstrukturen aus $data_i$ extrahiert. Genauer gesagt extrahiert die Funktion $relC(data_i)$ nur die Chiffrate, die von Restic verarbeitet werden (relevante Chiffrate).

$$relC(data_i) := \begin{cases} \{c_{data_i}\} & type_i \neq BlobChunk \\ \{c_{blob} \in data_i \mid blob \text{ wird von Restic verwendet}\} & \text{sonst} \end{cases}$$

c_{data_i} ist das Chiffrat das in $data_i$ steht und c_{blob} ist das Chiffrat eines Blobs.
Die Definition für die Funktion $check$, wann direkt für die neue Menge $constraints$ übernommen werden.
Zusammengefasst wird für $Exp_{Restic}^{EUF-CMA-NR}$ die Einschränkung zu der Menge $constraints$ hinzugefügt, dass das Chiffrat mindestens eine durch Befehl B' angeforderte Datenstruktur aus R_{mod} in keinem früheren Repository-Zustand R_t von R existiert hatte. Das bedeutet zwangsläufig, dass $\mathcal{A}$ eine eigene Datenstruktur zu R_{mod} hinzugefügt hat. Da durch jeden Restic-Befehl nur eine Key-Datenstruktur angefordert wird und zwar die des initialen Repositorys, muss der Angreifer eine Datenstruktur hinzugefügt haben, die kein Key ist. Da das Chiffrat der hinzugefügten Datenstruktur bisher in keinem Repository-Zustand in der Orakel-Phase aufgetaucht ist, heißt das, das Chiffrat der hinzugefügten Datenstruktur

unterscheidet sich in mindestens einem Bit zu dem Chiffrat jeder anderen verschlüsselten und authentifizierten Datenstruktur. Daraus folgt, dass sich aus einem Angreifer $\mathcal{A}$ für $Exp_{Restic}^{EUF-CMA-NR}$ ein Angreifer $\mathcal{B}$ auf die EUF-CMA Sicherheit von Restics Authentifizierungsverfahren rekonstruieren lässt.
Warum müssen sich unbedingt die Chiffrate unterscheiden und nicht beispielsweise die MACs für das gleiche Chiffrat? Ein Angreifer der nur andere MACs für die gleichen Chiffrate berechnen kann ist keine Gefahr für Restics Integrität, da er keine Daten verändern kann, sondern nur alternative Repository-Zustände erschafft, die sich strukturell durch andere MACs unterscheiden, aber inhaltlich komplett identisch sind, durch gleiche Chiffrate.

5.4.4. Teil-Reduktion des Sicherheitsspiels $Exp_{Restic}^{EUF-CMA-NR}$

Um die Integrität von Restic zu beweisen, besteht der Reduktionsbeweis darin die EUF-CMA-NR Sicherheit von Restic zu beweisen. In diesem Kapitel wird gezeigt, wie $Exp_{Restic}^{EUF-CMA-NR}$ sich unter der Annahme, dass der Schlüssel k_{auth} für das Authentifizierungsverfahren zufällig gleichverteilt gezogen wird, auf $Exp_{Auth}^{EUF-CMA}$ von Restics Authentifizierungsverfahren reduzieren lässt.
Es wird ein Angreifer $\mathcal{A}$ für das Sicherheitsspiel $Exp_{Restic}^{EUF-CMA-NR}(\mathcal{A}, 1^\lambda)$ betrachtet, wobei λ der asymptotische Sicherheitsparameter ist. Daraus wird ein Angreifer $\mathcal{B}$ für das EUF-CMA Sicherheitsspiel $Exp_{Auth}^{EUF-CMA}(\mathcal{B}, 1^\lambda)$ für ein von Restic verwendetes abstraktes Authentifizierungsverfahren konstruiert. Es wird gezeigt und bewiesen, wie dieser Angreifer $\mathcal{B}$ das EUF-CMA Sicherheitsspiel gewinnt, wenn $\mathcal{A}$ das Sicherheitsspiel $Exp_{Restic}^{EUF-CMA-NR}$ gewinnt.

Annahme: Uniforme Keys:
Damit die gezeigte Reduktion funktioniert, wird angenommen, dass der Masterkey bestehend aus $mk_{enc} \xleftarrow{\$} \mathcal{K}_{enc}(\lambda)$ und $mk_{auth} \xleftarrow{\$} \mathcal{K}_{auth}(\lambda)$ und der Userkey $uk \xleftarrow{\$} \mathcal{K}_{user}(\lambda)$ zu Beginn des Spiels uniform gezogen werden.

Wahl des IVs:
In Restic wird genau ein IV uniform aus dem IV-Raum $\mathcal{IV}(\lambda) \setminus \{0^{l(\lambda)}\}$ für eine verschlüsselte und authentifizierte Datenstruktur gewählt, wobei $l : \mathbb{N} \to \mathbb{N}$ die Größe eines IVs bestimmt. Bei dem klassischen EUF-CMA Spiel wird für Authentifizierungsverfahren, die eine Nonce benötigen diese Nonce vom Orakel gewählt. In dieser Reduktion würde das die Simulation für $\mathcal{A}$ kaputt machen, da $\mathcal{B}$ einen IV wählen würde für ein Verschlüsselungsverfahren und das Orakel $O_{k_{auth}}^{Auth}$ würde eine weitere Nonce wählen für die Berechnung des MACs. Damit existieren zwei Nonces, die für unterschiedliche Primitive verwendet werden und das entspricht nicht dem, was $\mathcal{A}$ von Restic erwarten würde.
$\mathcal{B}$ wählt durch Restic bereits für jedes Chiffrat c, das an $O_{k_{auth}}^{Auth}$ übergeben wird einen IV $iv \xleftarrow{\$} \mathcal{IV}(\lambda) \setminus \{0^{l(\lambda)}\}$. Da $\mathcal{B}$ kein eigenständiger Angreifer ist, sondern ein Algorithmus, der für die Reduktion konstruiert ist, kann $\mathcal{B}$ den uniform gezogenen IV iv als Teil der Nachricht an das Orakel übergeben mit $O_{k_{auth}}^{Auth}(iv||c)$. Falls das Authentifizierungsverfahren eine Nonce

benötigt, kann *iv* verwendet werden. Falls das Authentifizierungsverfahren größere Nonces benötigt als das Verschlüsselungsverfahren, müsste eventuell der Nonce-Raum angepasst werden.

5.4.4.1. Teilnehmer

Für die Reduktion werden drei Teilnehmer betrachtet. Der erste Teilnehmer ist ein Angreifer $\mathcal{A}$ für das vorgestellte Sicherheitsspiel $Exp^{EUF-CMA-NR}_{Restic}(\mathcal{A}, 1^\lambda)$. Der zweite Teilnehmer ist ein Challenger $\mathcal{C}$, der der Challenger für das Sicherheitsspiel $Exp^{EUF-CMA}_{Auth}(\mathcal{B}, 1^\lambda)$ ist, auf das reduziert wird. Außerdem gibt es den Angreifer $\mathcal{B}$, der den Angreifer für das Sicherheitsspiel $Exp^{EUF-CMA}_{Auth}(\mathcal{B}, 1^\lambda)$ darstellt und für $\mathcal{A}$ das Sicherheitsspiel $Exp^{EUF-CMA-NR}_{Restic}$ simuliert. Dafür übernimmt $\mathcal{B}$ die meisten Aufgaben von $\mathcal{O}_{uk}$ aus $Exp^{EUF-CMA}_{Restic}$. $\mathcal{B}$ hat außerdem Zugriff auf ein Verschlüsselungsorakel $\mathcal{O}^{Auth}_{k_{auth}}$ für das betrachtete abstrakte Authentifizierungsverfahren.
Damit $\mathcal{B}$ die Aufgaben von $\mathcal{O}_{uk}$ übernehmen kann, muss $\mathcal{B}$ in der Lage sein die Befehlsausführung eines Restic-Befehls zu emulieren oder $\mathcal{B}$ muss eine Instanz von Restic besitzen, mit der $\mathcal{B}$ sich Restic-Befehle ausführen lassen kann. Dazu ist $\mathcal{B}$ wie auch $\mathcal{O}_{uk}$ sowohl S_{Restic}, als auch S_{Backup} in einem. $\mathcal{B}$ besitzt ebenfalls ein Repository R, auf dem alle von $\mathcal{A}$ übergebenen Restic-Befehle ausgeführt werden. Der Unterschied zu dem Repository von $\mathcal{O}_{uk}$ ist, dass $\mathcal{B}$ nicht den mk_{auth} Teil des Masterkeys von R verwendet, um MACs zu erstellen, sondern ein Authentifizierungsorakel $\mathcal{O}^{Auth}_{k_{auth}}$ mit Schlüssel k_{auth}. Der Masterkey von R wird jedoch weiterhin benutzt, um alle Datenstrukturen zu verschlüsseln, die $\mathcal{A}$ sieht.

5.4.4.2. Initialisierung

Der Challenger $\mathcal{C}$ zieht einen uniformen Schlüssel $k_{auth} \xleftarrow{\$} \mathcal{K}_{auth}(\lambda)$ aus dem gleichen Schlüsselraum für das abstrakte Verschlüsselungsverfahren. $\mathcal{C}$ initialisiert das Authentifizierungsorakel $\mathcal{O}^{Auth}_{k_{auth}} := Init(1^\lambda, k_{auth})$ und gibt dieses als Black-Box an $\mathcal{B}$.
$\mathcal{B}$ führt den *init* Befehls von Restic aus, um ein neues Repository R erstellen. Der *init* Befehl wählt einen Masterkey mk bestehend aus zwei uniformen Schlüsseln $mk_{enc} \xleftarrow{\$} \mathcal{K}_{enc}(\lambda)$ und $mk_{auth} \xleftarrow{\$} \mathcal{K}_{auth}(\lambda)$ und $mk := mk_{enc}||mk_{auth}$, sowie einen Salt $s \xleftarrow{\$} \mathcal{S}(\lambda)$. mk_{auth} wird von $\mathcal{B}$ niemals verwendet. Alle Datenstrukturen der Restic-Befehle von $\mathcal{A}$, die erstellt werden und verschlüsselt werden, übergibt $\mathcal{B}$ zusammen mit einem IV seinem Authentifizierungsorakel und lässt sich für jede Datenstruktur einen MAC berechnen. Wie auch im EUF-CMA-NR Sicherheitsspiel für Restic wird kein Benutzerpasswort für den *init* Befehl verwendet, sondern $\mathcal{B}$ wählt direkt einen Userkey $uk \xleftarrow{\$} \mathcal{K}_{user}(\lambda)$. Der Masterkey mk wird mit uk verschlüsselt und authentifiziert und in den *data* Eintrag der Key-Datenstruktur K geschrieben. Die Key-Datei K wird daraufhin dem Repository R hinzugefügt. Außerdem erzeugt der *init* Befehl eine Config-Datei C für R. $\mathcal{B}$ verwendet sein Orakel, um für das Chiffrat $iv_{config}||c_{config}$ der Config-Datenstruktur authentifizieren zu lassen und erhält

$mac_{config} \xleftarrow{\$} O^{Auth}_{k_{auth}}(iv_{config}||c_{config})$. $\mathcal{B}$ bildet $C := iv_{config}||c_{config}||mac_{config}$ und fügt die Datei zu R hinzu. Ein initiales Repository ist leer, bis auf eine Key-Datei und eine Config-Datei.
$\mathcal{B}$ übergibt den initialen Repository-Zustand R_0 von R an $\mathcal{A}$.

5.4.4.3. Ablauf der Reduktion

$\mathcal{A}$ darf beliebige Restic-Befehle B und beliebige gültige Referenzen r auf frühere Repository-Zustande R_r wählen und an $\mathcal{B}$ schicken. $\mathcal{B}$ stellt den Repository-Zustand R_r für R wieder her und bestimmt ein Event-Tupel für die Befehlsausführung von B auf R. $\mathcal{B}$ simuliert die Befehlsausführung von B auf R und bestimmt damit ein Event-Tupel τ_t wie folgt. Soll eine Datenstruktur eines Events e_i verschlüsselt und authentifiziert werden, zieht $\mathcal{B}$ einen uniformen IV $iv \xleftarrow{\$} \mathcal{IV}(\lambda) \setminus \{0^{l(\lambda)}\}$ und verschlüsselt mit dem Masterkey-Teil mk_{enc} von R die Datenstruktur zu c. $\mathcal{B}$ benutzt das Orakel, um einen MAC $mac \xleftarrow{\$} O^{Auth}_{k_{auth}}(iv||c)$ für das Chiffrat c mit dem Schlüssel k_{auth} zu berechnen. Der *data* Eintrag des Events e_i wird dann auf $data_i := iv||c||mac$ gesetzt. Ist das Event ein Send-Event für ein Pack, besteht $data_i$ aus mehreren solcher Konkatenation ($data_i = iv_1||c_1||mac_1|| \ldots ||iv_m||c_m||mac_m$). Das Vorgehen ist jedoch immer dasselbe.
Müsste $\mathcal{B}$ eine verschlüsselte und authentifiziert Datenstruktur verifizieren wird die Verifikation übersprungen, $\mathcal{B}$ den Schlüssel k_{auth} nicht kennt und auch keinen Zugriff auf ein Verifikationsorakel besitzt. In Restic werden Datenstrukturen nur verifiziert, wenn sie aus dem Repository angefordert werden in einem Receive-Event. Da $\mathcal{B}$ vor der Ausführung des Befehls B einen bekannten Repository-Zustand R_r wiederherstellt, kann $\mathcal{B}$ sicher sein, dass alle Datenstrukturen aus R von Restic mit k_{auth} korrekt verifiziert werden würden. $\mathcal{A}$ bekommt von der Verifikation einer Datenstruktur nur durch das Event-Tupel nichts mit und daher stellt die fehlende Verifikation keine Verletzung der Simulation durch $\mathcal{B}$ dar. Die Verifikation hat auch keine Auswirkung auf die Befehlsausführung, bis auf das Sicherstellen, dass die Datenstruktur nicht manipuliert wurde.
Bis auf diese zwei Ausnahmen ist der die Konstruktion des Event-Tupels τ_t fest durch Restic definiert. Das resultierende τ_t jedes Befehls $\mathcal{B}$ entspricht der Ausführung des Befehls B auf dem Repository R mit einem Masterkey $mk_{enc}||k_{auth}$. Damit ist $\mathcal{A}$ kein Unterschied zu einer echten Restic-Instanz erkennbar, die den Authentifizierungsteil mk_{auth} ihres Masterkeys kennt und verwendet.
In der Challenge-Phase wählt $\mathcal{A}$ ein Tupel $(B', type(B'), R_{mod}, r')$ und schickt dieses an $\mathcal{B}$. $\mathcal{B}$ kennt alle früheren Repository-Zustände von R und damit auch $R_{r'}$. $\mathcal{B}$ überprüft, ob die Einschränkungen aus der Menge *constraints* von $Exp^{EUF-CMA-NR}_{Restic}$ für das Tupel des Angreifers erfüllt sind. Falls das nicht der Fall ist, verwirft $\mathcal{B}$ die Challenge und $\mathcal{A}$, sowie $\mathcal{B}$ verlieren beide ihre Sicherheitsspiele. Sind alle Einschränkungen erfüllt, muss nach Kapitel 5.4.3 mindestens ein Receive-Event e_{diff} in $\tau_{B',R_{mod}}$ vorkommen, dessen $data_{diff}$ durch $\mathcal{A}$ verändert wurde, sodass es mindestens ein Chiffrat enthält, das bisher nicht durch $O^{Auth}_{k_{auth}}$ authentifiziert wurde und von B' angefordert und verwendet wird.
Bis zu diesem Event e_{diff} führt $\mathcal{B}$ den Befehl B' aus und wann immer eine Datenstruktur

authentifiziert oder verifiziert wird, überspringt $\mathcal{B}$ diesen Schritt. Das ändert an der Bestimmung von $\tau_{B',R_{mod}}$ nichts, da die Befehlsausführung aller drei betrachteten Befehle durch Send-Events nicht beeinflusst wird. Somit ist egal, ob $\mathcal{B}$ die Datenstrukturen aus $\tau_{B',R_{mod}}$ korrekt authentifiziert oder nicht.
$\mathcal{B}$ führt so lange den Befehl B' auf R_{mod} aus, bis eine von $\mathcal{A}$ modifizierte Datenstruktur angefordert wird und durch Restic verifiziert werden würde. Diese Datenstruktur $iv_{diff}||c_{diff}||mac_{diff}$ gibt $\mathcal{B}$ als seine Fälschung für k_{auth} an den Challenger weiter.

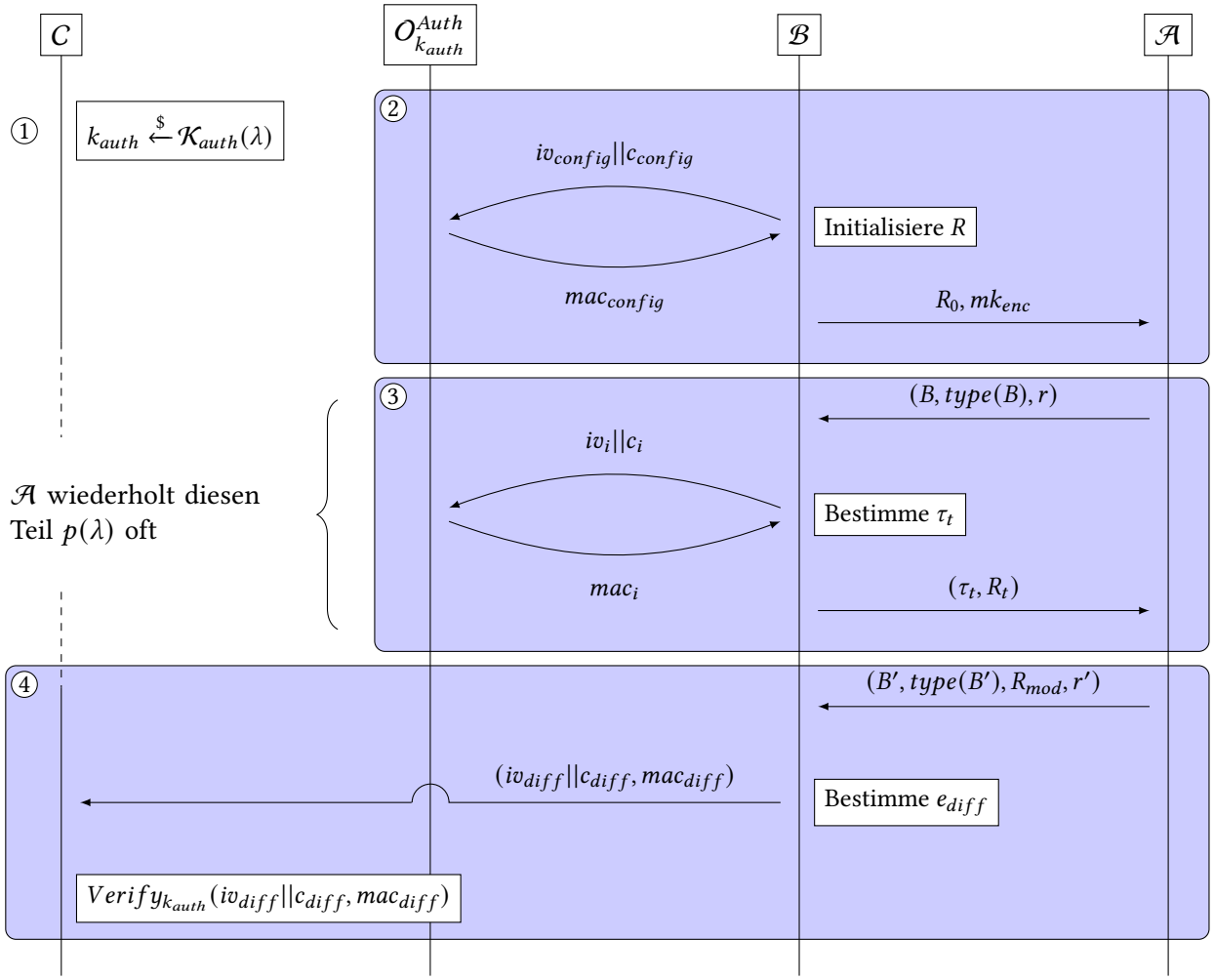

Figure 5.4.: Reduktion eines Angreifers $\mathcal{A}$ für das Sicherheitsspiel $Exp^{EUF-CMA-NR}_{Restic}(\mathcal{A}, 1^\lambda)$ auf einen Angreifer $\mathcal{B}$ für die EUF-CMA Sicherheit von Restics Authentifizierungsverfahren

1. Der Challenger wählt einen zufällig gleichverteilten Schlüssel $k_{auth} \xleftarrow{\$} \mathcal{K}_{auth}(\lambda)$. Der Challenger initialisiert das Authentifizierungsorakel $O^{Auth}_{k_{auth}} := Init(1^\lambda, k_{auth})$ für das EUF-CMA Sicherheitsspiel von Restics abstraktem Authentifizierungsverfahren.
2. $\mathcal{B}$ initialisiert das Repository R und zieht für die Config einen uniformen IV $iv_{config} \xleftarrow{\$} \mathcal{IV}(\lambda) \setminus \{0^{l(\lambda)}\}$. $\mathcal{B}$ verschlüsselt die Config-Datenstruktur mit mk_{enc}. Dann verwendet

$\mathcal{B}$ das Orakel $O^{Auth}_{k_{auth}}(iv_{config}||c_{config})$, um sich einen MAC mac_{config} für die Config-Datenstruktur berechnen zu lassen. $\mathcal{B}$ wählt einen uniformen Masterkey mk für R mit $mk_{enc} \xleftarrow{\$} \mathcal{K}_{enc}(\lambda)$ und $mk_{auth} \xleftarrow{\$} \mathcal{K}_{auth}(\lambda)$. Außerdem wählt $\mathcal{B}$ einen uniformen Userkey $uk \xleftarrow{\$} \mathcal{K}_{user}(\lambda)$. $\mathcal{B}$ verschlüsselt und authentifiziert den Masterkey mit uk und fügt das Ergebnis in den *data* Eintrag der Key-Datenstruktur von R ein. Da ein initiales Repository sonst keine Daten enthält, schickt $\mathcal{B}$ diesen Repository-Zustand R_0 zusammen mit mk_{enc} an $\mathcal{A}$.

3. $\mathcal{A}$ darf ein Befehlstupel B für einen beliebigen der drei betrachten Restic-Befehle und dessen Parameter wählen. Dieses Befehlstupel B schickt $\mathcal{A}$ zusammen mit dem Befehlstyp $type(B)$ und einer Referenz r auf einen früheren Repository-Zustand R_r von R an $\mathcal{B}$. R_r ist der Zustand von R nach der Ausführung des r-ten gewählten Befehls von $\mathcal{A}$. $\mathcal{B}$ stellt für R den Repository-Zustand R_r wieder her und führt den Befehl B auf R aus, um ein Event-Tupel τ_t zu erhalten. Während der Ausführung von B verwendet O den Masterkey mk_{enc}, um Datenstrukturen zu verschlüsseln und das Orakel $O^{Auth}_{k_{auth}}$, um die verschlüsselten Datenstrukturen zu authentifizieren. Nachdem der Befehl B auf R ausgeführt wurde, schickt $\mathcal{B}$ (τ_t, R_t) an $\mathcal{A}$ zurück.
   $\mathcal{A}$ darf diesen Ablauf für $p(\lambda)$ viele Befehlstupel wiederholen, wobei $p(\lambda)$ ein Polynom in λ ist.
4. Nachdem $\mathcal{A}$ diesen Ablauf für $p(\lambda)$ viele Restic-Befehle wiederholt hat, wählt $\mathcal{A}$ ein Tupel $(B', type(B'), R_{mod}, r')$, das die *constraints* für $Exp^{EUF-CMA-NR}_{Restic}$ erfüllt. $\mathcal{A}$ übergibt dieses Tupel an $\mathcal{B}$ und $\mathcal{B}$ führt den Befehl B' auf R_{mod} solange aus, bis ein Receive-Event auftritt, für das gilt, dass dieses Event eine Chiffrat c_{diff} anfordert zu dem $\mathcal{A}$ noch nie einen MAC gesehen hat. Nach der Definition der Menge *constraints* für $Exp^{EUF-CMA-NR}_{Restic}$ gibt es so ein Event in $\tau_{B',R_{mod}}$, sofern $\mathcal{A}$ das Sicherheitsspiel gewinnen will.
   $\mathcal{B}$ verwendet $(iv_{diff}||c_{diff})$ und mac_{diff} selbst als Fälschung für die EUF-CMA Sicherheit und schickt beides an den Challenger
   Der Challenger verifiziert das Paar $(iv_{diff}||c_{diff}, mac_{diff})$ mit dem Verifikationsalgorithmus für das abstrakte Authentifizierungsverfahren. Entspricht die Ausgabe der Verifikation eins, konnte $\mathcal{A}$ erfolgreich eine Fälschung finden und damit gewinnt $\mathcal{B}$ das EUF-CMA Sicherheitsspiel.

5.4.4.4. Erfolgswahrscheinlichkeit von $Adv^{EUF-CMA}_{Enc}(\mathcal{B}, \lambda)$

$\mathcal{B}$ kann das Sicherheitsspiel $Exp^{EUF-CMA-NR}_{Restic}$ in der Orakel-Phase mit $mk_{enc}||k_{auth}$ als Masterkey exakt für $\mathcal{A}$ simulieren. In der Challenge-Phase hingegen, reicht $\mathcal{B}$ eine gültige Fälschung, um sein Sicherheitsspiel zu gewinnen. Für $\mathcal{A}$ hingegen gelten weiter Einschränkungen für seine Gewinnbedingung, da ein kompletter Restic-Befehl erfolgreich ausgeführt werden muss.
$\mathcal{B}$ ist so definiert, dass es die erste Datenstruktur aus R_{mod} wählt, die in der Orakel-Phase noch nie authentifiziert wurde und von Restic bei der Ausführung von B' angefordert und verwendet wird. Da alle Datenstrukturen des Repositorys mit einem MAC versehen sind, wählt $\mathcal{B}$ diese Datenstruktur und den zugehörigen MAC als seine eigene Fälschung und

gibt beides an den Challenger. Dann gibt es folgende Fälle:
Fall 1: $\tau_{B',R_{mod}}$ enthält kein Receive Event, das eine solche Datenstruktur anfordert und verwendet. In diesem Fall verwirft $\mathcal{B}$ die Ausgabe von $\mathcal{A}$ und $\mathcal{A}$ verliert das Spiel ebenso wie $\mathcal{B}$, da die letzte Eigenschaft aus der Menge *constraints* für $Exp^{EUF-CMA-NR}_{Restic}$ nicht erfüllt ist.
Fall 2: Die Fälschung des Angreifers $\mathcal{A}$ ist nicht gültig und $\mathcal{B}$ verliert das Spiel. Dann verliert auch $\mathcal{A}$ das Spiel, da eine durch Restic angeforderte und verwendete Datenstruktur immer erfolgreich verifiziert werden muss, da es sonst zu einem Fehler bei der Befehlsausführung kommt.
Fall 3: Die gewählte Fälschung des Angreifers $\mathcal{A}$ war gültig. Dann gewinnt $\mathcal{B}$ sofort das Sicherheitsspiel. Es kann aber sein, dass $\mathcal{A}$ sein Sicherheitsspiel nicht gewinnt, da $\mathcal{A}$ weitere Fälschungen angefertigt hat, die in $\tau_{B',R_{mod}}$ verwendet werden aber keine erfolgreichen Fälschungen sind.
Damit ist die Erfolgswahrscheinlichkeit eines Angreifers $\mathcal{B}$ in jedem Fall mindestens so groß, wie die Erfolgswahrscheinlichkeit von $\mathcal{A}$. Daraus folgt:

$$Adv^{EUF-CMA-NR}_{Restic}(\mathcal{A}, \lambda) \leq Adv^{EUF-CMA}_{Auth}(\mathcal{B}, \lambda)$$

5.4.4.5. Laufzeit von $\mathcal{B}$

$\mathcal{B}$ führt in alle Befehle von $\mathcal{A}$ einmal auf R aus mit $O^{Auth}_{k_{auth}}$. Damit unterscheiden sich die beiden Laufzeiten nur durch den polynomiellen Overhead das Orakel $O^{Auth}_{k_{auth}}$ zu bedienen. $\mathcal{B}$ spart sogar etwas Zeit ein, indem keine Verifikationen in der Orakel-Phase durchgeführt werden. Für einen Angreifer $\mathcal{A}$ mit Laufzeit t, ist die Laufzeit t_1 von $\mathcal{B}$ also beschränkt durch $t_1 \leq t + poly(\lambda)$. Damit ist für alle PPT-Angreifer $\mathcal{A}$ in λ der konstruierte Angreifer $\mathcal{B}$ ebenfalls ein PPT-Angreifer in λ.

5.4.4.6. Fazit

Damit konnte ein PPT-Angreifer $\mathcal{A}$ für $Exp^{EUF-CMA-NR}_{Restic}(\mathcal{A}, \lambda)$ erfolgreich auf einen PPT-Angreifer $\mathcal{B}$ für $Exp^{EUF-CMA}_{Auth}(\mathcal{B}, \lambda)$ reduziert werden.

5.4.5. Reduktionsbeweis für die EUF-CMA-NR Sicherheit von Restic

In diesem Kapitel wird die EUF-CMA-NR Sicherheit von Restic auf die Sicherheit der verwendeten kryptografischen Primitive reduziert. Dazu wird gezeigt, dass jeder PPT-Angreifer $\mathcal{A}$ das Sicherheitsspiel $Exp^{EUF-CMA-NR}_{Restic}(\mathcal{A}, 1^\lambda)$ mit nicht vernachlässigbarer Wahrscheinlichkeit λ gewinnt, einen Angreifer $\mathcal{B}$ impliziert, der eins der zugrundeliegenden Primitive ebenfalls mit nicht vernachlässigbarer Wahrscheinlichkeit in λ bricht. Es werden zwei Welten betrachtet, wobei $\mathcal{W}_0$ die Welt des Sicherheitsspiels $Exp^{EUF-CMA-NR}_{Restic}$ mit einer abstrakten KDF ist und $\mathcal{W}_1$ die Welt von $Exp^{EUF-CMA-NR}_{Restic}$, bei der der Userkey uniform gezogen wird. Zunächst wird die Ununterscheidbar zwischen $\mathcal{W}_0$ und $\mathcal{W}_1$ abgeschätzt

und dann wird ein Angreifer auf $\mathcal{W}_1$ auf einen Angreifer der EUF-CMA Sicherheit des Authentifizierungsverfahrens reduziert.
Erst wird eine allgemeine Reduktion für abstrakte Sicherheitsprimitive vorgenommen. Danach werden die konkreten Sicherheitsabschätzungen für Restics konkrete Primitive eingesetzt und eine Gesamtabschätzung für Restics EUF-CMA-NR Sicherheit getroffen.

5.4.5.1. Allgemeine Reduktion

In diesem Kapitel werden zunächst nur abstrakte kryptografische Primitive betrachtet. Mitmilfe dieser abstrakten Primitive wird eine Gesamtabschätzung in λ für die Sicherheit von Restics EUF-CMA-NR Sicherheit getroffen. Damit kann die EUF-CMA-NR Sicherheit von Restic unabhängig der tatsächlichen kryptografischen Primitive ausgedrückt werden.

Schlüsselableitungsfunktion ($\mathcal{W}_0$ zu $\mathcal{W}_1$):
Schlüsselbasierte kryptografische Verfahren garantieren nur Sicherheit, wenn der verwendete Schlüssel mit hoher Entropie gewählt wurde oder sogar uniform ist. Da in $\mathcal{W}_1$ der Userkey durch eine KDF gewählt wird, muss zuerst gezeigt werden, dass die Ausgabe der KDF ununterscheidbar von einem uniform gezogenen Schlüssel ist. Daher findet der erste Schritt von $\mathcal{W}_0$ auf die Welt $\mathcal{W}_1$ statt, in der verwendete KDF durch eine echte Zufallsfunktion ersetzt wurde.

$$\mathcal{W}_0 : uk := KDF_{k_{KDF}}(1^\lambda, s) \qquad \rightarrow \qquad \mathcal{W}_1 : uk \xleftarrow{\$} \mathcal{K}_{user}(\lambda)$$

Die Wahrscheinlichkeit, dass ein Angreifer zwischen $\mathcal{W}_0$ und $\mathcal{W}_1$ unterscheiden kann, ist beschränkt durch die Wahrscheinlichkeit, dass die verwendete KDF von einer echten Zufallsfunktion unterscheidbar ist. Damit ist die Wahrscheinlichkeit, dass ein Angreifer zwischen $\mathcal{W}_0$ und $\mathcal{W}_1$ unterscheiden kann, beschränkt durch die PRF-Sicherheit $Adv^{PRF}_{KDF}(\mathcal{B}_1, \lambda)$ der verwendeten KDF. Somit ergibt sich die Abschätzung:

$$|Pr[\mathcal{A} \Rightarrow 1 \mid \mathcal{W}_0] - Pr[\mathcal{A} \Rightarrow 1 \mid \mathcal{W}_1]| \leq Adv^{PRF}_{KDF}(\mathcal{B}_1, \lambda)$$

Der Masterkey, der für die Verschlüsselung und Authentifizierung der Datenstrukturen verwendet wird, wird in dem IND$ Sicherheitsspiel und der abstrakten Modellierung von Restic bereits in $\mathcal{W}_0$ zufällig gleichverteilt gezogen.

EUF-CMA Reduktion:
Aus der Welt $\mathcal{W}_1$ kann $\mathcal{A}$ direkt auf einen Angreifer $\mathcal{B}_2$ für die EUF-CMA-NR Sicherheit von Restics Authentifizierungsverfahren reduziert werden. Diese Reduktion findet in Kapitel 5.4.4 statt und es ergibt sich die folgende Abschätzung durch die EUF-CMA Sicherheit von Restics abstraktem Authentifizierungsverfahren.

$$Adv^{EUF-CMA-NR}_{Restic}(\mathcal{A}, \lambda) \leq Adv^{EUF-CMA}_{Auth}(\mathcal{B}_2, \lambda)$$

Fazit der allgemeinen Reduktion:
Damit ist die Reduktion vollständig. Für die Wahrscheinlichkeit, dass ein Angreifer $\mathcal{A}$ das Sicherheitsspiel $Exp_{Restic}^{EUF-CMA-NR}(\mathcal{A}, 1^{\lambda})$ gewinnt, gilt die Gesamtabschätzung 5.5.

$$Adv_{Restic}^{EUF-CMA-NR}(\mathcal{A}, \lambda) \leq Adv_{KDF}^{PRF}(\mathcal{B}_1, \lambda) + Adv_{Auth}^{EUF-CMA}(\mathcal{B}_2, \lambda) \tag{5.5}$$

Da die endliche Summe vernachlässigbarer Funktionen ebenfalls vernachlässigbar ist, folgt folgende Aussage aus Abschätzung 5.5. Für einen PPT-Angreifer $\mathcal{A}$ gilt $Adv_{Restic}^{EUF-CMA-NR}(\mathcal{A}, \lambda) \leq negl(\lambda)$, sofern $Adv_{KDF}^{PRF}(\mathcal{B}_1, \lambda)$ und $Adv_{Auth}^{EUF-CMA}(\mathcal{B}_{\epsilon}, \lambda)$ vernachlässigbar in λ sind.

Restic ist damit EUF-CMA-NR-sicher, wenn die verwendete KDF PRF-sicher ist und das verwendete Authentifizierungsverfahren EUF-CMA-sicher ist.

Laufzeitbeschränkung der Reduktionsangreifer:
Die Reduktionsangreifer $\mathcal{B}_1$ und $\mathcal{B}_2$ führen den Angreifer $\mathcal{A}$ als Subroutine aus und erzeugen dabei nur einen polynomiellen Overhead. Dieser Overhead ist abhängig von λ. Da $\mathcal{A}$ polynomiell beschränkt in λ ist, sind auch die Reduktionsangreifer $\mathcal{B}_i$ polynomiell beschränkt in λ. Damit sind alle Reduktionsangreifer $\mathcal{B}_i$ effiziente Angreifer gegen die jeweiligen Primitive.

5.4.5.2. Betrachtung der Gesamtabschätzung bezüglich Restics Primitiven

In diesem Kapitel werden die abstrakten Primitive der allgemeinen Reduktion durch die konkreten von Restic verwendeten Primitive ersetzt. Damit wird die allgemeine Sicherheitsaussage auf Sicherheitseigenschaften der konkreten Primitive scrypt und Poly1305-AES128 reduziert. Die konkreten Schlüssellängen und Parametrisierungen der konkreten Primitive sind festgelegt. Daher wird der Sicherheitsparameter λ der asymptotischen Sicherheit durch eine konkrete Parametrisierungen der einzelnen Primitive ersetzt. Dadurch wird die asymptotische Vernachlässigbarkeitsaussage in λ durch konkrete Sicherheitsschranken ersetzt, die von der Laufzeit t des Angreifers $\mathcal{A}$ und der Anzahl der verschlüsselten und authentifizierten Datenstrukturen abhängt.
Da die Verschlüsselung nicht in der Reduktion betrachtet wird, geht der Schlüsselraum für Restics Verschlüsselungsverfahren nicht mit in die Betrachtung ein. Beim Einsetzen der Primitive werden die abstrakten Schlüsselräume durch folgende konkrete Räume ersetzt:

- $\mathcal{K}_{auth}(\lambda)$ wird zu zwei einzelnen Schlüsselräumen für Poly1305-AES128. Ein Schlüsselraum $\{0,1\}^{128}$ ist für den Schlüssel von AES128 und der andere Schlüsselraum ist für den 128 Bit langen Schlüssel von Poly1305. Wichtig hierbei zu erwähnen ist, dass der Schlüsselraum für Poly1305 wie in Kapitel 2.2.3 erwähnt nur eine Größe von 2^{106} besitzt.
- $\{0,1\}^{l(\lambda)}$ wird zu $\{0,1\}^{128} \setminus 0^{128}$ für den Nonce-Raum für Poly1305-AES128.

Alle betrachteten konkreten Schranken sind unabhängig von der Anzahl verschlüsselter Datenblöcke oder es wurden bereits Worst-Case-Abschätzungen vorgenommen für die maximale Anzahl Datenblöcke pro Datenstruktur. Daher ist die Anzahl verschlüsselter Datenblöcke kein konkreter Parameter. Die Sicherheitsaussage 5.6 gilt für alle Angreifer $\mathcal{A}$, die die folgenden Voraussetzung erfüllen:

- Die Gesamtanzahl q_{auth} generierten MACs überschreitet die Schranke q^max_{auth} nicht.
- Die Laufzeit t des Angreifers $\mathcal{A}$ liegt unterhalb der Brute-Force-Grenze von 2^{106}. Die Brute-Force-Grenze wird durch das schwächste verwendete Primitiv bestimmt. Ein Angreifer mit $t \geq 2^{106}$ könnte durch systematisches Durchprobieren aller IVs die Sicherheit trivial brechen. Solche Angreifer liegen außerhalb des betrachteten Bedrohungsmodells dieser Masterarbeit. Für $\mathcal{B}_1$ und $\mathcal{B}_2$ gilt $t_1 \leq t + p_{kdf}$ und $t_2 \leq t + q_{auth} \cdot p_{auth}$. Sowohl für t_1, als auch für t_2 ist der polynomielle Overhead vernachlässigbar in 2^{106}.

Abstrakte KDF $\rightsquigarrow$ scrypt:
Die abstrakte als PRF-sicher angenommene KDF aus Kapitel 5.2.2 wird in Restic durch das Primitiv scrypt ersetzt. Der abstrakte Schlüsselraum der $KDF_{k_{kdf}}$ für den Userkey wird konkret zu einem Schlüsselraum der Größe $2^{256} \cdot 2^{106} \cdot 2^{128} = 2^{490}$. Restic verwendet scrypt genau einmal pro Erstellung einer neuen Key-Datei. Da durch die drei betrachteten Restic-Befehle keine weitere Key-Datei, abseits der initialen Key-Datei, erstellt wird, wird scrypt genau ein mal verwendet. Damit wird der abstrakte Vorteil der abstrakten KDF zu folgendem konkreten Vorteil für scrypt, wobei $q_{kdf} = 1$ die Anzahl der scrypt Aufrufe ist:

$$Adv^{PRF}_{KDF}(\mathcal{B}_1, \lambda) \rightsquigarrow Adv^{PRF}_{scrypt}(\mathcal{B}_1, t_1, q_{kdf})$$

Für die Laufzeit gilt $t_1 \leq t + c_1$, wobei c_1 ein konstanter Overhead ist, da gilt $q_{kdf} = 1$. Scrypt ist ein speicherhartes Verfahren, das allgemein als sicher gegenüber Wörterbuchangriffen gilt. Ein konkreter Sicherheitsbeweis für die PRF-Eigenschaft konnte jedoch nicht gefunden werden. Dadurch, dass scrypt speicherhart ist, ist $Adv^{PRF}_{scrypt}(\mathcal{B}_1, t_1, 1)$ vernachlässigbar für einen Angreifer mit beschränkten Speicher- und Zeitressourcen. Die konkrete Schranke dieses Vorteils hängt dabei von den scrypt Parametern (n, r, p) ab, die Restic in jeder Key-Datei speichert. In Restic sind besitzen diese Parameter die Werte $(n = 2^{16}, r = 8, p = 1)$.

Abstraktes Authentifizierungsverfahren $\rightsquigarrow$ $Poly1305 - AES128$:
Das abstrakte EUF-CMA-sichere Authentifizierungsverfahren wird durch Restics Poly1305-AES128 ersetzt. Der abstrakte Schlüsselraum $\mathcal{K}_{auth}(\lambda)$ wird konkret durch die zwei Schlüsselräume der Größe 2^{106} für k_{Poly} und 2^{128} für k_{AES128} ersetzt. Der abstrakte IV-Raum $\mathcal{IV}(\lambda) \setminus 0^{l(\lambda)}$ für die Nonces von Poly1305-AES128 wird zu $\{0,1\}^{128} \setminus 0^{128}$. Die in Kapitel 4.4.2 hergeleitete obere Schranke von 2^{48} gibt die maximale Anzahl verschlüsselter Datenblöcke an, für die EUF-CMA Sicherheit von Restics Poly1305-AES128 gilt. Damit gilt nach Kapitel 4.4.2 für $q_{auth} \leq q^{max}_{auth} := 2^{48}$ und $\epsilon_{EUF-CMA} := Adv^{PRP}_{AES128}(\mathcal{B}_3, t_3, q_{auth}) + 2^{-31}$:

$$Adv^{EUF-CMA-NR}_{Auth}(\mathcal{B}_2, \lambda) \rightsquigarrow Adv^{EUF-CMA}_{Poly[AES128]}(\mathcal{B}_2, t_2, q_{auth}) \leq \epsilon_{EUF-CMA}$$

Gesamtabschätzung:
Durch das Einsetzen der konkreten Sicherheitsabschätzung der konkreten Primitive in die allgemeine Reduktion aus Gleichung 5.5 ergibt sich die konkrete Gesamtabschätzung 5.6 für die Erfolgswahrscheinlichkeit eines Angreifers $\mathcal{A}$ gegen die EUF-CMA-NR Sicherheit von Restic. Dabei muss der Angreife $\mathcal{A}$ allerdings die folgenden Beschränkungen erfüllen:

- Die Gesamtanzahl q_{auth} generierten MACs überschreitet die Schranke q^max_{auth} nicht.
- Für die Laufzeit t von $\mathcal{A}$ gilt $t < 2^{106}$.

$$\begin{aligned} Adv^{EUF-CMA-NR}_{Restic}(\mathcal{A}, t, q_{auth}) \leq\ & Adv^{PRF}_{scrypt}(\mathcal{B}_1, t_1, 1) \\ & + Adv^{PRP}_{AES128}(\mathcal{B}_3, t_3, q_{auth}) + 2^{-31} \end{aligned} \tag{5.6}$$

Die Gesamtschranke 5.6 ist eine Summe aus drei Termen, die jeweils die Sicherheit eines einzelnen Primitivs beschreiben oder eine statische Schranke darstellen. Alle Terme sind für die betrachteten Parameterbereiche vernachlässigbar. Damit gilt auch, dass $Adv^{EUF-CMA-NR}_{Restic}(\mathcal{A}, t, q_{auth})$ vernachlässigbar ist.

Theorem 4 (EUF-CMA-NR Sicherheit von Restic) *Sei $\mathcal{A}$ ein Angreifer gegen die EUF-CMA-NR Sicherheit von Restic mit Laufzeit $t < 2^{106}$.*
Unter den Annahmen:

- *scrypt ist PRF-sicher*
- *AES128 ist PRP-sicher*

gilt Restic ist EUF-CMA-NR-sicher. Konkreter gilt die Abschätzung 5.6.

5.4.5.3. Einordnung

Das Theorem 4 zeigt, dass die EUF-CMA-NR Sicherheit von Restic vollständig auf die Sicherheit der verwendeten Primitive zurückgeführt werden kann. Die Sicherheit wird dabei durch das schwächste Primitiv bestimmt. Im aktuellen Fall von Restic ist das Poly1305-AES128 mit einem Schlüsselraum der Größe 2^{106} für einen seiner Schlüssel, sowie die statische Schranke für die IV-Kollisionen. Solange allerdings die Schranke $q^{max}_{auth} = 2^{48}$ eingehalten wird, ist die Gesamtschranke 5.6 für alle relevanten Angreifer in der Praxis vernachlässigbar klein.

Nachwort:
Die Hauptreduktion des Sicherheitsspiels $Exp^{EUF-CMA-NR}_{Restic}$ hat auf die EUF-CMA Sicherheit des von Restic verwendeten Authentifizierungsverfahrens stattgefunden. Es ist ebenso möglich auf die Multi-Challenge Variante von EUF-CMA zu reduzieren. Auch bei dieser Variante ist die MC-EUF-CMA Sicherheit eine obere Schranke für den Vorteil eines Angreifers gegen EUF-CMA-NR von Restic. Lediglich die konkreten Sicherheitsparameter fallen wein wenig kleiner aus.

Auch für Restics Integrität wurden der Reduktionsbeweis und die konkreten Sicherheitsschranken nur für einen einzigen Aufruf der KDF betrachtet. Es sei dazu gesagt, dass die EUF-CMA-NR Sicherheit für Restic nicht bricht, wenn man weitere Key-Datenstrukturen zum Repository hinzufügt, solange die Anzahl in einem vertretbaren Rahmen bleibt. In der Realität werden nicht hunderte von Key-Datenstrukturen für ein Repository erstellt.

6. Fazit

In dieser Arbeit wurde erstmals ein formaler kryptografischer Sicherheitsbeweis für die Backup-Software Restic erbracht. Dazu wurden zwei Modellierungen für die drei wichtigsten Funktionalitäten von Restic vorgestellt. Die Backup-Funktion, die Restore-Funktion und die Prune-Funktion, vom Verwalten der erzeugten Backup-Daten.
Es wurde eine detaillierte Modellierung mit dem Anspruch an Realismus für alle drei Funktionen vorgestellt. Diese Modellierung ist einfacher verständlich als der Quellcode von Restic und nimmt trotzdem kaum Abstriche an Detailreichtum in Kauf. Mit dieser Modellierung lassen sich exakte realistische Abläufe nachvollziehen.
Für die Sicherheitsbeweise wurde jedoch eine abstraktere Modellierung gebraucht, bei der Restic als ein Zwei-Parteien-Protokoll modelliert wurde. Jeder Restic-Befehl kann als Tupel von Events dargestellt werden, die jeweils einen Nachrichtenaustausch repräsentieren. Mit dieser Darstellung lässt sich Restic als formales Protokoll betrachten, zu dem Sicherheitsspiele erstellt werden können.
Vorher wurde jedoch die informationstheoretische Sicherheit der von Restic verwendeten Primitive AES256 im Counter-Mode und Poly1305-AES128 betrachtet. Es wurden speziell für Restic Sicherheitsabschätzungen getroffen, die mögliche IV-Kollisionen berücksichtigen, die bei Restic auftreten können. Außerdem wurden Obergrenzen festgelegt für informationstheoretische Sicherheit der einzelnen Primitive. Basierend darauf wurden zwei Einsatzszenarien von Restic betrachtet und festgestellt, dass bei größeren Mengen von Daten die Sicherheit von Restic nach zwei Jahren ausgereizt ist. Es empfiehlt sich also alle zwei Jahre ein neues Repository mit einem neuen Masterkey anzulegen. Es wäre auch denkbar ein Re-Keying-Verfahren in Zukunft in Restic zu implementieren, allerdings müssten dafür noch einige Anpassung an Restic vorgenommen werden. Wird Restic beispielsweise von mehreren Benutzern betrieben, würde ein Re-Keying zwangsläufig zur Folge haben, dass alle Benutzer sich erneut bei dem Repository registrieren müssten. Das würde dann aber keinen Vorteil vom Neuerstellen eines Repository bieten. In jedem Fall kann die Sicherheit von Restics Vertraulichkeit nur bis zu einer verschlüsselten Datenmenge von ungefähr 4.000 Terrabyte gewährleistet werden.
Um in Zukunft die Sicherheit von Restic auch für andere kryptografische Primitive untersuchen zu können, wurden zwei Sicherheitsspiel für Restic definiert. Eins für die Vertraulichkeit und eins für die Integrität von Restic. Für beide dieser Sicherheitsspiele wurde ein allgemeiner Reduktionsbeweis mit abstrakten kryptografischen Primitiven geführt. Damit konnten zwei Abschätzungen für Restics Vertraulichkeit und Integrität unabhängig von den verwendeten Primitiven aufgestellt werden. Zum Schluss wurde die Sicherheit von Restic auf etablierte kryptografische Annahmen reduziert und so gezeigt, dass Restic bis zu einem gewissen Punkt Vertraulichkeit und Integrität garantieren kann.

Zukünftig könnten mit der vorgestellten abstrakten Modellierung weitere Restic-Befehle modelliert werden und andere Sicherheitseigenschaften bewiesen werden.

Bibliography

[1] Boris Alexeev, Colin Percival, and Yan X Zhang. *Chunking Attacks on File Backup Services using Content-Defined Chunking*. Cryptology ePrint Archive, Paper 2025/532. 2025. URL: https://eprint.iacr.org/2025/532.

[2] Daniel J Bernstein. "The Poly1305-AES message-authentication code". In: *International workshop on fast software encryption*. Springer. 2005, pp. 32–49.

[3] Morris Dworkin. "Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC, 2007". In: *NIST Special Publication (SP)* ().

[4] Valentin Lantigny. *Compress Backup Data from CBack*. Tech. rep. 2024.

[5] Meta. *ZSTD-Manual für ZSTD-Algorithmus von Meta*. URL: https://github.com/facebook/zstd/tree/dev/doc.

[6] Klaus Post. *Klauspost/compress/zstd: Eine GO Implementierung des ZSTD-Komprimierungsalgorithmus*. URL: https://github.com/klauspost/compress/tree/master/zstd.

[7] Restic. *Restic/rest-server: High performance HTTP server that implements Restic's rest backend API*. URL: https://github.com/restic/rest-server.

[8] Restic. *Restic/restic: Restic Software to create backups and restore backups*. URL: https://github.com/restic/restic.

[9] Phillip Rogaway. "Evaluation of some blockcipher modes of operation". In: *Cryptography Research and Evaluation Committees (CRYPTREC) for the Government of Japan* 630 (2011), p. 159.

[10] Kien Tuong Truong et al. "Breaking and fixing content-defined chunking". In: *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. 2025, pp. 2294–2308.

A. Appendix

A.1. Pseudocode für allgemeine Restic-Funktionen

Dieses Kapitel erklärt den Pseudocode bestimmter Restic-Funktionen, die in mehreren verschiedenen Restic-Befehlen verwendet werden.

A.1.1. Verschlüsseln und Authentifizieren

Die Funktion 16 zeigt, wie Restic einen Bytestring `plaintext` verschlüsselt und authentifiziert.

Funktion 16 Restics Verschlüsselungsalgorithmus für einen Bytestring

Require: k_{AES256}, (k_{Poly}, k_{AES128})
  function ENCRYPT($plaintext$)
    $Byte[16]\ iv \xleftarrow{\$} \{0,1\}^{128} \setminus 0^{128}$
    $Bytestring\ ciphertext := Enc^{iv}_{AES256}(plaintext, k_{AES256})$
    $Byte[16]\ mac := Auth^{iv}_{Poly1305}(ciphertext, k_{Poly}, k_{AES128})$
    return $iv||ciphertext||mac$
  end function

A.1.2. Entschlüsselung und Verifikation

Die Funktion 17 zeigt, wie Restic einen Bytestring `encryptedData` entschlüsselt und verifiziert. Die Konstante $size_{IV} := 16$ entspricht der Größe eines IVs in Bytes und $size_{MAC} := 16$ ist die Größe eines MACs in Bytes.

A.1.3. Anfordern einer Datei mit ID-Präfix

Für die Kommandozeilenbefehle von Restic können für bestimmte Optionen ID-Präfixe übergeben werden. Das sorgt dafür, das Restic nicht einfach irgendwelche Snapshots oder Keys aus dem Backupsystem lädt, sondern genau die Dateien, deren ID zu dem übergebenen ID-Präfix passt. Die Modellierung von Restics vorgehen befindet sich in Funktion 18. Zunächst fordert S_{Restic} die Metadaten aller Dateien des vorgegebenen Typs von S_{Backup} an.

Funktion 17 Restics Verfikation und Entschlüsselung verschlüsselter Daten

Require: k_{AES256}, (k_{Poly}, k_{AES128})
  function DECRYPT($encryptedData$)
    $Int\ size := size(encryptedData)$
    $Byte[16]\ iv := encryptedData[: size - size_{IV}]$
    $Bytestring\ ciphertext := encryptedData[size_{IV} : size_{MAC}]$
    $Byte[16]\ mac := Auth^{iv}_{Poly1305}(ciphertext, k_{Poly}, k_{AES128})$
    if $mac == encryptedData[size - 16 :]$ **then** ▷ Vergleich der MACs
      return $Dec^{iv}_{AES256}(ciphertext, k_{AES256})$
    else
      return $error$
    end if
  end function

Danach wird für alle Metadaten geprüft, ob die ID der zugehörigen Datei zu dem ID-Präfix passt und nicht leer ist. Wurde eine solche Datei gefunden, fordert S_{Restic} diese Datei über deren ID von S_{Backup} an.

Funktion 18 Get file from ID-Prefix

  function GETFILEFROMIDPREFIX($type$, $idPrefix$)
    SEND($type$) ▷ $S_{Restic} \longrightarrow S_{Backup}$
    $List_{Metadata}\ metadatas$:= RECEIVE() ▷ $S_{Restic} \longleftarrow S_{Backup}$
    for all m in $metadatas$ **do**
      if $idPrefix$ is prefix of $m.id$ **and** $m.size \neq 0$ **then**
        SEND($m.id, type$) ▷ $S_{Restic} \longrightarrow S_{Backup}$
        $Bytestring\ file$:= RECEIVE() ▷ $S_{Restic} \longleftarrow S_{Backup}$
        if ID($file[size_{IV} : size_{MAC}]$) == $m.id$ **then**
          return $file$
        else
          return $error$
        end if
      end if
    end for
    return $error$
  end function

A.1.4. Initialer Zugriff auf ein Repository

Die meisten Restic-Befehle beginnen damit, dass S_{Restic} auf ein Repository zugreift, das auf S_{Backup} liegt und damit seine interne Repository-Datenstruktur `repo` initialisiert.. Restic übernimmt den mit der *r* Option übergebenen Repository-Pfad oder liest ihn aus einer Textdatei ein, deren Pfad mit der *repository-file* Option übergeben wurde. Danach baut S_{Restic} eine Verbindung zu S_{Backup} auf. Diese Verbindung wird für den restlichen Restic-Befehl

zur Kommunikation zwischen S_{Restic} und S_{Backup} genutzt. S_{Restic} fordert die Config des Repositorys von S_{Backup} an. Ist die Größe der Config 0, kommt es zu einem Fehler.
Restic erstellt eine neue leere Repository Datenstruktur. Restic wartet auf die Passworteingabe des Benutzers. Das Passwort des Benutzers wird Benutzerpasswort genannt.
Wenn mit der *key-hint* Option ein Präfix einer Key-Datenstruktur-ID übergeben wurde, fordert S_{Restic} die Metadaten zu allen Key-Dateien des Repositorys von S_{Backup} an. Danach wird für alle Metadaten geprüft, ob die ID der zugehörigen Datei zu dem ID-Präfix passt und nicht leer ist. Wurde eine passende Key-Datei gefunden, fordert S_{Restic} die Key-Datei mit der ID von S_{Backup} an. Restic berechnet aus dem Benutzerpasswort zusammen mit den Parametern n, r und p, sowie dem $salt$ aus der Key-Datenstruktur den Benutzerkey. Der Benutzerkey ist eine Masterkey-Datenstruktur, mit der der verschlüsselte Repository-Masterkey aus der Key-Datenstruktur versucht wird zu verifizieren und zu entschlüsseln.. Restic versucht den verschlüsselten und authentifizierten Masterkey aus der Key-Datenstruktur mit dem Benutzerkey zu verifizieren.
Wurde keine passende ID zu dem ID-Präfix gefunden, der Masterkey nicht verifiziert werden oder wurde die *key-hint* Option nicht verwendet, fordert S_{Restic} ebenfalls die Metadaten aller Key-Dateien des Repositorys von S_{Backup} an. Restic führt folgendes für die ersten 20 Metadaten aus, die S_{Backup} zurücksendet:

1. S_{Restic} fordert die Key-Datei mit der ID der nächsten Key-Datei-Metadaten von S_{Backup} an.
2. S_{Backup} schickt S_{Restic} den Inhalt der angeforderten Key-Datei zurück.
3. Restic berechnet aus dem Benutzerpasswort mit den Parametern n, r und p, sowie dem $salt$ aus der Key-Datenstruktur den Benutzerkey.
4. Restic versucht den verschlüsselten und authentifizierten Masterkey aus der Key-Datenstruktur mit dem Benutzerkey zu verifizieren.
5. War die Verifikation nicht erfolgreich Beginne von vorne mit den nächsten Key-Datei-Metadaten.

Konnte eine Key-Datei mit dem Benutzerpasswort korrekt verifiziert werden, entschlüsselt Restic den Masterkey mit dem Benutzerpasswort und speichert ihn in der internen Repository-Datenstruktur. Die Masterkey-Datenstruktur wird in der internen Repository-Datenstruktur gespeichert.
S_{Restic} fordert die Config des Repositorys von S_{Backup} erneut an. Restic verifiziert die Config mit dem Masterkey und entschlüsselt sie daraufhin. Die Config-Datenstruktur wird in der internen Repository-Datenstruktur gespeichert.

A.1.4.1. Ein Lock für ein Repository erstellen

Mit der *connections* Option eines Befehls kann man den maximalen Wert für die Anzahl an parallelen Verbindungen zwischen S_{Restic} und S_{Backup} festlegen. Der Standardwert für die maximale Anzahl an Verbindungen ist 5 und für ein lokales Backupsystem 2. Als Wert für die maximale Anzahl an Verbindungen wird mit *CONNECTIONS* bezeichnet.

Funktion 19 Open a repository

```
Require: REPOSITORY_PATH
  function openRepository
    connect(REPOSITORY_PATH)
    send(TYPE_CONFIG)                                  ▷ S_Restic ⟶ S_Backup
    if size(receive()) == 0 then                       ▷ S_Restic ⟵ S_Backup
      return error
    end if
    Repository repo := newRepo()
    String userPassword := readUserPassword()
    if getMasterKey(userPassword) is error then
      return error
    end if
    send(TYPE_CONFIG)                                  ▷ S_Restic ⟶ S_Backup
    Config C := decrypt(receive())                     ▷ S_Restic ⟵ S_Backup
    if C is error then
      return error
    end if
    repo.config := C
  end function
```

Dieser Restic-Funktion bezieht sich immer auf das Repository, zu dem der Restic-Befehl, der diese Funktion nutzt, eine Verbindung mit Funktion 19 aufgebaut hat.
S_{Restic} fordert die Metadaten aller Lock-Dateien des Repositorys von S_{Backup} an. Restic startet so viele parallele Prozesse, wie der Wert von *CONNECTIONS* vorgibt. Jeder dieser Prozesse wird auf S_{Restic} ausgeführt. Auf die parallelen Prozesse werden nach und nach einzelne Lock-Datei-IDs verteilt, sowie die Information, ob das zu erstellende Lock ein exklusives Lock ist (`dispatch(m.id, isExclusive)`). Jeder Prozess, der die ID einer Lock-Datei erhält, startet seine Ausführung mit der Funktion 22.
Der Prozess fordert das Lock mit der ID von S_{Backup} an. S_{Backup} schickt den Inhalt der passenden Lock-Datei an S_{Restic} zurück. Restic verifiziert die erhaltene Lock-Datenstruktur und entschlüsselt und dekomprimiert sie anschließend. Konnte die Lock-Datenstruktur, die der Prozess angefordert hatte, nicht korrekt verifiziert, entschlüsselt oder anderweitig verarbeitet werden, wird dieses Lock ignoriert. Handelt es sich bei dem zu erstellenden Lock um ein exklusives Lock, reicht irgendein Lock im Repository aus, um die Erstellung des neuen Locks zu blockieren. Andernfalls prüfen die parallelen Prozesse nur, ob die Locks im Repository exklusiv sind und damit die Erstellung des neuen Locks blockieren.
Konnten keine blockierenden Locks gefunden werden, wird ein neues Lock L erstellt. S_{Restic} generiert eine zufällige Nonce IV und schickt $IV \,||\, Cipher^{IV}_{L_{comp}} \,||\, MAC^{IV}_{L_{comp}}$ an S_{Backup}. S_{Backup} speichert die empfangenen Daten als neue Lock-Datei in dem Repository.

Funktion 20 Get masterkey of repository

```
Require: KEY_HINT
  function getMasterKey(userPassword)
    if KEY_HINT is defined then
      Key key := getFileFromIdPrefix(TYPE_KEY, KEY_HINT)
      if key is not error then
        Masterkey userKey := getKeyFromUserPassword(userPassword, key)
        Masterkey K := decrypt(key.data)            ▷ decrypt benutzt userKey
        if K is not error then
          repo.masterkey := K
          return
        end if
      end if
    end if
    if repo.masterkey is empty then
      send(TYPE_KEY)                                ▷ S_Restic ⟶ S_Backup
      List_Metadata metadatas := receive()          ▷ S_Restic ⟵ S_Backup
      for i = 0; i < 20; i++ do
        send(metadatas[i].id, TYPE_KEY)             ▷ S_Restic ⟶ S_Backup
        Key key := receive()                        ▷ S_Restic ⟵ S_Backup
        if id(key) == metadatas[i].id then
          Masterkey userKey := getKeyFromUserPassword(userPassword, key)
          Masterkey K := decrypt(key.data)          ▷ decrypt benutzt userKey
          if K is not error then
            repo.masterkey := K
            return
          end if
        else
          return error
        end if
      end for
    end if
    return error
  end function
```

A.1.4.2. Laden eines bestimmten oder neusten Snapshots

Dieses Kapitel beschreibt die Funktion 23, mit der ein bestimmter Snapshot aus einem Repository geladen wird. Zum einen kann ein Präfix einer Snapshot-ID verwendet werden, um den Snapshot zu laden, dessen ID zu diesem Präfix gehört. Als alternative kann der letzte/neuste (latest) Snapshot geladen werden, der alle übergebenen Dateipfade oder Verzeichnispfade in seinen Target-Pfaden enthält.
Wenn der Funktion ein Präfix ($idPrefix$) einer Snapshot-ID übergeben wird, versucht S_{Restic} die zu dem ID-Präfix passende Datei von S_{Backup} anzufordern.

Funktion 21 Create and adds a lock to a repository on S_{Backup}

```
Require: CONNECTIONS
  function addLock(isExclusive)
    isRepositoryLocked := false
    send(TYPE_LOCK)                                   ▷ S_Restic ⟶ S_Backup
    List_Metadata metadatas := receive()              ▷ S_Restic ⟵ S_Backup
    startParallelProcesses(CONNECTIONS)
    for all m in metadatas do
      dispatch(m.id, isExclusive)
    end for
    waitBarrier()
    if isRepositoryLocked then
      return error
    end if
    Lock L̃_new := newLock(isExclusive)
    send(encrypt(compress(L̃_new)))                    ▷ S_Restic ⟵ S_Backup
  end function
```

Funktion 22 Start procedure for add lock

```
Require: Shared variable: isRepositoryLocked
  procedure start(id, isExclusive)
    send(id, TYPE_LOCK)                               ▷ S_Restic ⟶ S_Backup
    Bytestring encryptedLock := receive()             ▷ S_Restic ⟵ S_Backup
    if id(encryptedLock[size_IV : size_MAC]) == id then
      Lock L_i := decompress(decrypt(encryptedLock))
      if L_i is error then
        return
      end if
      if isExclusive then
        isRepositoryLocked := true
      end if
      if L_i.exclusive then
        isRepositoryLocked := true
      end if
    end if
    return error
  end procedure
```

Wird das *idPrefix* Argument nicht verwendet wird, wird eine Liste von Dateipfaden und Verzeichnispfaden erwartet (*includedPaths*). S_{Restic} fordert die Metadaten aller Snapshots des Repositorys von S_{Backup} an. Daraufhin startet Restic so viele parallele Prozesse, wie die *connections* Option angibt. Auf die parallelen Prozesse werden die IDs einzelner Snapshots, sowie alle Pfade, die der Snapshot enthalten soll, nach und nach verteilt mit `dispatch(m.id, includedPaths)`. Jeder Prozess, der die ID eines Snapshots des Repositorys erhält, startet

seine Ausführung mit der Funktion 24.
Der Prozess fordert den Snapshot, der zu der ID gehört, bei S_{Backup} an. S_{Backup} schickt den Inhalt der Snapshot-Datei für diese ID an S_{Restic} zurück. Restic verifiziert, entschlüsselt und dekomprimiert die erhaltene Snapshot-Datenstruktur.
Die parallelen Prozesse bestimmen den neusten Snapshot, der die übergebenen Dateipfade und Verzeichnispfade in seinen Target-Pfaden enthält.
Der ausgewählte Snapshot wird Parent-Snapshot genannt und wird in der internen Repository-Datenstruktur gespeichert.

Funktion 23 Get specific snapshot from S_{Backup}

```
procedure LOADSNAPSHOT(idPrefix, includedPaths)
    Snapshot SS_parent ← nil
    if idPrefix is not a prefix then
        SEND(TYPE_SNAPSHOT)                          ▷ S_Restic ⟶ S_Backup
        List_Metadata metadatas := RECEIVE()         ▷ S_Restic ⟵ S_Backup
        STARTPARALLELPROCESSES(CONNECTIONS)
        for all m in metadatas do
            dispatch(m.id, includedPaths)
        end for
        WAITBARRIER()
    else
        SS_parent := GETFILEFROMIDPREFIX(TYPE_SNAPSHOT, idPrefix)
        if SS_parent == error then
            return error
        end if
    end if
    repo.snapshot := SS_parent
end procedure
```

Funktion 24 Start procedure for getting latest snapshot

```
Require: Shared variable: SS_parent
procedure START(id, includedPaths)
    SEND(id, TYPE_SNAPSHOT)                          ▷ S_Restic ⟶ S_Backup
    Bytestring encryptedSnapshot := RECEIVE()        ▷ S_Restic ⟵ S_Backup
    if ID(encryptedSnapshot[size_IV : size_MAC]) == id then
        Snapshot SS_i := DECOMPRESS(DECRYPT(encryptedSnapshot))
        if SS_i.timestamp > selectedSnapshot.timestamp and includedPaths is in
SS_i.targets then
            SS_parent := SS_i
        end if
    end if
    return error
end procedure
```

A.1.4.3. Alle Indices des Repositorys laden

Restic erstellt eine Liste, in der alle Index-Datenstrukturen des Repositorys zwischengespeichert werden. S_{Restic} fordert die Metadaten aller Index-Dateien dieses Repositorys von S_{Backup} an.
Restic startet so viele parallele Prozesse, wie die *connections* Option angibt. Auf die parallelen Prozesse werden die IDs einzelner Indices nach und nach verteilt mit `dispatch(m.id)`. Jeder Prozess, der die ID eines Index des Repositorys erhält, startet seine Ausführung mit der Funktion 26.
Jeder parallele Prozess fordert den Index, der zu der ID gehört, bei S_{Backup} an. S_{Backup} schickt den Inhalt der Index-Datei für diese ID an S_{Restic} zurück. Restic verifiziert, entschlüsselt und dekomprimiert die erhaltene Index-Datenstruktur. Die parallelen Prozesse legen eine Liste mit allen geladenen Indices an.
Nachdem alle Index-Datenstrukturen in einer Liste zusammengefasst wurden, erstellt Restic eine große finale Index-Datenstruktur. Dieser finale Index wird Masterindex genannt. Sein *packs* Attribut entspricht der Konkatenation aller *packs* Attribute der gesammelten Indices (siehe auch Tabelle 2.17). Dieser Masterindex wird in der internen Repository-Datenstruktur gespeichert.

Funktion 25 Loads all indexes of the Repository

```
Require: CONNECTIONS
  procedure LOADINDEX
    List_Index allIndexes := {}
    SEND(TYPE_INDEX)                          ▷ S_Restic ⟶ S_Backup
    Metadata metadatas ← RECEIVE()            ▷ S_Restic ⟵ S_Backup
    STARTPARALLELPROCESSES(CONNECTIONS)
    for all m in metadatas do
      dispatch(m.id)
    end for
    WAITBARRIER()
    repo.masterindex := COMBINETOONEINDEX(allIndexes)
  end procedure
```

A.1.4.4. Hochladen einer Pack-Datenstruktur in ein Repository

Restic startet im Laufe der Befehlsabarbeitung parallele Prozesse, denen eine Pack-Datenstruktur übergeben werden kann, um sie in das Repository auf S_{Backup} hochzuladen. Diese parallelen Prozesse werden Pack-Uploader-Prozesse genannt. Die Funktion, die bei Übergabe einer Pack-Datenstruktur an einen Pack-Uploader-Prozess gestartet wird, ist Funktion 27.
Restic generiert einen Pack-Header für die Pack-Datenstruktur und verschlüsselt und authentifiziert den Pack-Header. Der verschlüsselte und authentifizierte Pack-Header, sowie dessen Größe, wird der Pack-Datenstruktur hinzugefügt. S_{Restic} schickt die Pack-Datenstruktur an S_{Backup}, welches dafür verantwortlich ist die Daten in eine Pack-Datei zu schreiben und

Funktion 26 Start procedure for load index

```
Require: Shared variable: allIndexes
  procedure START(id)
    SEND(id, TYPE_INDEX)                                  ▷ S_Restic ⟶ S_Backup
    Bytestring encryptedIndex := RECEIVE()                ▷ S_Restic ⟵ S_Backup
    if ID(encryptedIndex[size_IV : size_MAC]) == id then
      Index I_i := DECOMPRESS(DECRYPT(encryptedIndex))
      allIndexes := allIndexes||I_i
    end if
    return error
  end procedure
```

diese im Repository zu speichern.
Restic wählt die erste Index-Datenstruktur, die nicht als *final* markiert ist, aus einer Liste von neu erstellten Index-Datenstrukturen. Diese Liste `repo.waiting_indexes` ist zu Beginn der Ausführung eines Restic-Befehls noch leer. Konnte keine Index-Datenstruktur in dieser Liste gefunden werden wird eine neue Index-Datenstruktur erstellt und zu dieser Liste hinzugefügt. Restic speichert die ID der Pack-Datenstruktur und Informationen über die enthaltenen Blobs in der gewählten Index-Datenstruktur. Sind seit dem Erstellen dieser Index-Datenstruktur mehr als 10 Minuten vergangen oder enthält sie mehr als 50.000 Blobs wird sie als *final* markiert.
Danach führt Restic für jede als *final* markierte Index-Datenstruktur Folgendes aus. Restic komprimiert, verschlüsselt und authentifiziert die Index-Datenstruktur. S_{Restic} schickt den verschlüsselten und authentifizierten Index an S_{Backup}, das dafür verantwortlich ist die Daten in eine Index-Datei zu schreiben und diese im Repository zu speichern. Zum Schluss wird diese Index-Datenstrukturen aus der Liste alle neu erstellten Index-Datenstrukturen `repo.waiting_indexe` entfernt und der Inhalt des Indexes zu dem Masterindex der Repository-Datenstruktur hinzugefügt.

A.1.4.5. Laden eines Blobs

Restic durchsucht alle Index-Datenstrukturen der internen Repository-Datenstruktur nach der ID des zu ladenden Blobs. Dabei wird eine Liste alle Pack-IDs von Packs, die diesen Blob enthalten, erstellt. Restic itteriert über diese Liste von Pack-IDs und führt Folgendes solange aus, bis ein gültiger Blob gefunden wurde:

1. S_{Restic} fordert nur die Bytes, die zu dem Blob gehören, aus der Pack-Datei mit der zugehörigen ID von S_{Backup} an.
2. S_{Backup} schickt die Bytes an der übergebenen Position mit der übergebenen Länge an S_{Restic} zurück.
3. Restic verifiziert den erhaltenen Blob, entschlüsselt und dekomprimiert ihn.

Funktion 27 Uploads a pack to S_{Backup} and adds it to an index

```
procedure UPLOADPACK(P̃_i)
    P̃_i.header := ENCRYPT(CREATEPACKHEADER(P̃_i))
    P̃_i.header_length := size(P̃_i.header)
    SEND(P̃_i)                                              ▷ S_Restic ⟶ S_Backup
    Index index := GETFIRSTNONFINALINDEX(repo.waiting_indexes)
    if index is not defiened then
        index := NEWINDEX()
        repo.waiting_indexes := repo.waiting_indexes||index
    end if
    SAVEPACKINFOININDEX(P̃_i, index)
    if GETTIMESINCECREATION(index) > 10min or GETAMOUNTOFBLOBS(index) > 50000
then
        MARKASFINAL(index)
    end if
    for all Ĩ_i in repo.waiting_indexes do
        if ISFINAL(Ĩ_i) then
            SEND(ENCRYPT(COMPRESS(Ĩ_i)))                   ▷ S_Restic ⟶ S_Backup
            REMOVEFROMLIST(Ĩ_i, repo.waiting_indexes)
            repo.masterindex := COMBINEINDEXES(Ĩ_i, repo.masterindex)
        end if
    end for
end procedure
```

4. Ist ein Fehler beim Empfangen des Blobs von S_{Backup} aufgetreten, oder ist das Verifizieren, Entschlüsseln oder Dekomprimieren fehlgeschlagen, wird dieser Vorgang für die nächste Pack-ID der Liste wiederholt.
5. Wenn kein Fehler aufgetreten ist und ist die ID des Blobs korrekt, wird die restliche Liste verworfen und die Blob-Datenstruktur kann für die weitere Verarbeitung verwendet werden.

Konnte kein gültiger Blob gefunden werden, gibt diese Funktion einen Fehler zurück.

A.1.4.6. Laden mehrerer Blobs aus einem Pack

Das Laden mehrerer Blobs aus einem Pack erfolgt chunkweise. Dazu wird ein zusammenhängender Bytestring aus dem Pack angefordert, der mehrere der ausgewählten Blobs, sowie alle dazwischen liegenden Blobs, enthält. Dieser zusammenhängende Bytestring wird Blob-Chunk genannt und kann durch S_{Restic} von S_{Backup} angefordert werden.
Restic sortiert die Liste der zu ladenden Blob-IDs `blobIds` aufsteigend nach deren Offset in dem zugehörigen Pack. Es wird eine Liste angelegt, die einen Blob-Chunk repräsentiert. Restic iteriert der Reihenfolge nach über die Blob-IDs und fügt die IDs zu dem Blob-Chunk hinzu. Überschreitet die Größe des Blob-Chunks 32MiByte oder ist die Entfernung zum

Funktion 28 Load blob from Repository

```
function loadBlob(blobId)
    List_Hexstring[64] packIds := getPackIdsFromIndexes(blobId)
    for all packId in packIds do
        int blobOffset := getBlobOffsetInPack(packId, blobId)
        int blobLength := getBlobLengthInPack(packId, blobId)
        send(packId, blobOffset, blobLength)          ▷ S_Restic ⟶ S_Backup
        BB_i := decompress(decrypt(receive()))        ▷ S_Restic ⟵ S_Backup
        if BB_i is not error and id(BB_i) == blobId then
            return BB_i
        end if
    end for
    return error
end function
```

nächsten benötigten Blob in diesem Pack größer als 1MiByte, wird dieser Blob-Chunk mit der Funktion 30 geladen.
Für Funktion 30 berechnet Restic die Position und Größe dieses Blob-Chunks im Pack und schickt diese Daten an S_{Backup}. S_{Backup} schickt den zugehörigen Bytestring aus der Pack-Datei für die gesendete Pack-ID an S_{Restic} zurück. Restic verifiziert, entschlüsselt und dekomprimiert alle benötigten Blobs aus dem erhaltenen Blob-Chunk. Für jeden Blob aus der ursprünglich übergebenen Liste `blobIds` wird die Funktion `execute(...)` mit dem Blob als Argument ausgeführt. Bei dieser Funktion handelt es sich um einen Platzhalter, der in späteren Kapiteln spezifiziert wird, wenn die Funktion 29 verwendet wird.

A.1.4.7. Speichern eines Tree-Blobs oder Data-Blobs

Restic startet im Laufe der Befehlsabarbeitung parallele Prozesse, denen eine Blob-Datenstruktur übergeben werden kann, um sie in einem Pack zu speichern. Diese parallelen Prozesse werden Saver-Prozesse genannt. Die Funktion, die bei Übergabe einer Blob-Datenstruktur an einen Saver-Prozess gestartet wird, ist Funktion 31.
Der Funktion 31 wird entweder ein Tree-Blob oder ein Chunk einer Datei, auch Data-Blob genannt, übergeben. Restic überprüft, ob dieser Blob in den Indices des Repositorys oder den noch nicht gespeicherten Pack-Datenstrukturen vorhanden ist. Konnte Restic den Blob finden und wurde nicht explizit mit `storeDuplicates` angegeben, dass Restic den Blob erneut speichern soll, wird dieser Blob nicht erneut gespeichert.
Ansonsten wird der Blob komprimiert, verschlüsselt und authentifiziert. Die angestrebte Pack-Größe wird mit der Option *pack-size* an einen Restic-Befehl übergeben und wird hier *PACK_SIZE* genannt. Gilt $size(blob) \geq PACK\_SIZE$, dann erstellt `getNewPack()` eine neue leere Pack-Datenstruktur, zu der der verschlüsselte und authentifizierte Blob hinzugefügt wird. Außerdem wird die erstellte Pack-Datenstruktur direkt an einen der parallelen Pack-Uploader-Prozesse (siehe Funktion 27) verteilt mit `dispatch(pack)`.
Ist die Größe eines Blobs nicht größer als *PACK_SIZE* wählt Restic mit `getPendingPack()`

Funktion 29 Loads multiple blobs from a single pack

```
procedure loadBlobsFromPack(packId, blobIds)
    sortByOffsetInPack(blobIds)
    List_Hexstring[64] blobIdsInChunk := {}
    for all id in blobIds do
        if getOffset(id) − getEndOfChunk(blobIdsInChunk) > 1MiByte then
            loadBlobChunk(packId, blobIdsInChunk)
            blobIdsInChunk := {id}
        else
            blobIdsInChunk := blobIdsInChunk||id
            if getChunkSize(blobIdsInChunk) > 32MiByte then
                loadBlobChunk(packId, blobIdsInChunk)
                blobIdsInChunk := {}
            end if
        end if
    end for
    if size(blobIdsInChunk) > 0 then
        loadBlobChunk(packId, blobIdsInChunk)
    end if
end procedure
```

Funktion 30 Loads a Blob-Chunk and executes a given function for every selected blob

```
Require: execute(BB_i) function, which will be called on every blob
procedure loadBlobChunk(packId, blobIdsInChunk)
    chunkOffset := getChunkOffset(blobIdsInChunk)
    chunkLength := getChunkSize(blobIdsInChunk)
    send(packId, chunkOffset, chunkLength)              ▷ S_Restic ⟶ S_Backup
    Bytestring chunkOfBlobs := receive()                ▷ S_Restic ⟵ S_Backup
    for all blobId in blobIdsInChunk do
        BB_i := decompress(decrypt(extractBlob(blobId, chunkOfBlobs)))
        if id(BB_i) == blobId then
            execute(BB_i)
        else
            return error
        end if
    end for
end procedure
```

zufällig eine Pack-Datenstruktur aus, die noch nicht an einen Pack-Uploader-Prozess übergeben wurde und zu dem Typ des Blobs passt. Für einen Blob vom Typ Data-Blob wird nur ein Pack ausgewählt, das Data-Blobs speichert und für Blobs vom Typ Tree-Blobs wird nur ein Pack ausgewählt, das Tree-Blobs speichert. Restic fügt den verschlüsselten und authentifizierten Blob der *blobs* Liste des ausgewählten Packs hinzu.
Restic überprüft nach dem Hinzufügen eines Blobs, ob die Pack-Datenstruktur die angestrebte

Pack-Größe überschreitet oder ob der Header des Packs größer als 16MiByte ist. Ist dies der Fall, wird die Pack-Datenstruktur an einen der parallelen Pack-Uploader-Prozesse (siehe Funktion 27) verteilt mit `dispatch(pack)` und eine neues leere Pack wird für zukünftige Blobs des gleichen Typs erstellt.
Damit ist sichergestellt, dass es für Data-Blobs und Tree-Blobs immer die gleiche Anzahl freier Packs (pending Packs) gibt, auf die die Blobs zufällig verteilt werden können. In Restic gibt es standardmäßig immer zwei pending Packs für Data-Blobs und zwei pending Packs für Tree-Blobs.

Funktion 31 Saves a Tree-Blob or Data-Blob

Require: *PACK_SIZE*

```
procedure saveBlob(blob, storeDuplicates)
    if isPresentInRepository(blob) and not storeDuplicates then
        return
    end if
    Bytestring cipher := encrypt(compress(blob))
    if size(blob) ≥ PACK_SIZE then
        Pack P̃_i := getNewPack()
        P̃_i.blobs := P̃_i.blobs||cipher
        dispatch(P̃_i)
    else
        Pack P̃_i ←$ getPendingPack(type(blob))
        P̃_i.blobs := P̃_i.blobs||cipher
        if size(P̃_i) > PACK_SIZE or getHeaderSize(P̃_i) > 16MiByte then
            createNewPendingPack(type(blob))
            dispatch(P̃_i)
        end if
    end if
end procedure
```

B. Glossar

S_{Backup}

Ein System, auf dem das von Restic erstellte Repository gespeichert wird und für dieses System zu jeder Zeit einsehbar ist. Dieses System Kann das gleiche System wie S_{Restic} sein, ein anderer Computer oder ein anderer Server. Dieses System kann ein System sein, auf dem der Restic-Rest-Server installiert ist, aber auch ein System auf dem kein spezielles Restic-Backend installiert sein muss..

S_{Restic}

Ein Computersystem, auf dem das Programm Restic installiert ist und von einem Benutzer bedient wird..