Berner
Fachhochschule

Bachelor-Thesis

Digital und sicher: Neue Wege für die Schulkommunikation

Studiengang	Bachelor of Science in Computer Science
Autor	Nicolin Dora und Abidin Vejseli
Dozent	Prof. Dr. Philipp Locher
Experte	Martin Arnold

Version 1.0 vom 12. Juni 2025

Abstract

Die Digitalisierung eröffnet Schulen neue Möglichkeiten für Kommunikation und Verwaltung, steigert jedoch gleichzeitig die Anforderungen an den Schutz personenbezogener Daten von Kindern und Jugendlichen. Bestehende Systeme genügen diesen Anforderungen häufig nicht, sie vernachlässigen zentrale Datenschutzprinzipien, gewähren Plattformbetreibenden weitreichende Zugriffsmöglichkeiten und gefährden so die informationelle Selbstbestimmung der betroffenen Personen.

Ziel dieser Arbeit ist die Konzeption und Spezifikation einer Schulkommunikationslösung, die konsequent nach dem Prinzip «Privacy by Design» entwickelt wurde. Im Zentrum steht der Schutz sensibler Daten vor unbefugtem Zugriff, insbesondere durch den Plattformbetreibenden. Angestrebt wurden sichere 1:1- und Gruppenchats, die hohe Sicherheitsstandards mit benutzerfreundlicher Bedienung kombinieren, sowie eine klare, praktisch umsetzbare Spezifikation der Sicherheitsmechanismen.

Methodisch wurde ein durchgängig technischer Ansatz verfolgt, der Datenschutz von Beginn an strukturell verankert und Modularität, insbesondere im Hinblick auf den Prototyp, in den Vordergrund stellt. Dies erlaubt den flexiblen Austausch grundlegender kryptographischer Mechanismen. Ein S/MIME-inspirierter, dateibasierter Ansatz wurde aufgrund seiner besseren Erfüllung der Anforderungen an eine sichere Backup-Funktionalität und der einfacheren Handhabung im schulischen Kontext dem Double-Ratchet-Protokoll vorgezogen.

Das zentrale Ergebnis ist eine modulare, wiederverwendbare Spezifikation einer Schulkommunikationsplattform. Diese basiert auf einer dateibasierten, verschlüsselten Ablagestruktur, minimiert die zentrale Speicherung personenbezogener Daten und gewährleistet durchgängige Ende-zu-Ende-Verschlüsselung (E2EE) für alle Nachrichten. Die detaillierte Spezifikation umfasst kryptographische Protokolle, Datenstrukturen und Sicherheitsmechanismen, die in Pseudocode dargestellt werden. Ein begleitender Proof of Concept (PoC) demonstriert die technische Umsetzbarkeit der Kernkomponenten und der eingesetzten kryptographischen Primitive.

Die Arbeit zeigt, dass ein hohes Mass an Datenschutz und Datensouveränität im schulischen Kontext technisch realisierbar ist und liefert eine belastbare Grundlage für zukünftige Entwicklungen oder die Weiterentwicklung sicherer digitaler Kommunikationssysteme im Bildungsbereich.

Danksagung

Wir möchten uns herzlich bei allen bedanken, die uns während der Erstellung dieser Bachelorarbeit sowie im Verlauf unseres Studiums unterstützt haben.

Unser besonderer Dank gilt Prof. Dr. Philipp Locher für die kompetente Betreuung unserer Arbeit. Seine fachlichen Anregungen und die konstruktive Kritik haben wesentlich zur Qualität dieser Arbeit beigetragen.

Für die sorgfältige Durchsicht der Arbeit sowie wertvolle Hinweise auf technische, sprachliche und formale Aspekte danken wir Laura Sommer und David Bimmler.

Darüber hinaus danken wir Laura Sommer und Mona El Baqqali für ihre beständige Unterstützung und Ermutigung, nicht nur während der Anfertigung dieser Arbeit, sondern über das gesamte Studium hinweg.

Inhaltsverzeichnis

1. Einleitung

Dieses Kapitel beschreibt die Ausgangslage, gibt einen Überblick über bestehende Kommunikationslösungen für Schulen, formuliert Ziel und Zweck der Arbeit, grenzt den Untersuchungsrahmen ab und erläutert das Projektmanagement.

1.1. Ausgangslage

Die fortschreitende Digitalisierung bietet Schulen neue Möglichkeiten zur Optimierung von Kommunikations- und Verwaltungsprozessen. Digitale Lösungen tragen dazu bei, die schulische Kommunikation effizienter zu gestalten, indem sie administrative Aufgaben automatisieren und den Informationsaustausch zwischen Lehrpersonen, Eltern sowie Schülerinnen und Schülern erleichtern. Gleichzeitig müssen Datenschutz- und Sicherheitsaspekte berücksichtigt werden, insbesondere bei der Verarbeitung sensibler Daten.

Ein besonders aufschlussreiches Beispiel für die Relevanz dieses Themas stellt der Datenschutzvorfall bei der Schulkommunikationslösung *Stay Informed* im Jahr 2024 dar. Aufgrund eines Konfigurationsfehlers waren die Datenbanken der App unzureichend abgesichert, wodurch über einen längeren Zeitraum sensible Informationen wie Namen, Geburtsdaten, E-Mail-Adressen, Gesundheitsinformationen sowie interne Nachrichten und Dokumente ungeschützt im Internet einsehbar waren. Schätzungen zufolge waren mehr als 100'000 Kinder, Eltern und Erziehungsberechtigte an mehreren tausend Schulen betroffen [1]. Auch ein Blick über den Bildungsbereich hinaus verdeutlicht die Risiken ungenügend abgesicherter digitaler Infrastrukturen. So musste beispielsweise das digitale Impfbüchlein in der Schweiz infolge schwerer Datenschutzmängel eingestellt werden [2]. Diese Vorfälle zeigen, dass Datenschutzmassnahmen nicht als nachgelagerte Zusatzaufgabe verstanden werden dürfen, sondern von Beginn an ein integraler Bestandteil der Systemarchitektur sein müssen, insbesondere bei Anwendungen im Bildungsbereich, die Daten wie Noten, Abwesenheitsmeldungen oder persönliche Informationen im Austausch zwischen Schulmitarbeitenden und Eltern verarbeiten.

Im vorherigen Projekt [3] wurde die Schulkommunikationslösung Klapp hinsichtlich der Datenschutzanforderungen untersucht. Die Analyse zeigte, dass Klapp bereits einige Sicherheitsmechanismen implementiert hat, darunter die Speicherung von Daten ausschliesslich in der Schweiz, Transportverschlüsselung mittels *Transport Layer Security* (*TLS*), regelmässige Penetrationstests und eine logische Mandan-

tentrennung innerhalb der zentralen MongoDB-Datenbank. Dennoch wurde festgestellt, dass keine *End-to-End Encryption* (*E2EE*) für Nachrichten implementiert ist, was Risiken für die *Confidentiality* (Vertraulichkeit) der Kommunikation birgt. Der Verzicht auf *E2EE* wurde von Klapp damit begründet, dass die Plattform neben 1:1-Kommunikation auch Broadcast-Nachrichten an grosse Empfängergruppen unterstützen muss, was mit klassischen *E2EE*-Ansätzen schwer umzusetzen ist.

Vor diesem Hintergrund besteht die zentrale Herausforderung darin, Schulkommunikationslösungen so zu gestalten, dass sie Datenschutzanforderungen von Beginn an erfüllen und sich an den Prinzipien von *Privacy by Design* (*PbD*) orientieren [4, 5, 6].

1.2. Marktanalyse bestehender Schulkommunikationslösungen

Im Rahmen dieser Arbeit wurde geprüft, ob auf dem Markt bereits Schulkommunikationslösungen existieren, die eine umfassende Umsetzung des *PbD*-Ansatzes bieten und damit potenziell das Ziel dieser Arbeit obsolet machen könnten. Zu den betrachteten Anwendungen gehören unter anderem *Untis Messenger* [7], *SchoolFox* [8], *Sdui* [9], *Threema Education* [10], *schul.cloud* [11] und *IServ* [12].

Ein Vergleich dieser Lösungen zeigt, dass lediglich *Threema Education* und *schul.cloud* eine *E2EE* implementieren. Allerdings beschränkt sich *Threema Education* auf reine Messaging-Funktionen ohne schulspezifische Funktionen, während *schul.cloud* zwar *E2EE* unterstützt, jedoch keine klar strukturierte, erweiterbare Spezifikation aufweist, welche die Umsetzung datenschutzkonformer Abläufe systematisch adressiert [10, 11]. Die übrigen untersuchten Lösungen wie *Untis Messenger*, *SchoolFox*, *Sdui* und *IServ* verzichten vollständig auf *E2EE* und sichern Daten lediglich während der Übertragung oder durch serverseitige Schutzmechanismen [7, 8, 9, 12].

Darüber hinaus zeigt die Analyse, dass das Prinzip *PbD* in keiner der betrachteten Lösungen konsequent umgesetzt wurde. Zwar dokumentieren viele Anbieter die Einhaltung datenschutzrechtlicher Rahmenbedingungen wie der DSGVO sowie den Einsatz moderner Verschlüsselungsprotokolle für Datenübertragungen, jedoch fehlt bei allen Anwendungen eine durchgängige Architektur, die auf Prinzipien wie Datenminimierung, Benutzerkontrolle, Trennung von Datenflüssen und robuste *E2EE* ausgerichtet ist. Datenschutz wird oftmals als ergänzender Aspekt behandelt, jedoch nicht als integraler Bestandteil des Systemdesigns.

Die Marktanalyse zeigt, dass aktuell keine Lösung existiert, die sowohl dem Prinzip von *PbD* entspricht als auch die spezifischen funktionalen Anforderungen einer ganzheitlichen Schulkommunikationslösung erfüllt. Daher besteht weiterhin ein Bedarf an einer neu konzipierten, datenschutzkonformen und funktional vollständigen Anwendung.

1.3. Ziel und Zweck der Arbeit

Die vorliegende Bachelorarbeit verfolgt das Ziel, eine Schulkommunikationslösung zu spezifizieren, die den Prinzipien von *PbD* konsequent folgt. Ausgehend von der Analyse der bestehenden Lösung Klapp und deren identifizierten Datenschutzdefiziten soll eine neue, sicherheitsorientierte Plattform konzipiert und spezifiziert werden. Die zentrale Herausforderung liegt darin, datenschutzkonforme Kommunikation nicht nur für den 1:1-Austausch, sondern auch für die Gruppenkommunikation praktikabel umzusetzen, ohne Kompromisse bei Benutzerfreundlichkeit oder Funktionalität einzugehen.

Grundlage der Arbeit ist die konzeptionelle Entwicklung zweier Varianten einer datenschutzfreundlichen Schulkommunikationslösung, die hinsichtlich Datenschutzanforderungen, technischer Umsetzbarkeit und Benutzerfreundlichkeit bewertet werden. Auf Basis dieser Analyse wird eine Lösung ausgewählt und detailliert spezifiziert, wobei der Schwerpunkt auf sicherheitsrelevanten Aspekten wie Datenspeicherung, Schlüsselaustausch (*Bootstrapping*) und der sicheren Umsetzung von 1:1- und Gruppenkommunikation liegt. Ergänzend zeigt ein *Proof of Concept* (*PoC*), wie zentrale Funktionalitäten praktisch umgesetzt werden können.

Der methodische Schwerpunkt liegt in der Spezifikation eines kryptographischen Protokolls für eine Schulkommunikationslösung. Dieses wird in Form von Algorithmen in Pseudocode dargestellt, die detailliert genug sind, um eine Implementierung in einer modernen Programmiersprache zu ermöglichen. Dabei werden kryptographische Bibliotheken ausschliesslich für Standardprimitive benötigt, etwa symmetrische und asymmetrische Verschlüsselung, Signaturen, Hashfunktionen, Pseudozufallszahlengeneratoren und Schlüsselableitungsfunktionen. Die Spezifikation schlägt damit eine praxisorientierte Brücke zwischen Theorie und Implementierung.

Das Ergebnis ist eine wiederverwendbare, modulare Spezifikation, die als Grundlage für zukünftige Entwicklungsprojekte dienen kann, etwa zur Neuentwicklung datenschutzkonformer Kommunikationssysteme oder zur Weiterentwicklung bestehender Lösungen wie Klapp. Der Nutzen der Arbeit liegt insbesondere für Softwareentwicklerinnen und -entwickler, IT-Sicherheitsverantwortliche im Bildungsbereich sowie Anbieter von Schulkommunikationslösungen in der praktischen Anwendbarkeit der konzipierten Lösung. Damit versteht sich die Arbeit als Beitrag zur Verbesserung der digitalen Bildungsinfrastruktur unter besonderer Berücksichtigung datenschutzrechtlicher Anforderungen.

1.4. Abgrenzung

Die Arbeit ist als konzeptionelle Entwicklungsstudie mit sicherheitstechnischem Schwerpunkt konzipiert. Im Mittelpunkt steht die Spezifikation einer datenschutz-

freundlichen Schulkommunikationslösung, die dem Prinzip *PbD* folgt und insbesondere eine sichere 1:1- sowie Gruppenkommunikation durch den Einsatz moderner kryptographischer Verfahren ermöglicht. Die Betrachtung konzentriert sich dabei ausschliesslich auf die zugrunde liegenden kryptographischen Protokollkomponenten.

Nicht Bestandteil der Arbeit sind weiterführende Funktionalitäten wie Stundenpläne, Aufgabenverwaltung, Lernplattform-Integrationen oder die Verarbeitung schulischer Leistungsdaten. Ebenso werden keine grafischen Benutzeroberflächen, Designvorschläge oder Mockups zur App-Oberfläche entwickelt. Auch der begleitende *PoC* beschränkt sich auf die technische Demonstration zentraler Sicherheitsfunktionen. Ziel ist es, die praktische Umsetzbarkeit der konzipierten Mechanismen zu veranschaulichen. Eine produktionsreife Implementierung oder eine anwendungsfertige Benutzeroberfläche wird dagegen nicht angestrebt. Das bedeutet, dass Benutzerfreundlichkeit in dieser Arbeit nicht als Eigenschaft einer ansprechenden oder praktischen grafischen Oberfläche verstanden wird. Vielmehr bezieht sich der Begriff darauf, dass die eingesetzte Kryptographie für Endnutzende weitgehend unsichtbar bleibt und ihre Nutzungserfahrung nicht beeinträchtigt.

Eine empirische Evaluation im produktiven Umfeld sowie Nutzerstudien erfolgen nicht. Die Validierung erfolgt ausschliesslich in Form eines prototypischen Nachweises im Sinne eines technischen Machbarkeitsbelegs.

Geografisch liegt der Fokus auf dem Schweizer Bildungsraum, wobei die rechtlichen Rahmenbedingungen und Datenschutzstandards der Schweiz berücksichtigt werden. Internationale Datenschutzregelungen wie die DSGVO werden lediglich dann einbezogen, wenn sie für Vergleichsbetrachtungen relevant sind, bilden aber keinen Schwerpunkt.

Diese Abgrenzung gewährleistet eine klare thematische Fokussierung und ermöglicht eine vertiefte Auseinandersetzung mit den sicherheitsrelevanten Aspekten der Plattformkonzeption.

1.5. Projektmanagement

Die Umsetzung der Arbeit erfolgte anhand eines hybriden Vorgehensmodells. Der Projektverlauf wurde durch einen vordefinierten Zeitplan strukturiert (vgl. Anhang A.0.1), während regelmässige Iterationen Raum für methodische Anpassungen liessen. Dadurch konnten sowohl planbare Meilensteine als auch flexible Entscheidungen berücksichtigt werden.

Ein zentrales Element der Arbeitsorganisation war der Einsatz von *User Stories*, welche der internen Koordination und Aufgabenverteilung dienten (vgl. Anhang A.0.3). Die Arbeitspakete wurden anhand dieser *User Stories* strukturiert und ermöglichten eine effiziente asynchrone Zusammenarbeit.

Ergänzend wurde ein strukturiertes Risikomanagement durchgeführt, um potenzielle Projektrisiken frühzeitig zu identifizieren und geeignete Gegenmassnahmen zu definieren (vgl. Anhang A.0.2).

Alle zwei Wochen fanden begleitende Besprechungen mit dem betreuenden Dozenten statt. Diese dienten der Fortschrittskontrolle, dem fachlichen Austausch sowie der methodischen Feinjustierung. In diesen Sitzungen wurden:

- der aktuelle Arbeits- und Forschungsstand präsentiert,
- methodische und technische Entscheidungen reflektiert,
- inhaltliche Fragen und Herausforderungen adressiert.

Durch diese regelmässigen Iterationen konnte sowohl die Qualität der Spezifikation als auch die Zielorientierung der Umsetzung gewährleistet werden.

Die Sitzungen wurden jeweils in kompakter Form als Markdown-Dokumente protokolliert und im entsprechenden *Repository* auf dem *GitLab* der BFH abgelegt. Aus Gründen der Übersichtlichkeit wird auf die Aufnahme sämtlicher Protokolle im Anhang verzichtet. Auf Anfrage können jedoch gerne Leserechte auf das Repository gewährt werden.

Fachbegriffe, insbesondere Anglizismen, die nicht allgemein gebräuchlich sind, werden in *kursiver Schrift* gesetzt. Damit wird ihre Funktion als eingeführte, fremdsprachliche Begriffe im technischen Kontext kenntlich gemacht. Ausgewählte Begriffe werden zusätzlich im Glossar erläutert. Symbole und Bezeichner, die auch in Protokollschritten oder Pseudocode-Algorithmen verwendet werden (z. B. uid, usk, PF), erscheinen in mathematischer Schrift mit `\ensuremath{}`, um ihre formale Rolle hervorzuheben.

1.6. Übersicht

Diese Arbeit gliedert sich in sieben Hauptkapitel, die im Folgenden kurz vorgestellt werden, um einen klaren Überblick über den Aufbau der Arbeit zu geben:

- **Kapitel 1: Einleitung** – Dieses Kapitel beschreibt die Motivation und Ausgangslage, formuliert die zentralen Ziele sowie den Zweck dieser Arbeit und gibt einen Überblick über den weiteren Aufbau.
- **Kapitel 2: Grundlagen** – In diesem Kapitel werden die für das Verständnis der Arbeit zentralen Begriffe und Konzepte eingeführt. Die Definitionen schaffen eine theoretische Basis, auf der die weiteren Kapitel aufbauen.
- **Kapitel 3: Variantenentscheidung** – In diesem Kapitel werden zwei Lösungsvarianten konzeptionell gegenübergestellt und bewertet.

- **Kapitel 4: Beschreibung der Plattform** – Dieses Kapitel erläutert die Architektur und Funktionsweise der konzipierten Plattform. Es beschreibt die beteiligten Rollen, den Plattformaufbau, den Nachrichtenaustausch sowie datenschutzrelevante Mechanismen auf konzeptioneller Ebene.
- **Kapitel 5: Protokoll Spezifikation** – Es folgt eine detaillierte Spezifikation des zugrunde liegenden Kommunikationsprotokolls. Die Kapitelinhalte umfassen kryptographische Primitive, Datenstrukturen, Bedrohungsmodellierung und Algorithmen in Pseudocode.
- **Kapitel 6: Proof of Concept** – Aufbauend auf der Spezifikation wird ein technischer Machbarkeitsnachweis in Form eines *Proof of Concept* präsentiert. Die Umsetzung demonstriert zentrale Aspekte der konzipierten Plattform und bildet die Grundlage für die nachfolgende Diskussion.
- **Kapitel 7: Diskussion und Ausblick** – Abschliessend reflektiert dieses Kapitel die erzielten Ergebnisse, diskutiert Limitationen der aktuellen Lösung und skizziert mögliche Erweiterungen sowie offene Forschungsfragen. Es zeigt auf, wie die entwickelte Spezifikation in zukünftigen Projekten weiterverwendet oder adaptiert werden kann.

2. Grundlagen

Dieses Kapitel führt die für die Arbeit zentralen Begriffe und Konzepte ein, die das Verständnis der weiteren Ausführungen erleichtern. Darauf aufbauend werden die wichtigsten Ergebnisse der vorangegangenen Projektarbeit zusammengefasst und die daraus abgeleiteten Anforderungen an die zu entwickelnde Kommunikationsplattform formuliert.

2.1. Begriffsdefinitionen

In diesem Abschnitt werden die für die Arbeit relevanten Fachbegriffe präzisiert. Falls unterschiedliche Definitionen existieren, werden diese gegenübergestellt und es wird eine Arbeitsdefinition festgelegt. Die Begriffe Verschlüsselung (2.1.3) und Signatur (2.1.4) werden an dieser Stelle allgemein verständlich eingeführt und in den Kapiteln 5.1.5 sowie 5.1.7 im Detail erläutert.

2.1.1. Privacy by Design – Prinzipien und rechtliche Grundlagen

Privacy by Design (*PbD*) ist ein zentraler Ansatz zur systematischen und proaktiven Gewährleistung des Datenschutzes in der digitalen Welt. Das Konzept wurde von Ann Cavoukian, der ehemaligen Informations- und Datenschutzbeauftragten von Ontario (Kanada), entwickelt. *PbD* ist eng mit dem Konzept der *Privacy-Enhancing Technologies (PET)* verbunden, das erstmals 1995 in dem gemeinsamen Bericht *Privacy-Enhancing Technologies: The Path to Anonymity* [13] von der niederländischen Datenschutzbehörde und Cavoukian vorgestellt wurde. Dabei standen insbesondere Technologien zur Minimierung der Datensammlung und zur Gewährleistung anonymisierter Interaktionen in digitalen Systemen im Fokus.

Peter Hustinx betont in seinem Artikel *Privacy by Design: Delivering the Promises* [14] aus dem Jahr 2010, dass *PbD* auf den Prinzipien von PET aufbaut. Der Geltungsbereich von *PbD* geht jedoch über die rein technologische Ebene hinaus und umfasst auch organisatorische sowie prozessuale Massnahmen, um den Datenschutz standardmässig als integralen Bestandteil zu gewährleisten. Dieses Kapitel beleuchtet die Grundprinzipien von *PbD* und ihre Bedeutung im Kontext moderner Datenschutzrahmenwerke.

Die sieben Grundprinzipien von Privacy by Design

Die sieben Grundprinzipien von *PbD* bilden eine systematische Grundlage, um Datenschutz frühzeitig und durchgehend in Systeme und Prozesse zu integrieren [15]:

1. **Proaktiv, nicht reaktiv; präventiv, nicht kurativ** *PbD* setzt auf präventive Massnahmen, um Datenschutzverletzungen im Voraus zu verhindern, anstatt erst nach deren Auftreten Lösungen zu suchen.
2. **Datenschutz als Standardeinstellung** *PbD* stellt sicher, dass persönliche Daten standardmässig geschützt sind. Nutzende müssen keine zusätzlichen Schritte unternehmen, um ihre Privatsphäre zu wahren.
3. **Datenschutz integriert im Design** Datenschutz wird von Beginn an in IT-Systeme und Geschäftsprozesse eingebettet, statt ihn nachträglich hinzuzufügen. Dadurch wird Datenschutz ein fester Bestandteil der Systemarchitektur.
4. **Volle Funktionalität – Kein Nullsummenspiel** *PbD* fördert Lösungen, die Datenschutz und andere legitime Ziele wie Sicherheit und Effizienz gleichermassen berücksichtigen, ohne unnötige Kompromisse einzugehen.
5. **Ende-zu-Ende-Sicherheit** *PbD* gewährleistet den Schutz von Daten über deren gesamten Lebenszyklus hinweg – von der Erhebung bis zur sicheren Löschung.
6. **Sichtbarkeit und Transparenz** *PbD* schafft Nachvollziehbarkeit und Transparenz, sodass Datenschutzprozesse den zugesicherten Anforderungen entsprechen und von unabhängigen Stellen überprüft werden können.
7. **Respekt für die Privatsphäre der Nutzenden** *PbD* legt besonderen Wert auf nutzerfreundliche und datenschutzfreundliche Voreinstellungen, klare Kommunikation und transparente Auswahlmöglichkeiten für die Nutzenden.

Privacy by Default

Im Zusammenhang mit *PbD* wird oft auch von *Privacy by Default* gesprochen. *Privacy by Default* bedeutet, dass Systeme und Dienste standardmässig den höchstmöglichen Datenschutz bieten, ohne dass die Nutzenden aktiv eingreifen müssen. Durch **Prinzip 2: Datenschutz als Standardeinstellung** und **Prinzip 7: Respekt für die Privatsphäre der Nutzenden** wird *Privacy by Default* von Cavoukian als Bestandteil von *PbD* gesehen. Häufig werden die beiden Ansätze jedoch getrennt betrachtet.

Rechtliche Definition in der Schweiz

In der Schweiz sind die Grundsätze *PbD* und *Privacy by Default* in Artikel 7 des Datenschutzgesetz (DSG) verankert. Das Gesetz verlangt, dass technische und organisatorische Massnahmen (TOM) den aktuellen Stand der Technik berücksichtigen und

darauf abzielen, die Risiken für die Persönlichkeits- und Grundrechte der betroffenen Personen zu minimieren [16, 17].

Das DSG gibt keine detaillierten Vorgaben zu den TOM, diese werden jedoch im Leitfaden des Eidgenössischen Datenschutz- und Öffentlichkeitsbeauftragten (EDÖB) präzisiert. Beispielsweise wird in Kapitel 8.2 des Leitfadens empfohlen, gespeicherte Daten (*at rest*) zu verschlüsseln, während Kapitel 9.2 beschreibt, wie die Kommunikation zwischen Kommunikationspartnern verschlüsselt werden sollte, um die *Confidentiality* zu gewährleisten [18].

Arbeitsdefinition

In dieser Arbeit wird *PbD* als Ansatz verstanden, der Datenschutz von Beginn an systematisch in die Entwicklung und den Betrieb von Anwendungen integriert. Dies beinhaltet insbesondere die Umsetzung technischer und organisatorischer Massnahmen, die dem aktuellen Stand der Technik entsprechen und gängigen Standards folgen, um sensible Daten wirksam zu schützen.

2.1.2. End-to-End Encryption

End-to-End Encryption (*E2EE*) bedeutet, dass eine Nachricht ausschliesslich von den beteiligten Kommunikationspartnern gelesen werden kann. Möchte Alice beispielsweise eine Nachricht an Bob senden, verschlüsselt ihr Gerät die Nachricht sofort, und erst Bobs Gerät kann sie wieder entschlüsseln. Der Plattformanbieter, etwa WhatsApp, leitet die Nachricht zwar weiter, hat aber zu keinem Zeitpunkt Einsicht in deren Inhalt (siehe Abbildung 2.1).

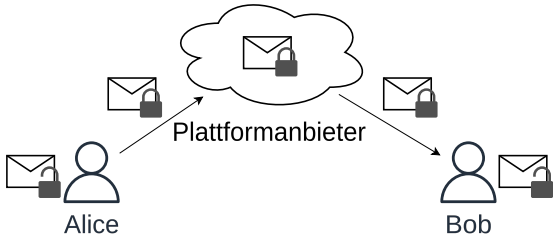


Abbildung 2.1.: End-to-End Encryption: Nur Alice und Bob sehen den Klartext.

Damit gewährleistet *E2EE Confidentiality*, *Integrity* und *Authenticity* der Kommunikation und ist ein zentrales Sicherheitsprinzip in Anwendungen wie Messenger-Diensten oder Cloud-Speicherlösungen [19, 20].

Abgrenzung zur Transportverschlüsselung

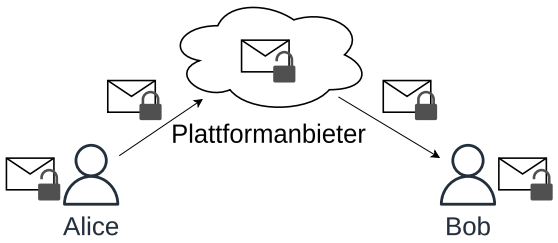


Abbildung 2.2.: Transportverschlüsselung: Plattformanbieter sieht den Klartext.

Transportverschlüsselung sichert lediglich die Verbindung zwischen Client und Server, beispielsweise mit *TLS*. Auf dem Server liegen die Daten im Klartext vor und können dort vom Plattformanbieter eingesehen werden (siehe Abbildung 2.2).

Im Gegensatz dazu schützt *E2EE* den gesamten Kommunikationsweg einschliesslich der Server. Nur die Endgeräte von Alice und Bob haben Zugriff auf den Klartext [19, 21].

2.1.3. Verschlüsselung

Verschlüsselung bedeutet, eine lesbare Information mit Hilfe eines Schlüssels so zu verändern, dass sie für Aussenstehende unverständlich wird. Erst mit einem passenden Schlüssel kann die ursprüngliche Information wieder lesbar gemacht werden. Dadurch lässt sich sicherstellen, dass nur berechtigte Parteien bestimmte Nachrichten oder Dateien lesen können.

Grundsätzlich gibt es zwei Arten von Verschlüsselung:

Bei der symmetrischen Verschlüsselung verwenden die sendende sowie die empfangende Partei denselben Schlüssel. Man kann sich das vorstellen wie bei einer verschlossenen Truhe, zu der beide Parteien denselben Schlüssel besitzen. Wer diesen Schlüssel hat, kann die Truhe öffnen, den Inhalt lesen oder etwas hinzufügen und sie wieder verschliessen [22, Kap. 3].

Bei der asymmetrischen Verschlüsselung besitzt jede Partei ein eigenes Schlüsselpaar mit einem öffentlichen und einem geheimen Schlüssel. Den öffentlichen Schlüssel kann man sich wie ein offenes Schloss vorstellen, das allen zur Verfügung gestellt wird. Wer einer bestimmten Partei eine Nachricht senden möchte, verschliesst sie mit diesem offenen Schloss. Nur die Partei, die im Besitz des passenden geheimen Schlüssels ist, kann das Schloss wieder öffnen. So wird eine sichere Kommunikation ermöglicht, ohne dass zuvor ein geheimer Schlüssel ausgetauscht werden muss. Der sendenden Partei muss lediglich das offene Schloss der empfangenden Partei bekannt sein [22, Kap. 12].

2.1.4. Signatur

Eine digitale Signatur ist vergleichbar mit einem persönlichen Siegel. Sie zeigt, wer die Information erstellt hat und stellt sicher, dass diese Information seitdem nicht verändert wurde.

Man kann sich das Prinzip anhand eines Beispiels aus dem Mittelalter veranschaulichen. Wenn eine Person einen wichtigen Brief verschickte, wurde dieser mit einem persönlichen Wachssiegel versehen. Ein solches Siegel konnte ausschliesslich von der absendenden Person angebracht werden, da nur sie im Besitz des entsprechenden Siegelwerkzeugs war. Traf der Brief beim Empfänger ein und war das Siegel unversehrt, so bestand hinreichende Sicherheit darüber, dass der Brief authentisch war, nicht verändert wurde und tatsächlich vom angegebenen Absender stammte.

Ein vergleichbares Verfahren kommt bei digitalen Signaturen zum Einsatz. Eine Partei erzeugt mithilfe ihres geheimen Schlüssels eine Signatur über eine bestimmte Information. Jede andere Partei kann mithilfe des zugehörigen öffentlichen Schlüssels überprüfen, ob diese Signatur gültig ist [22, Kap. 13].

2.2. Vorarbeit

Die vorliegende Arbeit baut auf den Erkenntnissen und Ergebnissen der Projektarbeit „Privacy by Design in Schulkommunikations-Apps“ [3] auf. In dieser Vorarbeit wurde eine Schulkommunikationsplattform[1] untersucht und theoretische Grundlagen für die Konzeption einer datenschutzfreundlichen Lösung erarbeitet. Ziel war es, ein fundiertes Verständnis für die Herausforderungen und Anforderungen im Bereich der Schulkommunikation zu entwickeln. Im Folgenden werden die Kernergebnisse dieser Vorarbeit zusammengefasst.

2.2.1. Analyse bestehender Lösungen und kryptographischer Protokolle

Ein wesentlicher Bestandteil der Projektarbeit war die Analyse marktüblicher Messenger-Dienste und einer Schulkommunikationslösung hinsichtlich ihrer Datenschutz- und Sicherheitsmechanismen. Dabei wurden unter anderem die Protokollfamilien *Double Ratchet* (*DR*) und *Secure/Multipurpose Internet Mail Extensions* (*S/MIME*) evaluiert. Diese Untersuchung zeigte klar ausgeprägte Stärken und Schwächen beider Ansätze in Bezug auf *E2EE*, Schlüsselmanagement, Gruppenkommunikationsfähigkeit und den damit verbundenen Wartungsaufwand. Es wurde festgestellt, dass beide Protokolle prinzipiell für den schulischen Kontext adaptierbar sind, jedoch unterschiedliche Implikationen für *PbD* und Benutzerfreundlichkeit mit sich bringen. Diese Differenzen und deren Relevanz für die Systemkonzeption werden in der Variantenentscheidung (Kapitel 3) im Detail diskutiert.

[1] Bei der Plattform handelt es sich um Klapp, vgl. `https://www.klapp.pro`

2.2.2. Bootstrapping und Schlüsselauthentizität

Die sichere initiale Verteilung und Authentifizierung von kryptographischen Schlüsseln stellt eine zentrale Herausforderung für *E2EE*-Systeme dar. Die Projektarbeit [3] untersuchte verschiedene Ansätze und kam zu dem Ergebnis, dass ein papiergestütztes *Bootstrapping*-Verfahren, beispielsweise mittels QR-Codes auf physischen Einladungsschreiben, eine praktikable und für den Schulkontext geeignete Lösung darstellt. Er ermöglicht eine asynchrone Registrierung und reduziert technische Hürden für die Nutzenden, da kein manueller Abgleich von kryptographischen Fingerprints erforderlich ist, wie er bei einigen *DR*-basierten Systemen üblich ist.

2.2.3. Daten-, Geräte- und Wiederherstellungsmanagement

Die Projektarbeit verdeutlichte die Notwendigkeit eines robusten Managements von Benutzerdaten sowie die Unterstützung mehrerer Endgeräte pro Nutzenden und damit verbunden einer zentralen Ablage relevanter Informationen. Für die Chatfunktionalität wurde eine Backup-Möglichkeit als essenziell identifiziert, damit Nachrichtenverläufe bei einem Gerätewechsel oder -verlust ohne Beeinträchtigung der *E2EE* wiederhergestellt werden können. Zudem wurde untersucht, wie bestehende Messenger-Dienste Backups umsetzen (z. B. clientseitig verschlüsselte Cloud-Backups) und welche Kompromisse dabei zwischen Sicherheit und Benutzerakzeptanz eingegangen werden.

2.2.4. Plattformstrategie und Implikationen für diese Arbeit

Die Projektarbeit legte nahe, dass ein hybrider Plattformansatz, der sowohl mobile Applikationen als auch einen Web-Client umfasst, die Zugänglichkeit für alle Zielgruppen (Lehrpersonen, Eltern, Schülerinnen und Schüler mit unterschiedlicher technischer Ausstattung) erhöht, ohne funktionale Einschränkungen hinnehmen zu müssen.

2.3. Anforderungen an die Kommunikationsplattform

Die zentralen Ergebnisse aus der Projektarbeit (siehe Abschnitt 2.2) bilden die Grundlage für die Definition der Anforderungen an die Schulkommunikationslösung, die in dieser Arbeit spezifiziert wird. Diese Anforderungen sind massgeblich für die nachfolgende Variantenentscheidung (Kapitel 3) sowie für das Design und die Spezifikation der Plattform (Kapitel 4 und 5). Sie sind technologieneutral formuliert, um eine frühzeitige Festlegung auf spezifische kryptographische Protokolle oder Implementierungsdetails zu vermeiden. Es wird zwischen funktionalen (F) und nichtfunktionalen (NF) Anforderungen unterschieden.

2.3.1. Funktionale Anforderungen

Funktionale Anforderungen definieren die spezifischen Operationen und Fähigkeiten, die die Plattform bereitstellen muss:

F1 **Kommunikation unter End-to-End Encryption:** Jegliche 1:1-, Gruppen- und Broadcast-Nachrichten müssen unter *E2EE* versendet werden.

F2 **Authentifizierter Schlüsselaustausch:** Der initiale Austausch von öffentlichen Schlüsseln zwischen den Kommunikationspartnern muss auf eine Weise erfolgen, die deren *Authenticity* sicherstellt.

F3 **Rollen- und Berechtigungsmodell:** Die Plattform muss klar definierte Rollen (z. B. Admin, Lehrperson, Bezugsperson, Schülerin/Schüler) mit jeweils feingranularen Zugriffsrechten auf Funktionen und Daten abbilden.

F4 **Nutzenden- und Geräteverwaltung:** Für jeden Nutzenden muss der parallele Einsatz mehrerer Endgeräte ermöglicht werden. Ein Gerätewechsel oder -verlust muss ohne permanenten Datenverlust und ohne Kompromittierung der Sicherheit handhabbar sein.

F5 **Zentrale Datenablage:** Alle Daten rund um den Schulalltag, einschliesslich Nachrichtenverläufen, müssen zentral gespeichert und durchgehend mit *E2EE* geschützt werden.

F6 **Sicheres Schlüsselmanagement:** Private Schlüssel der Benutzenden müssen verschlüsselt und signiert abgelegt werden, sodass ausschliesslich der jeweilige Schlüsselinhaber Zugriff darauf hat.

F7 **Plattformzugang (Web & Desktop/Mobile):** Die Lösung muss sowohl im Browser als auch als dedizierte Desktop- oder Mobile-Applikation nutzbar sein, um unterschiedliche Nutzungsszenarien und Präferenzen abzudecken.

F8 **Asynchrone Kommunikation:** Die Plattform muss asynchrone Kommunikation unterstützen, sodass Nachrichten auch dann zugestellt und später gelesen werden können, wenn einzelne Teilnehmende temporär offline sind.

2.3.2. Nicht-funktionale Anforderungen

Nicht-funktionale Anforderungen beschreiben Qualitätsmerkmale und Randbedingungen, die das System erfüllen muss:

NF1 **Privacy by Design:** Sämtliche Prinzipien von *PbD* sind im gesamten Lösungsdesign nachweisbar und konsequent umzusetzen. Die Plattform darf keine personenbezogenen Daten in zentral einsehbarer Form speichern.

NF2 **Sicherheit:** Die Plattform muss die grundlegenden Sicherheitsziele *Confidentiality*, *Integrity* und *Authenticity* für alle Kommunikations- und Speicheraspekte gewährleisten.

NF3 **Benutzerfreundlichkeit und Barrierefreiheit:** Grundfunktionen der Plattform (wie Registrierung, Nachrichtenversand sowie Nachrichten- und Datenspeicherung) sind so umzusetzen, dass sie ohne vertiefte technische Kenntnisse einfach und intuitiv nutzbar sind.

3. Variantenentscheidung

Im Rahmen dieser Arbeit erfolgte eine sorgfältige Evaluation zweier konzeptioneller Ansätze zur Realisierung einer datenschutzfreundlichen Schulkommunikationslösung. Ziel des Evaluationsprozesses war es, die Vor- und Nachteile der Varianten systematisch zu analysieren, um eine fundierte Entscheidung für eine geeignete und detailliert zu spezifizierende Lösung zu treffen.

Basierend auf den Vorarbeiten aus der Projektarbeit [3] und den in Abschnitt 2.3 definierten Anforderungen wurden zwei zentrale Varianten in Betracht gezogen. Ein Ansatz, der auf dem etablierten *DR*-Protokoll basiert, und eine Alternative, die konzeptionell an *S/MIME* angelehnt ist und auf einer dateibasierten Architektur beruht. Dieses Kapitel erläutert den Entscheidungsprozess, der zur Wahl der *S/MIME*-inspirierten Variante führte.

3.1. Gegenüberstellung der Varianten

Um die Grundlage für die Entscheidung nachvollziehbar zu machen, werden die beiden evaluierten Varianten in diesem Abschnitt vorgestellt.

Variante 1: Ansatz basierend auf dem Double-Ratchet-Protokoll

Dieser Ansatz sieht vor, das *DR*-Protokoll als Kern für den Nachrichtenaustausch zu verwenden. *DR*, bekannt aus Anwendungen wie Signal, gilt als Goldstandard für sichere Messenger und bietet starke Sicherheitsgarantien wie *Forward Secrecy* und *Post-Compromise Security*. Die Kommunikation würde dabei auf ephemeren Sitzungsschlüsseln basieren, die kontinuierlich aktualisiert werden.

Variante 2: S/MIME-inspirierter, dateibasierter Ansatz

Diese Variante basiert auf einer Architektur, bei der jede Nachricht als separat verschlüsselte Datei auf der Plattform gespeichert wird. Die Verschlüsselung orientiert sich konzeptionell an *S/MIME*. Heisst für jede Nachricht wird ein symmetrischer *Session Key* erzeugt. Dieser Schlüssel wird dann für jeden Empfänger einzeln mit dessen öffentlichem Schlüssel asymmetrisch verschlüsselt. Dieser Ansatz priorisiert eine robuste und sichere Datenpersistenz sowie ein niederschwelliges Verfahren zur Etablierung von Vertrauen (*Bootstrapping*).

3.2. Ablehnungsgründe für das Double-Ratchet-Protokoll

Obwohl das *DR*-Protokoll starke Sicherheitsgarantien bietet, zeigten sich im Kontext der spezifischen Anforderungen dieser Arbeit grundlegende Zielkonflikte. Diese führten zu der Entscheidung, *DR* nicht als Basisprotokoll zu verwenden. Die wesentlichen Gründe werden nachfolgend erläutert.

3.2.1. Zielkonflikt mit der geforderten Datenpersistenz

Eine zentrale Anforderung an die Plattform ist die Bereitstellung einer sicheren und praktikablen Möglichkeit zur Wiederherstellung der Nachrichtenhistorie, beispielsweise nach einem Gerätewechsel. Das *DR*-Protokoll ist aufgrund seiner Architektur, insbesondere der kontinuierlichen Schlüsselableitung zur Gewährleistung von *Forward Secrecy*, inhärent nicht darauf ausgelegt.

Eine Wiederherstellung der Nachrichtenhistorie wäre nur durch zusätzliche Massnahmen realisierbar, beispielsweise durch Speicherung der entschlüsselten Inhalte oder der ephemeren Nachrichtenschlüssel. Beides würde jedoch die Kernsicherheitsmerkmale von *DR* untergraben. Würde man beispielsweise die Nachrichten entschlüsselt oder mit einem separaten, langlebigen symmetrischen Schlüssel (z.B. abgeleitet von einem Benutzerpasswort) sichern, entstünde ein neuer Angriffsvektor. Ein Angreifer würde sich auf den *Weakest Link*, den Schutzmechanismus des persistenten Datenspeichers, konzentrieren, anstatt das komplexe *DR*-Protokoll selbst zu brechen. Die starken Garantien von *DR* wären somit untergraben. Die Implementierung würde ein trügerisches Sicherheitsgefühl erzeugen, da die Gesamtsicherheit des Systems durch die gewählte Persistenzlösung limitiert wäre.

3.2.2. Zielkonflikt mit etablierten Authentifizierungsmethoden

Das Prinzip *PbD* fordert, dass Sicherheit für Nutzende niederschwellig und ohne aktive, fehleranfällige Mitwirkung realisiert wird. Für die Etablierung einer vertrauenswürdigen Kommunikation mittels *DR* ist ein authentifizierter initialer Schlüsselaustausch unerlässlich. In gängigen Implementierungen (oft in Verbindung mit *X3DH*) wird die finale *Authenticity* der Kommunikationspartner durch einen manuellen Abgleich von Sicherheitsnummern (Fingerprints der öffentlichen Schlüssel) durch die Nutzenden sichergestellt [23, 24].

Ein solcher manueller Verifikationsprozess ist im schulischen Umfeld mit einer heterogenen Nutzerschaft, die Lehrpersonen und Bezugspersonen mit unterschiedlichem technischen Verständnis umfasst, als unrealistisch einzustufen. Die Erfahrung zeigt, dass dieser Schritt selbst von technisch versierten Nutzenden häufig übergangen wird, wodurch ein wesentlicher Sicherheitsaspekt des Protokolls – die Sicherstellung, dass man mit der korrekten Gegenstelle kommuniziert, gennant *Authenticity* – entfällt. Dieser Umstand steht im Widerspruch zu einem umfassenden

PbD-Ansatz, der die Nutzenden nicht mit sicherheitskritischen Aufgaben überfordern sollte, deren Nichterfüllung die Schutzziele kompromittiert.

3.2.3. Komplexität und Entwicklungsaufwand

Die korrekte Implementierung des *DR*-Protokolls ist technisch anspruchsvoll. Sie erfordert tiefgreifendes kryptographisches Wissen, um Fallstricke zu vermeiden und die versprochenen Sicherheitsgarantien tatsächlich zu erreichen. Angesichts der genannten Zielkonflikte, insbesondere der Problematik der Datenpersistenz, welche die Kernvorteile von *DR* relativiert, wäre der hohe Implementierungsaufwand für diese Bachelorarbeit nicht gerechtfertigt.

3.3. Konzeption und Begründung des S/MIME-inspirierten Ansatzes

Der in dieser Arbeit entwickelte Ansatz wurde gezielt konzipiert, um die Anforderungen aus Abschnitt 2.3 zu erfüllen und dabei insbesondere die Schwachstellen des *DR*-Ansatzes in Bezug auf Datenpersistenz und initiale Authentifizierung zu vermeiden. Die Architektur dieser Lösung stützt sich auf die folgenden konzeptionellen Pfeiler.

Dateibasierte Ablagestruktur und Berechtigungskonzept

Im Kern des Ansatzes steht die zentrale Speicherung aller relevanten Daten, wie Nachrichten und Profildateien, als separate, verschlüsselte Dateien auf der Plattform. Inspiriert durch Konzepte wie Cryptree [25] und Proton Drive [26] werden Zugriffsrechte explizit über signierte *.permissions*-Dateien verwaltet. Diese Struktur unterstützt inhärent die asynchrone Kommunikation und legt die Grundlage für eine sichere Datenpersistenz.

S/MIME-inspirierter Verschlüsselungsmechanismus

Für die Kommunikation wird ein Ansatz gewählt, der konzeptionell an *S/MIME* angelehnt ist. Jede Nachricht wird mit einem eigens für sie generierten, symmetrischen *Session Key* verschlüsselt. Anschliessend wird der *Session Key* für jeden einzelnen Empfänger (einschliesslich des Senders selbst, um den Zugriff auf gesendete Nachrichten zu ermöglichen) mit dessen öffentlichem Schlüssel asymmetrisch verschlüsselt. Diese Sammlung verschlüsselter *Session Key*s wird zusammen mit der verschlüsselten Nachricht abgelegt. Dies ermöglicht eine effiziente und sichere 1:1- sowie Gruppenkommunikation mit durchgängiger *E2EE*.

Authentizität durch zentralen Vertrauensanker

Um den manuellen Schlüsselvergleich zu umgehen, wird ein *Bootstrapping*-Verfahren über einen authentischen Papierkanal etabliert (z.B. ein Einladungsbrief, der an einem Elternabend übergeben wird). Der Schuladmin fungiert als initialer Vertrauensanker. Er lädt neue Nutzende ein und verifiziert deren öffentliche Schlüssel, nachdem sie sich erfolgreich über das *Bootstrapping*-Verfahren registriert haben. Alle öffentlichen Schlüssel werden in einer zentralen, vom Admin signierten Datei verwaltet, was eine vertrauenswürdige Basis für die gesamte Kommunikation schafft.

Schlüsselverwaltung und Datenwiederherstellung

Die zentrale Speicherung verschlüsselter Daten bedingt einen robusten Mechanismus zur Datenwiederherstellung, insbesondere im Falle eines Geräteverlusts. Da die privaten Schlüssel des Nutzers lokal auf dem Gerät gespeichert sind, würde deren Verlust den Zugriff auf alle zentral gespeicherten Daten unmöglich machen. Um diesem Risiko zu begegnen, integriert die Architektur ein dezidiertes Schlüsselverwaltungskonzept. Die privaten Schlüssel werden mit einem symmetrischen Schlüssel (*User Secret Key* (usk)) verschlüsselt, der aus einem nur dem Nutzer bekannten Geheimnis (*User Secret* (us), z.B. Passwort oder *Passkey*) abgeleitet wird. Dieser verschlüsselte Container, der *Keystore*, wird ebenfalls sicher auf der Plattform hinterlegt, was es den Nutzern ermöglicht, ihre Schlüssel auf einem neuen Gerät allein durch die Eingabe ihres Geheimnisses wiederherzustellen und somit den Zugriff auf ihre Kommunikationshistorie zu gewährleisten.

Diese Design-Entscheidungen führen zu einer Architektur, die alle zentralen Anforderungen erfüllt und eine pragmatische, sichere Lösung für den schulischen Kontext darstellt. Die wesentlichen Stärken sind:

- **Ganzheitliches Privacy by Design:** Die Kombination aus der dateibasierten Architektur, dem nutzerfreundlichen Bootstrapping und der Minimierung von Klartext-Metadaten setzt die *PbD*-Prinzipien konsequent um, ohne die Anwendbarkeit zu beeinträchtigen.
- **Resilienz und Datenverfügbarkeit:** Die zentrale, verschlüsselte Speicherung aller Daten entkoppelt die Kommunikationshistorie vom physischen Gerät. Das Keystore-Konzept liefert den robusten Mechanismus, um den Zugriff nach einem Geräteverlust wiederherzustellen, was eine Kernanforderung erfüllt, die mit *DR*-basierten Systemen nur schwer zu vereinbaren ist.
- **Nahtlose Sicherheit und Plattformunabhängigkeit:** Der *S/MIME*-inspirierte Mechanismus gewährleistet durchgängige *E2EE* für alle Kommunikationsformen, die für den Nutzer unsichtbar im Hintergrund abläuft. Die serverseitige Speicherung des verschlüsselten Keystores ermöglicht zudem einen nahtlosen Mehrgeräte-Support und die Nutzung über einen Webbrowser.

Der Verzicht auf perfekte *Forward Secrecy* ist dabei ein bewusster Entwurfsentscheid. Er ermöglicht eine Lösung, die durch ihre Robustheit, Benutzerfreundlichkeit und kontextspezifische Architektur den realen Bedürfnissen im Bildungswesen präzise gerecht wird.

4. Beschreibung der Plattform

Dieses Kapitel beschreibt die Architektur und Funktionsweise der konzipierten Schulkommunikationslösung, die auf den in Kapitel 3 getroffenen Entscheidungen basiert. Es wird erläutert, wie die Plattform unter Berücksichtigung der Prinzipien von *PbD* aufgebaut ist, welche Parteien involviert sind und wie deren Interaktionen, insbesondere der Nachrichtenaustausch, gestaltet sind. Der Fokus liegt auf einer konzeptionellen Darstellung der Systemkomponenten und der zugrundeliegenden Sicherheitsmechanismen, um ein klares Verständnis der Gesamtarchitektur zu vermitteln, bevor in Kapitel 5 die detaillierte Protokollspezifikation erfolgt.

4.1. Parteien, Schutzbedarf und technische Sicherheitsprinzipien

Die hier beschriebene Schulkommunikationslösung orientiert sich an realen Abläufen im Schulalltag und berücksichtigt die verschiedenen Parteien, die an der schulischen Kommunikation beteiligt sind. Durch eine vertrauliche und authentifizierte Kommunikation zwischen diesen Parteien wird ein verantwortungsvoller Umgang mit sensiblen Daten ermöglicht, ohne die Nutzerfreundlichkeit zu beeinträchtigen.

4.1.1. Parteien

Im Rahmen der Schulkommunikationslösung werden verschiedene Parteien unterschieden, die jeweils spezifische Aufgaben und Verantwortlichkeiten übernehmen. Diese Rollen bilden die Grundlage für das Protokolldesign. Analog zu den technischen Begriffen werden die Parteien in englischer Sprache bezeichnet und als Fachbegriffe kursiv hervorgehoben. Im Folgenden werden die Parteien beschrieben.

- Der *Admin* vertritt die schulische Instanz und verwaltet die *Platform*. Zu seinen Aufgaben gehören das Erfassen von *Students*, das Einladen von *Caregivers* und *Teachers*, das Erstellen von Klassen sowie die Verwaltung von Zugriffsrechten. Zudem prüft und bestätigt er Registrierungsanfragen.
- Der *Teacher* nutzt die *Platform* zur Kommunikation mit *Caregivers* und zum Teilen von Informationen über den Schulalltag. Dazu stehen ihm direkte Chats (1:1) sowie klasseninterne Kanäle zur Verfügung. Innerhalb des Klassenchats

können *Teachers* Beiträge verfassen, während *Caregivers* dort nur Lesezugriff haben.

- Der *Caregiver* (z. B. Eltern oder Erziehungsberechtigte) verwendet die *Platform*, um Informationen über die ihm zugewiesenen *Students* zu erhalten, Absenzen zu melden und mit *Teachers* zu kommunizieren. Er besitzt Lesezugriff auf den Klassenchat und kann über direkte Chats mit den jeweiligen *Teachers* in Kontakt treten.
- Der *Student* wird durch den *Admin* im System erfasst, nutzt die *Platform* selbst jedoch nicht aktiv. Für das Schulkommunikationslösung ist die eindeutige Zuordnung von *Students* zu ihren *Caregivers* erforderlich, damit organisatorische Informationen korrekt übermittelt werden können.
- Die *Platform* bildet das zentrale Back-End-System, das alle Benutzerdaten verwaltet und Signaturen auf Korrektheit prüft. Da ein *E2EE*-Ansatz verfolgt wird, kann die *Platform* die Inhalte (z. B. Chats) nicht einsehen. Sie dient lediglich dem sicheren Speichern und Verteilen der Daten.
- Ein *Client* bezeichnet das Endgerät, über das eine Partei mit der *Platform* interagiert. Beispiele sind Desktop-Computer, Notebooks, Tablets oder Smartphones mit ausreichender Rechenleistung für kryptographische Operationen. Die auf dem *Client* installierte Software (z. B. Web- oder Mobile-App), kurz «App», fungiert als Benutzeroberfläche und Kommunikationsschnittstelle. Um eindeutig zu kennzeichnen, welcher Partei ein *Client* zugeordnet ist, wird ein Index verwendet, etwa $client_a$ für den *Admin*, $client_t$ für einen *Teacher* oder $client_c$ für einen *Caregiver*.

Jede menschliche Partei verfügt über eine eindeutige und hochentropische Benutzeridentifikationsnummer (*User Identifier User* i ($uid_i,\ i \in \{a, t, c, s\}$)) sowie ein kryptographisches *Key Pair*, bestehend aus dem *Public Key User* i ($PUK_i,\ i \in \{a, t, c\}$) und dem *Private Key User* i ($prk_i,\ i \in \{a, t, c\}$), wobei der Index i die jeweilige Partei identifiziert.

4.1.2. Schutzbedarf und Datenschutzprinzipien

Aus Sicht einer nach *PbD* gestalteten Schulkommunikationslösung sind alle personenbezogenen Informationen als sensibel zu behandeln, insbesondere dann, wenn sie Rückschlüsse auf Minderjährige zulassen.

Dazu zählen unter anderem:

- Identitäts- und Kontaktdaten von *Students*, *Caregivers* und *Teachers*,
- schulische Leistungen, Verhalten oder Abwesenheiten,
- physische und psychische Erkrankungen,

- Inhalte von Nachrichten zwischen *Teachers* und *Caregivers*

Auch wenn einige dieser Informationen oberflächlich betrachtet harmlos erscheinen mögen, können sie zur Bildung eines Persönlichkeitsprofils verwendet werden. Besonders bei *Students* ist daher höchste Zurückhaltung geboten. Die *Platform* verfolgt deshalb das Prinzip der Datenminimierung. Das bedeutet, dass jede Partei nur auf jene Informationen zugreifen können soll, die sie zur Erfüllung ihrer Rolle unbedingt benötigt.

Die konkrete Umsetzung dieser Zugriffsbeschränkung orientiert sich an den analogen Strukturen des Schulalltags. Der *Admin* verfügt über erweiterte Berechtigungen und kann auf die meisten schulorganisatorischen Daten zugreifen, etwa auf Klassenzuordnungen oder Namen der *Students*. Kein Zugriff besteht jedoch auf Direktnachrichten, etwa zwischen *Teachers* und *Caregivers*.

Teachers sehen ausschliesslich Informationen über *Students* ihrer eigenen Klassen. Ein Einblick in andere Klassen oder schulübergreifende Daten ist nicht vorgesehen. *Caregivers* wiederum erhalten nur Zugriff auf Informationen zu den ihnen zugewiesenen *Students* sowie auf allgemein verfügbare Informationen innerhalb deren Klassen.

4.1.3. Anwendung der Kryptographie in der Schulkommunikationslösung

Die Schulkommunikationslösung folgt dem Prinzip der *E2EE*. Das bedeutet, dass alle sensiblen Daten ausschliesslich in der jeweiligen App der nutzenden Partei ver- und entschlüsselt sowie signiert werden. Die *Platform* selbst verarbeitet ausschliesslich verschlüsselte Inhalte und hat zu keinem Zeitpunkt Zugriff auf Klartextdaten.

Verschlüsselung dient dabei zur Steuerung der Lesezugriffe. Nur wenn eine Partei den passenden symmetrischen Schlüssel besitzt, kann sie eine damit verschlüsselte Datei lesen.

Signaturen dienen zur Steuerung der Schreib- bzw. Bearbeitungsrechte. Nur Parteien, denen explizit Schreibrechte zugewiesen wurden, können eine Datei verändern und anschliessend eine gültige neue Signatur erzeugen.

Die folgenden Dateien sind nicht verschlüsselt und können somit prinzipiell von jeder Partei gelesen werden. Dies stellt jedoch kein Sicherheitsrisiko dar, da sie keine personenbezogenen Daten enthalten:

- *publickeys* Datei: Enthält die öffentlichen Schlüssel der Parteien. Diese muss für alle zugänglich sein, damit Informationen verschlüsselt bzw. Signaturen geprüft werden können.
- *permissions* Dateien: Enthalten die mit öffentlichen Schlüsseln verschlüsselten symmetrischen Schlüssel zum Entschlüsseln der jeweiligen Dateien, welche auf der *Platform* liegen. Die *permissions* Dateien selbst können daher nicht

verschlüsselt werden.

- *otps* und *totps* Dateien: Enthalten die *One Time Passwords* (otps) für die *Caregivers* und die *Teachers*. Die darin enthaltenen otps sind für den *Admin* verschlüsselt. Die übrigen Daten sind nicht personenbezogen.
- *keystore* Datei: Enthält die privaten Schlüssel der nutzenden Partei, welche jeweils mit einem benutzerspezifischen Geheimnis (z.B. Passwort) verschlüsselt sind. Die Datei selbst ist nicht verschlüsselt, jedoch ist ihr sensitiver Inhalt geschützt (Details in Abschnitt 5.1.8).

Alle Dateien[1], unabhängig davon, ob sie verschlüsselt sind oder nicht, sind zusätzlich digital signiert.

Damit diese Rechte überprüfbar sind, wird in der jeweiligen *permissions*-Datei des Ordners festgehalten, welche Partei eine bestimmte Datei lesen bzw. entschlüsseln und wer sie signieren bzw. bearbeiten darf. Das Leserecht wird technisch so umgesetzt, dass der symmetrische Schlüssel, mit dem die Datei verschlüsselt wurde, zusätzlich mit dem öffentlichen Schlüssel jeder leseberechtigten Partei verschlüsselt wird. Diese kann den symmetrischen Schlüssel mit ihrem privaten Schlüssel entschlüsseln und damit anschliessend die Datei lesen.

Alle kryptographischen Operationen wie Verschlüsselung, Entschlüsselung, Signatur und Verifikation werden vollständig in der jeweiligen App ausgeführt und laufen für die nutzende Partei im Hintergrund ab. Tritt dabei ein sicherheitsrelevanter Fehler auf, wird die betroffene Partei automatisch informiert. In einem solchen Fall stellt die App den Betrieb ein, um die Sicherheit der Daten zu gewährleisten. Zusätzlich besteht die Möglichkeit, den Vorfall an die zuständigen Stellen zu melden.

4.1.4. Schlüsselsicherung und Datenwiederherstellung

Ein fundamentales Spannungsfeld in *E2EE*-Systemen ist der Konflikt zwischen Sicherheit und Benutzerfreundlichkeit im Falle eines Geräteverlusts. Die Handhabung von Wiederherstellungsmechanismen stellt bei etablierten Messenger-Diensten eine komplexe Herausforderung dar, die unterschiedlich gelöst wird. WhatsApp beispielsweise ermöglicht es Nutzern, den Nachrichtenverlauf passwortgeschützt in externen Cloud-Diensten wie Google Drive oder iCloud zu sichern [27]. Threema bietet mit Threema Safe eine eigene serverbasierte Lösung an, während Signal einen restriktiveren Ansatz ohne integrierte Cloud-Sicherung verfolgt, bei dem die Datenübertragung ein funktionstüchtiges Altgerät erfordert [28, 29]. Diese Ansätze verdeutlichen einen steten Kompromiss zwischen Benutzerfreundlichkeit und potenziellen Datenschutzrisiken. Insbesondere die Nutzung externer, nicht-schweizerischer Cloud-Dienste ist im schulischen Kontext inakzeptabel.

[1] Eine Ausnahme stellt das *Registrationpackage* dar, siehe dazu Abschnitt 4.2.2.

Die hier konzipierte Plattform verfolgt daher einen fundamental anderen Ansatz. Statt die verschlüsselten Daten in einem separaten Prozess zu sichern, wird der Zugriff auf die persistent auf der Plattform gespeicherten Daten wiederherstellbar gemacht. Die Architektur entkoppelt somit den Zugriff auf die eigenen Daten von der Bindung an ein einzelnes physisches Gerät.

Das Herzstück dieses Konzepts ist der benutzerspezifische *Keystore*. Dabei handelt es sich um einen verschlüsselten Container, der die privaten Schlüssel des Nutzers enthält. Dieser Keystore wird nicht nur lokal auf dem Gerät, sondern auch auf der Plattform selbst gespeichert. Seine Sicherheit wird dadurch gewährleistet, dass er mit einem symmetrischen Schlüssel (usk) verschlüsselt ist, der ausschliesslich aus einem nur dem Nutzer bekannten Geheimnis (us) wie etwa einem starken Passwort oder einem *Passkey* abgeleitet wird. Für solche Geheimnisse existieren bewährte Wiederherstellungsverfahren, die auch bei einem vergessenen Passwort den Zugriff auf die Plattformdaten ermöglichen. Die konkrete Ausgestaltung dieser Wiederherstellungsverfahren ist jedoch nicht Gegenstand dieser Arbeit.

Diese Architektur bietet entscheidende Vorteile und macht die Lösung ganzheitlich überlegen.

- **Resilienz bei Geräteverlust.** Verliert ein Nutzer sein Gerät, verliert er nicht den Zugriff auf seine Daten. Durch die Eingabe seines Geheimnisses auf einem neuen Gerät kann der Keystore von der Plattform geladen, entschlüsselt und die privaten Schlüssel wiederhergestellt werden. Der Zugriff auf die gesamte, zentral gespeicherte Kommunikationshistorie ist damit sofort wieder möglich.
- **Nahtloser Mehrgeräte- und Browser-Support.** Da sowohl die verschlüsselten Daten als auch der verschlüsselte Keystore zentral verfügbar sind, können Nutzer sich auf beliebig vielen Geräten gleichzeitig anmelden. Der Prozess der Schlüssel-Synchronisation wird für den Nutzer im Hintergrund abgewickelt.
- **Maximale Einfachheit für den Endnutzer.** Die gesamte Komplexität der Schlüsselsicherung und -wiederherstellung wird von der Applikation gehandhabt. Der Nutzer muss keine manuellen Backups erstellen oder Daten zwischen Geräten transferieren. Seine einzige Verantwortung liegt in der sicheren Wahl und Aufbewahrung seines persönlichen Geheimnisses.

Damit wird die Datenwiederherstellung kein nachträglich hinzugefügter Prozess, sondern ein integraler Bestandteil der Sicherheitsarchitektur, der die Resilienz und Benutzerfreundlichkeit des Systems von Grund auf gewährleistet.

4.2. Aufsetzen der Plattform

Die *Platform* wird durch den Plattformbetreiber vorbereitet und besitzt dadurch eine initiale Grundstruktur, bestehend aus Ordnern, wie in Abb. 4.2 dargestellt. Zu Be-

ginn eines Schuljahres muss die *Platform* für die Nutzung vorbereitet werden. Dieser Vorgang erfolgt durch den *Admin* und wird in den folgenden Abschnitten erläutert.

Zu jedem Abschnitt wird eine Abbildung gezeigt, die den jeweiligen Ablauf visuell veranschaulicht. Elemente, die bereits im vorherigen Schritt vorhanden waren, sind in Schwarz dargestellt. Neue Elemente, die im aktuellen Abschnitt hinzukommen, werden in Grün hervorgehoben. Wo sinnvoll, werden Teilschritte nummeriert, um Abläufe klarer zu strukturieren.

Die in den Abbildungen 4.2 bis 4.7 verwendete Notation wird in Abbildung 4.1 erläutert. Die Grafiken sind von oben nach unten zu lesen. Dabei stellen weiter unten liegende und über eine Linie verbundene Ordner jeweils Unterordner dar. Gleiches gilt für Dateien, die durch Wörter ohne Rahmen, aber mit Signatur- und/oder Schloss-Symbol dargestellt werden.

Falls mehrere Parteien eine Datei lesen (entschlüsseln) oder bearbeiten (signieren) dürfen, werden entsprechend mehrere Schloss- (für Verschlüsselung bzw. Leserecht) bzw. Signatursymbole (für Signierung bzw. Bearbeitungsrecht) in den Farben der jeweiligen Schlüssel dargestellt. Eine zusammenfassende Übersicht der Berechtigungen für die einzelnen Dateien findet sich im Abschnitt 4.2.5.

An dieser Stelle wird bewusst darauf verzichtet, die Initialisierung des *Admin*-Accounts im Detail zu beschreiben. Der Fokus der Schulkommunikationslösung liegt auf dem sicheren Ablegen und Austauschen schulinterner Informationen. Da die korrekte Einrichtung des *Admin* essenziell für die Gesamtsicherheit der *Platform* ist, wird dieser Aspekt in der technischen Spezifikation in Abschnitt 5.2.2 ausführlich behandelt. Für die im Folgenden beschriebenen Schritte wird davon ausgegangen, dass der *Admin*-Account bereits korrekt initialisiert wurde.

4.2.1. Erstellung einer Klasse

Zu Beginn werden durch den *Admin* die Klassen erstellt. Dies ist anhand einer Klasse in Abb. 4.3 veranschaulicht. Jede Klasse verfügt über eine *students* und eine *teachers*-Datei, welche alle *User Identifier Student* (uid_s) der *Students* enthält, die dieser Klasse zugeordnet sind, bzw. die *User Identifiers* (uids) der *Teachers*, welche diese Klasse unterrichten. Zudem besitzt jede Klasse einen Klassenchat (*Classchat*) in Form eines Ordners. Darin wird jede Nachricht als einzelne Datei abgelegt, beispielsweise die Datei *message-0*. In der Datei *parties* werden die Chat-Teilnehmenden vom *Admin* eingetragen. Dabei ist zu beachten, dass nur *Teachers* im Klassenchat Nachrichten senden können. *Caregivers* haben lediglich Leserechte auf den Chat. Details zum Nachrichtenaustausch werden in Abschnitt 4.3 erläutert.

Abbildung 4.1.: In den Abbildungen 4.2 bis 4.7 verwendete Notation.

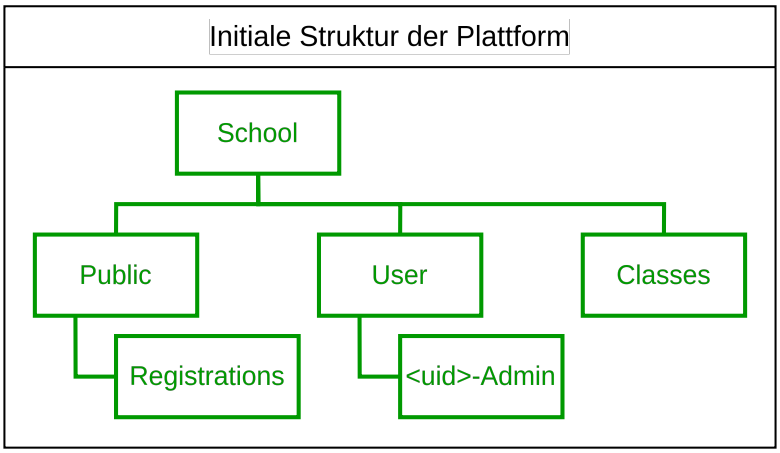


Abbildung 4.2.: Initiale Ordnerstruktur der *Platform*, erstellt durch den Plattformbetreiber.

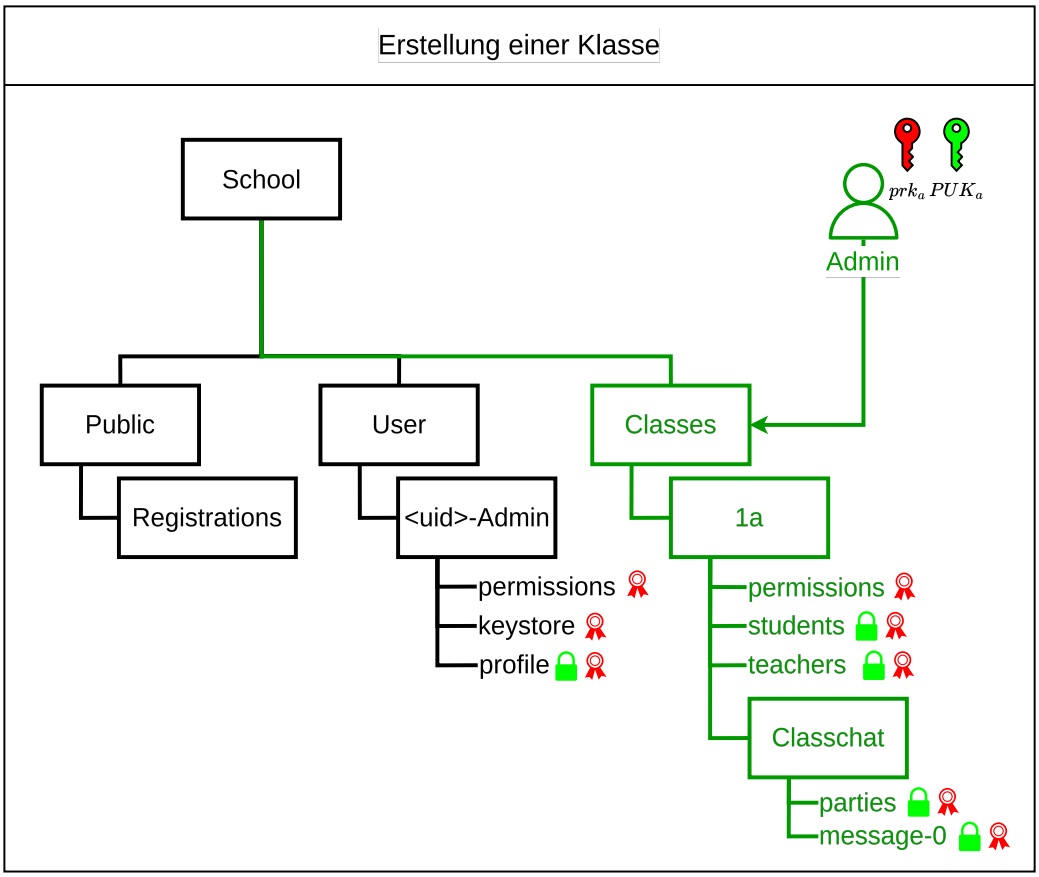


Abbildung 4.3.: Erstellung eines Klassenordners durch den *Admin*, inklusive leerer Dateien.

4.2.2. Registration eines Teachers

Der Registrierungsprozess eines *Teachers* wird vom *Admin* initiiert und folgt einem mehrstufigen Protokoll, das durch Abb. 4.4 ergänzt wird. Die Verweise in Klammern, wie (1) oder (2), beziehen sich auf die nummerierten Teilschritte in der Grafik.

Zunächst generiert die App des *Admins* für jeden neuen *Teacher* eine Benutzeridentifikationsnummer (*User Identifier Teacher* (uid_t)), die keine Rückschlüsse auf die Person zulässt. Zusätzlich erzeugt die App für jeden *Teacher* ein zufälliges, einmalig verwendbares Passwort, ein sogenanntes otp. Dieses otp wird für den *Admin* verschlüsselt und in der Datei *Teacher OTPs* ($TOTPS$) gespeichert (1).

Anschliessend werden die uid_t, das zugehörige otp sowie der öffentliche Schlüssel des *Admin* in einem QR-Code codiert und auf ein Einladungsschreiben (*Invitationletter*) gedruckt. Dieses wird über einen authentischen Kanal, beispielsweise durch persönliche Übergabe, an den *Teacher* übermittelt (2).

Nach Erhalt des Einladungsschreibens scannt der *Teacher* den QR-Code mit seiner App und gibt seine persönlichen Daten sowie ein nur ihm bekanntes Geheimnis (us) ein. Die App initiiert daraufhin die sichere Generierung der kryptographischen Schlüssel direkt auf dem Endgerät des *Teachers*. Die neu erstellten privaten Schlüssel werden mit dem abgeleiteten Benutzergeheimnis (usk) verschlüsselt und im *Keystore* abgelegt. Gleichzeitig erstellt die App ein Registrierungspaket (*Registrationpackage*), das sowohl das otp als auch den öffentlichen Schlüssel des *Teachers* enthält. Die *Keystore* Datei und das für den *Admin* verschlüsselte *Registrationpackage*, mit dem Namen *rpt-‹uid›.enc*, werden anschliessend auf die *Platform* hochgeladen (3).

Zu beachten ist, dass das *Registrationpackage* als einzige Datei nicht signiert wird. Eine Signatur wäre an dieser Stelle nicht verlässlich überprüfbar, da der öffentliche Schlüssel des *Teachers* dem *Admin* erst durch diesen Schritt bekannt wird.

Die App des *Admin* lädt das *Registrationpackage* und die *Keystore* Datei herunter, entschlüsselt das *Registrationpackage* und prüft die Gültigkeit des enthaltenen otp (4). Bei erfolgreicher Prüfung wird der öffentliche Schlüssel des *Teachers* in der Datei *publickeys* gespeichert (5), welche vom *Admin* signiert wird. Die *Keystore* Datei und *profile* Datei, welche die personenbezogenen Daten des *Teachers* enthält, werden im persönlichen Ordner des *Teachers* abgelegt, identifizierbar über dessen uid_t. Die *profile*-Datei wird sowohl für den *Teacher* selbst als auch für den *Admin* verschlüsselt. Der dabei verwendete symmetrische Schlüssel wird in der *Admin* App mit dem öffentlichen Schlüssel des *Teachers* verschlüsselt und in der Datei *permissions* abgelegt. Dadurch kann der *Teacher* zu einem späteren Zeitpunkt beim herunterladen der Datei durch das entschlüsseln dieses symmetrischen Schlüssels auf seine Datein zugreifen. Dieser Prozess wird für den *Admin* selbst wiederholt, um die Berechtigungen wie in der Tabelle 4.1 beschrieben sicherzustellen. Weiter wird die uid des *Teachers* in der Datei *teachers* jener Klasse abgelegt, welche er unterrichtet (6). Die

Zugriffsberechtigungen des *Teachers* werden in der Datei *permissions* festgehalten und sind in Tabelle 4.1 ersichtlich.

4.2.3. Registrieren der Students

Der *Admin* erhält vom Kanton eine Liste mit den *Students*, welche sämtliche Informationen wie Name, Vorname usw. enthält und importiert diese über die App. Für jeden *Student* wird daraufhin auf dem Endgerät des *Admins* eine Benutzeridentifikationsnummer (uid_s) generiert, die keine Rückschlüsse auf die Identität des *Student* zulässt.

Alle personenbezogenen Daten wie Name, Vorname und Geburtsdatum werden lokal auf dem Gerät des *Admins* verarbeitet und in die *profile*-Datei geschrieben bevor sie auf die *Platform* hochgeladen werden. Zusätzlich wird eine leere *absences*-Datei für die künftige Erfassung von Absenzen erstellt.

Des Weiteren werden die *Teachers*, welche den jeweiligen *Student* unterrichten, in der Datei *teachers* eingetragen. Zudem wird die uid des *Student* in der Datei *students* der entsprechenden Klasse erfasst. Die Abbildung 4.5 verdeutlicht den Ablauf exemplarisch für einen *Student*. Sowohl der *Admin* als auch die *Teachers*, welche den *Student* unterrichten, erhalten Zugriffsrechte auf die Dateien des *Student*. Details dazu finden sich in der Tabelle 4.1.

4.2.4. Registration eines Caregivers

Der Ablauf der Registration eines *Caregiver* ist in Abb. 4.6 dargestellt.

Für jeden *Student* erstellt der *Admin* mit der App n (n kann beliebig gewählt werden) Benutzeridentifikationsnummern (*User Identifier Caregiver* (uid_c)). Zusätzlich generiert er für jede uid_c ein otp. Dieses otp wird für den *Admin* verschlüsselt in der Datei *COTPS* Datei ($COTPS$) abgelegt (1).

Aus der uid_c, dem zugehörigen otp, dem öffentlichen Schlüssel des *Admin* und den relevanten Informationen aus dem Profil des betroffenen *Student* wird ein *Invitationletter* erstellt. Dieses enthält einen QR-Code mit allen benötigten Informationen und wird über einen vertrauenswürdigen Kanal, z.B. persönlich durch den *Student* oder an einem Elternabend, an den vorgesehenen *Caregiver* übermittelt (2).

Der *Caregiver* scannt den QR-Code mit seiner App, bestätigt die angezeigte Zuordnung zum *Student* und gibt seine persönlichen Daten sowie ein nur ihm bekanntes Geheimnis (us) ein. Die App initiiert daraufhin die sichere Generierung der kryptographischen Schlüssel direkt auf dem Endgerät des *Caregivers*. Die neu erstellten privaten Schlüssel werden mit dem abgeleiteten Benutzergeheimnis (usk) verschlüsselt und im *Keystore* abgelegt. Gleichzeitig erstellt die App ein *Registrationpackage*,

das die öffentlichen Schlüssel, das otp und die Profildaten enthält. Die *Keystore* Datei und das für den *Admin* verschlüsselte *Registrationpackage* werden anschliessend auf die *Platform* hochgeladen (3).

Wie bei der Registrierung eines *Teacher* wird auch dieses *Registrationpackage* nicht signiert, da der öffentliche Schlüssel des *Caregiver* dem *Admin* erst durch diesen Schritt bekannt wird und eine Signatur daher nicht zuverlässig überprüfbar wäre.

Die App des *Admin* lädt das *Registrationpackage* und die *Keystore* Datei herunter, entschlüsselt das *Registrationpackage* und prüft die Gültigkeit des enthaltenen otp (4). Hierbei ist zu erwähnen, dass die App des *Admin* die *Registrationpackage* durch die unterschiedlichen Namen für *Caregiver* und *Teacher* unterscheiden kann. Ist das otp gültig, wird der öffentliche Schlüssel des *Caregiver* in der Datei *publickeys* gespeichert (5), welche vom *Admin* signiert wird. Die *Keystore* Datei und die *profile* Datei, welche die personenbezogenen Daten des *Caregiver* enthält, werden im persönlichen Ordner des *Caregiver* abgelegt, identifizierbar über dessen uid_c. Die *profile*-Datei wird sowohl für den *Caregiver* selbst als auch für den *Admin* sowie für jene *Teachers* verschlüsselt, welche einen *Student* des *Caregiver* unterrichten. Der dabei verwendete symmetrische Schlüssel wird in der *Admin* App jeweils mit dem öffentlichen Schlüssel des *Caregiver* verschlüsselt und in der *permissions* Datei abgelegt. Dadurch kann der *Caregiver* in einem späteren Zeitpunkt beim herunterladen der Datei durch das entschlüsseln dieses symmetrischen Schlüssels auf seine Datein zugreifen. Dieser Prozess wird für den *Admin* und *Teacher* wiederholt, um die Berechtigungen wie in der Tabelle 4.1 beschrieben sicherzustellen. Im Ordner des jeweiligen *Student* wird eine Datei mit einem Verweis auf den zugehörigen *Caregiver* abgelegt und im Ordner des *Caregiver* ein entsprechender Verweis auf den betreuten *Student*. Diese Referenzdateien *caregivers* und *students* dürfen vom *Caregiver* gelesen, jedoch nur vom *Admin* bearbeitet werden (6). Die Zugriffsberechtigungen des *Caregiver* werden in der Datei *permissions* festgehalten und sind in Tabelle 4.1 ersichtlich.

4.2.5. Finale Struktur der Plattform

Die Abbildung 4.7 zeigt die vollständige Struktur der *Platform*, einschliesslich der jeweils gültigen Zugriffsberechtigungen auf die einzelnen Dateien. Eine zusammenfassende Übersicht, wer welche Datei lesen (R) oder schreiben (W) darf, sobald sie sich auf der *Platform* befindet, ist in Tabelle 4.1 dargestellt.

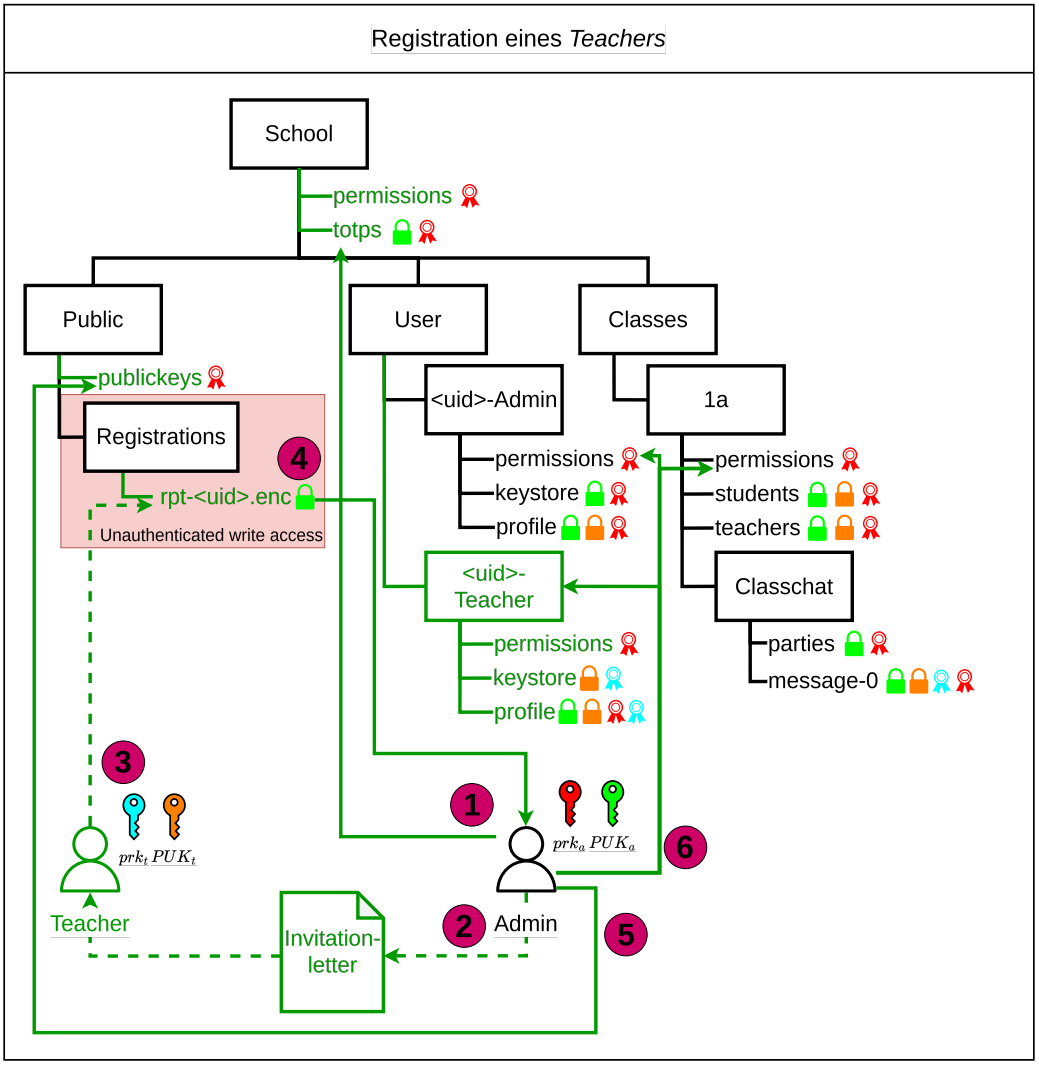


Abbildung 4.4.: Registrierung eines *Teacher* auf der *Platform*, inklusive Erstellung aller benötigten Dateien und Vergabe der Berechtigungen.

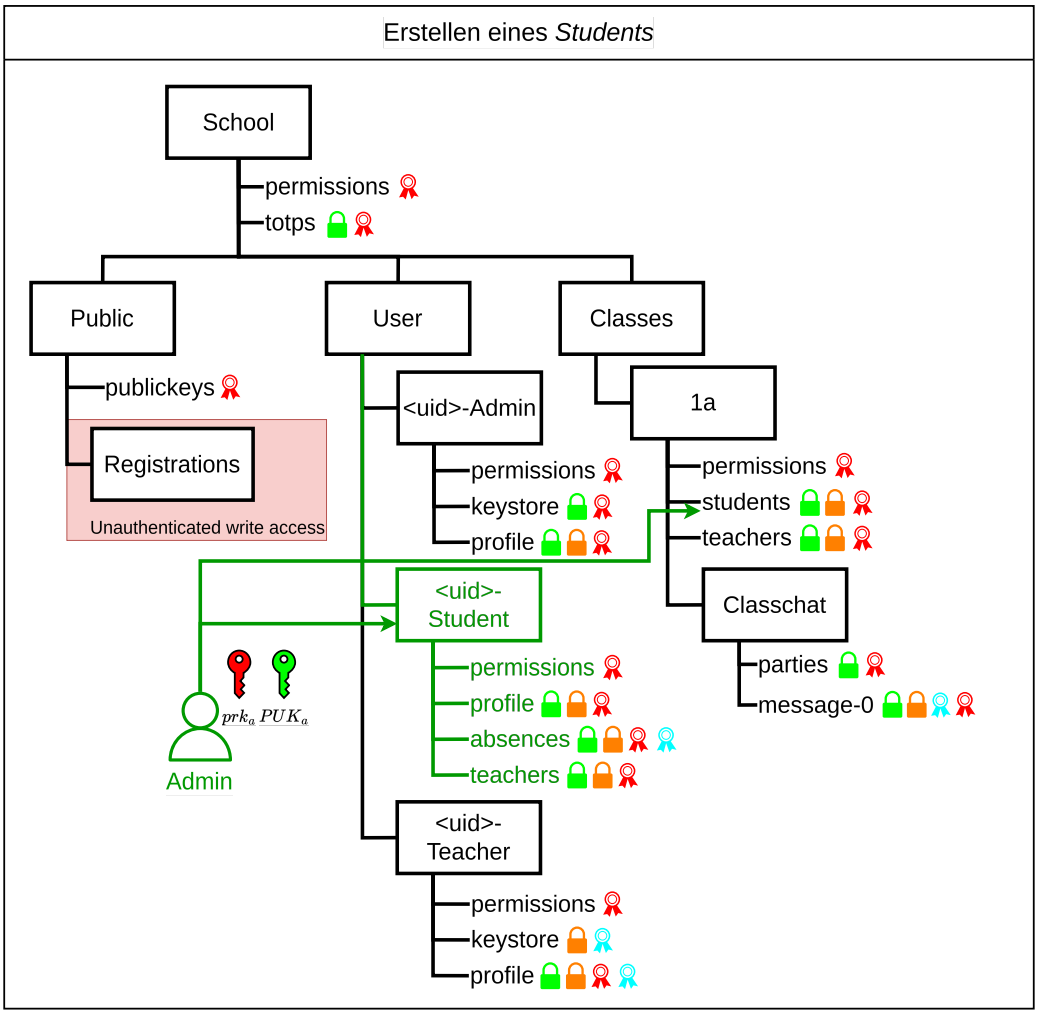


Abbildung 4.5.: Erstellen eines *Student* Ordners, inklusive Erstellung aller benötigten Dateien und Vergabe der Berechtigungen.

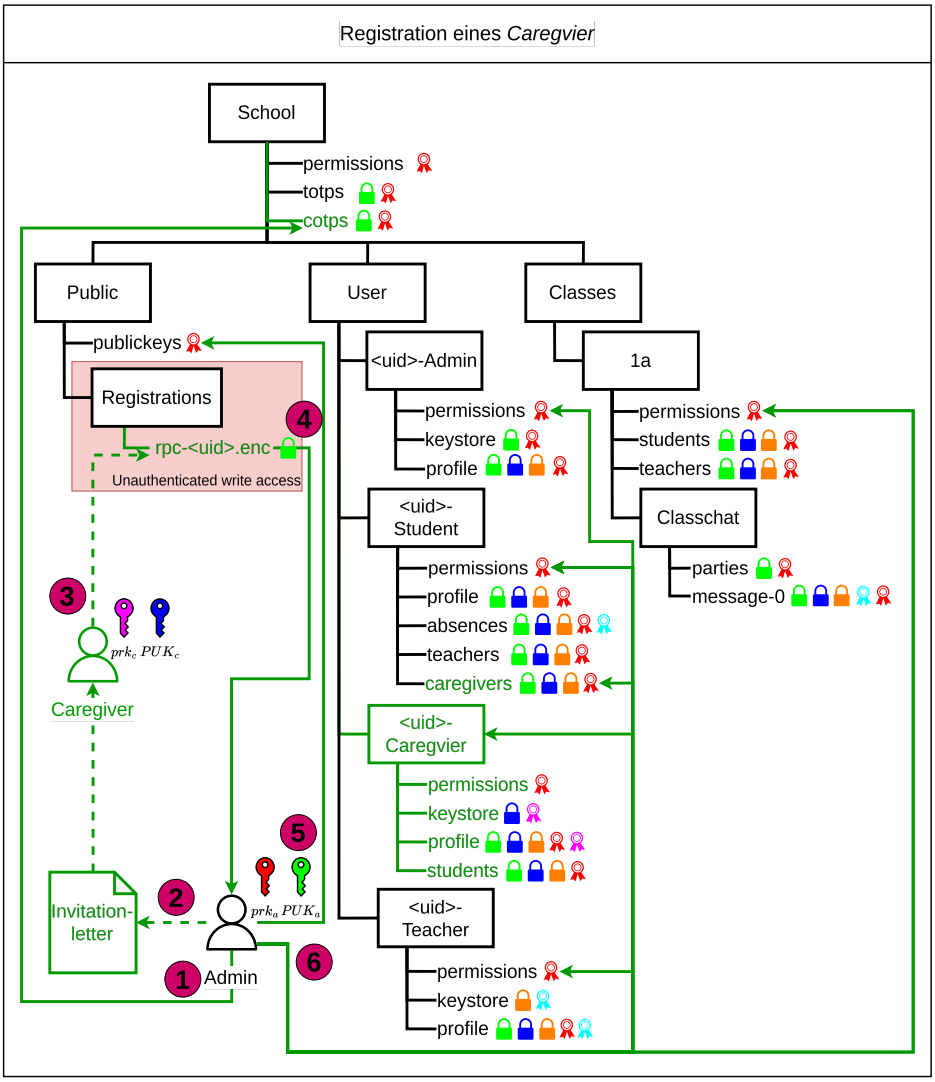


Abbildung 4.6.: Registrierung eines *Caregiver* auf der *Platform*, inklusive Erstellung aller benötigten Dateien und Vergabe der Berechtigungen.

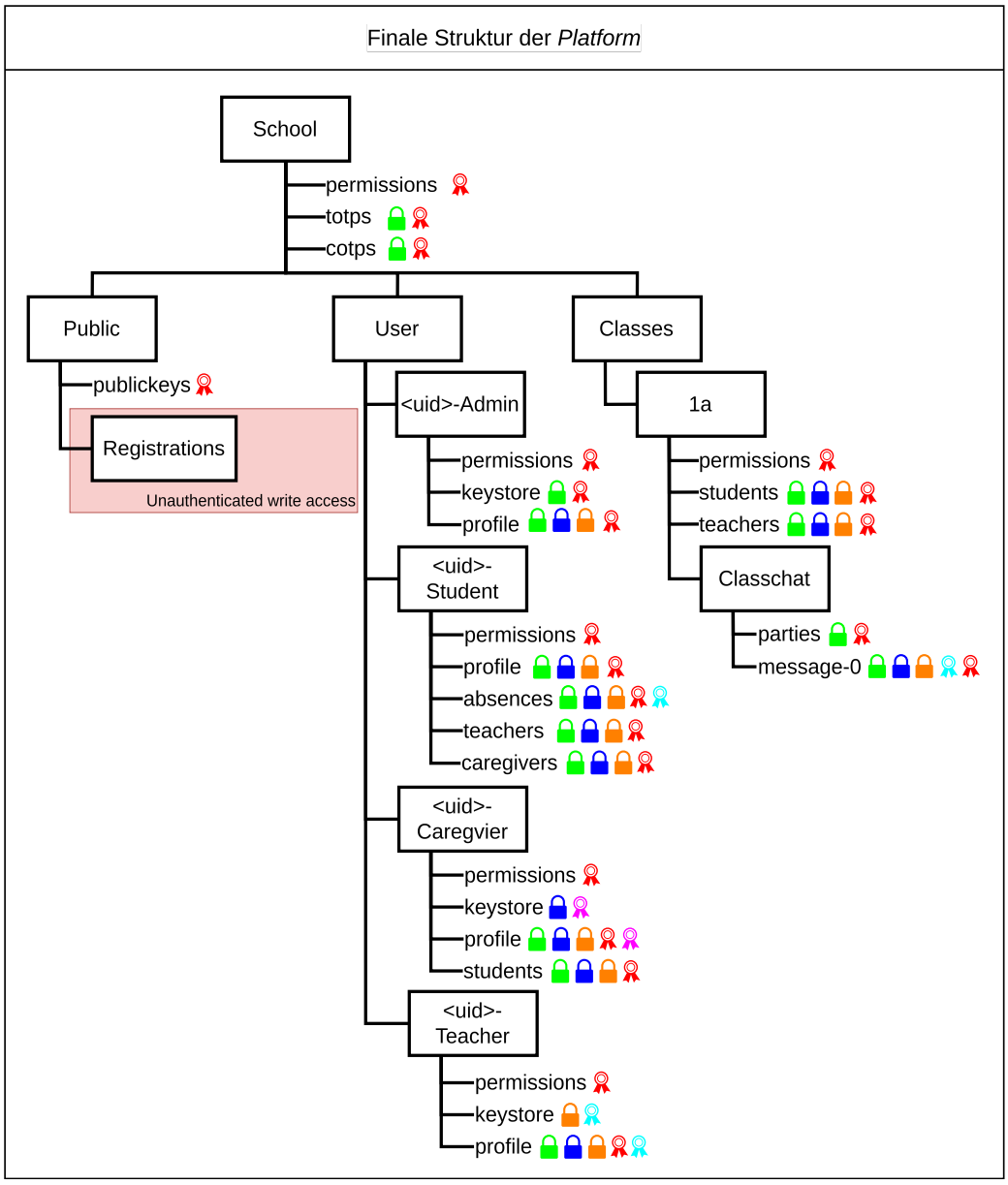


Abbildung 4.7.: Finale Ordnerstruktur der *Platform* mit den zugehörigen Berechtigungen (Schloss = Leserecht, Signatursymbol = Bearbeitungsrecht).

Datei	*Admin*	*Teacher*	*Caregiver*
permissions[#]	R, W	R	R
totps	R, W	R[‡‡]	R[‡‡]
cotps	R, W	R[‡‡]	R[‡‡]
publickeys	R, W	R	R
Registrationpackage-Teacher	R, W[†]	–	–
Registrationpackage-Caregiver	R, W[†]	–	–
profile-Admin	R, W	R	R
profile-Teacher	R, W	R, W	R
profile-Caregiver	R, W	R	R, W
profile-Student	R, W	R[*]	R[‡]
absences	R, W	R[*], W[*]	R
teachers	R, W	R	R
caregivers	R, W	R	R
students-Caregiver	R, W	R	R[‡]
students-Classes	R, W	R	R
message-Classchat	R, W	R[§], W[§]	R[§]

Tabelle 4.1.: Übersicht der Berechtigungen auf Dateien, sobald diese auf der *Platform* liegen.

[#] Bezieht sich auf jede *permissions*-Datei auf der *Platform*.
[‡‡] Die Datei kann zwar gelesen werden, das darin enthaltene otp ist jedoch nur für den *Admin* lesbar, da nur für ihn verschlüsselt.
[†] Der *Admin* könnte das *Registrationpackage* eines *Teachers* oder *Caregivers* verändern. Es wird jedoch angenommen, dass er dies nicht tut - siehe Angreifermodell in Abschnitt 5.1.2.
[*] Nur die *Teacher*, die den jeweiligen *Student* unterrichten.
[‡] Nur die *Caregiver*, die für den jeweiligen *Student* verantwortlich sind.
[§] Nur die *Teacher* bzw. *Caregiver* der *Students* der entsprechenden Klasse.

4.3. Nachrichtenaustausch

Der Austausch von Nachrichten ist eine zentrale Funktion einer Schulkommunikationslösung. Über diesen Kanal werden nicht nur Absenzen gemeldet, sondern auch sensible Themen kommuniziert. Daher ist es entscheidend, dass sowohl der Versand als auch der Empfang von Nachrichten unter *E2EE* erfolgt.

Die Kommunikationsarchitektur der Plattform ist strikt kontextbezogen. Mit Ausnahme des Klassenchats ist jeder Chat immer einem spezifischen *Student* zugeordnet. Standardmässig gibt es pro *Student* einen *Default-Chat*, an dem alle zugeordneten *Teachers* und *Caregivers* mit Lese- und Schreibrechten teilnehmen. Zusätzlich können *Teachers* und *Caregivers* Ad-hoc-Chats für den Austausch in kleineren Gruppen oder 1:1 initiieren.

Alle diese studentenbezogenen Chat-Verläufe, ob *Default-* oder Ad-hoc-Chat, werden als separate Unterordner im Verzeichnis des jeweiligen *Students* gespeichert. Der Name eines solchen Chat-Ordners wird zur eindeutigen Identifizierung in der Regel aus einer Konkatenation der uids der beteiligten Parteien gebildet. Chats, die diesen studentenbezogenen Kontext verlassen, etwa zwischen *Caregivers* untereinander oder zwischen *Students*, sind in dieser Spezifikation nicht vorgesehen.

Die aktuelle Spezifikation sieht keine dynamische Änderung der Teilnehmerliste eines bestehenden Chats vor. Soll ein Teilnehmer hinzugefügt oder entfernt werden, muss ein neuer Chat mit der angepassten Teilnehmerliste initiiert werden. Die Erweiterung um dynamische Chat-Mitgliedschaften bleibt zukünftigen Arbeiten vorbehalten.

Jeder Chat-Ordner enthält eine *parties*-Datei, welche die Mitgliedschaft und die Berechtigungen innerhalb dieses spezifischen Chats definiert. Sie legt fest, welche Parteien (uids) an der Konversation teilnehmen und wer von ihnen berechtigt ist, Nachrichten zu verfassen. Die eigentliche Durchsetzung der Leserechte für die Teilnehmer erfolgt jedoch pro Nachricht individuell durch die gezielte Verschlüsselung. Deren Struktur wird detailliert in Kapitel 5.1.8 beschrieben.

Im Falle des *Default-Chats* wird die initiale *parties*-Datei vom *Admin* erstellt und signiert. Bei Ad-hoc-Chats übernimmt dies die initiierende Partei.

Im Folgenden wird der Prozess exemplarisch für den Versand einer Nachricht durch einen *Caregiver* im *Default-Chat* und deren Empfang durch einen *Teacher* beschrieben. Dieser Ablauf gilt analog für alle anderen Chat-Konstellationen. Dabei wird jede Nachricht auch für den Absender selbst verschlüsselt, um zu gewährleisten, dass auch dieser den Zugriff auf seine gesendeten Nachrichten behält.

4.3.1. Nachrichtenversand

Abbildung 4.8 zeigt den konzeptionellen Ablauf beim Versenden einer Nachricht. Die sendende Partei, in diesem Fall ein *Caregiver*, liest zunächst aus der *parties*-Datei des entsprechenden Chat-Ordners, welche Parteien am Chat beteiligt sind. Anschliessend holt sie deren öffentliche Schlüssel aus der globalen *publickeys*-Datei (1). Diese Datei wurde im Rahmen des *Platform*-Setups (Abschnitt 4.2) etabliert und enthält die für die Verschlüsselung benötigten öffentlichen Schlüssel aller registrierten Nutzenden.

Anschliessend verfasst die Partei die zu übermittelnde Nachricht. Für diese Nachricht wird ein neuer symmetrischer Schlüssel generiert mit welchem die Nachricht selbst verschlüsselt wird. Der symmetrische Schlüssel selbst wird darauffolgend für jede empfangende Partei (inklusive der sendeten Partei) mit dem jeweiligen öffentlichen Schlüssel dieser Partei asymmetrisch verschlüsselt. Diese Sammlung von verschlüsselten symmetrischen Schlüsseln, zusammen mit den zugehörigen Empfänger-uids, bildet einen Teil der Nachrichtendatei. Die unverschlüsselten Metadaten der Nachricht (wie Absender, Empfänger-uids, Datum, Nachrichten-ID) sind ebenfalls Bestandteil der zu speichernden Datei.

Zur Sicherstellung der *Authenticity* und *Integrity* wird über die relevanten Teile der Nachricht (insbesondere die unverschlüsselten Metadaten und den Nachrichteninhalt) von der sendenden Partei mit ihrem privaten Signaturschlüssel eine Signatur erzeugt. Die resultierende digitale Signatur werden ebenfalls in der Nachrichtendatei abgelegt. Die vollständige Nachrichtendatei wird auf der *Platform* im jeweiligen Chat-Ordner gespeichert (2).

4.3.2. Nachrichtenempfang

Abbildung 4.9 illustriert den konzeptionellen Ablauf des Nachrichtenempfangs. Die Applikation der empfangenden Partei, hier ein *Teacher*, lädt neue Nachrichtendateien aus dem entsprechenden Chat-Ordner herunter.

Aus den unverschlüsselten Metadaten der Nachricht entnimmt die Applikation die uid der sendenden Partei und lädt deren öffentlichen Schlüssel aus der globalen *publickeys*-Datei (1).

Anschliessend wird die empfangene Nachrichtendatei verarbeitet. Die *Integrity* und *Authenticity* der Nachricht werden sichergestellt, indem die in der Nachricht enthaltene digitale Signatur mit dem zuvor geladenen öffentlichen Schlüssel des Senders überprüft wird.

Bei erfolgreicher Verifikation sucht die empfangende Partei in der Nachrichtendatei den für ihre eigene uid bestimmten, asymmetrisch verschlüsselten symmetrischen Schlüssel. Dieser wird mithilfe des eigenen privaten Schlüssels der empfangenden

Partei entschlüsselt, um den ursprünglichen symmetrischen Schlüssel wiederherzustellen. Mit diesem symmetrischen Schlüssel kann schliesslich der eigentliche Nachrichteninhalt entschlüsselt und der Klartext der Nachricht angezeigt werden (2). Schlägt einer dieser Schritte fehl (z.B. die Signaturprüfung), wird die Nachricht als ungültig betrachtet und verworfen.

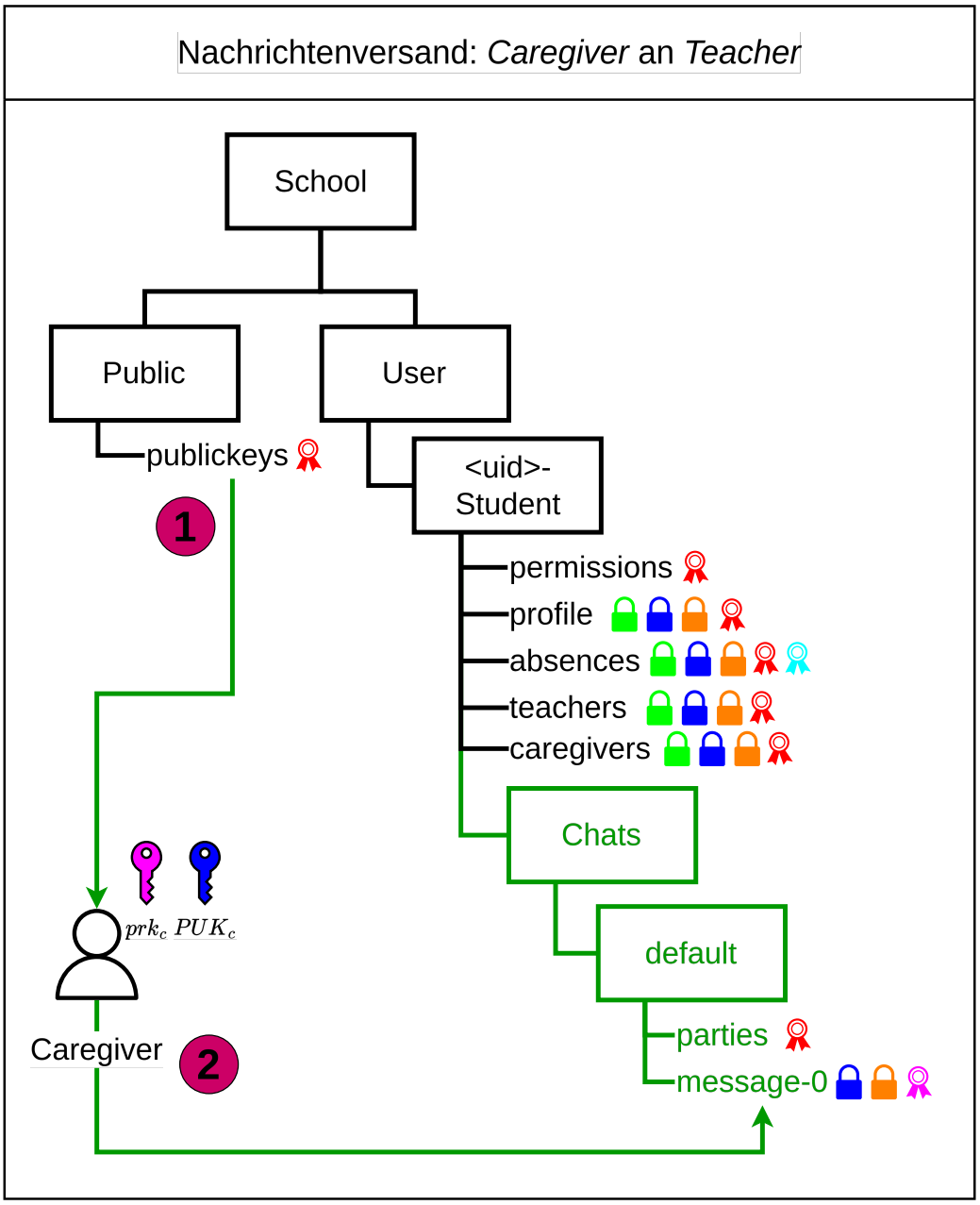


Abbildung 4.8.: Die sendende Partei holt die öffentlichen Schlüssel der Empfänger und legt die konstruierte Nachrichtendatei auf der *Platform* ab.

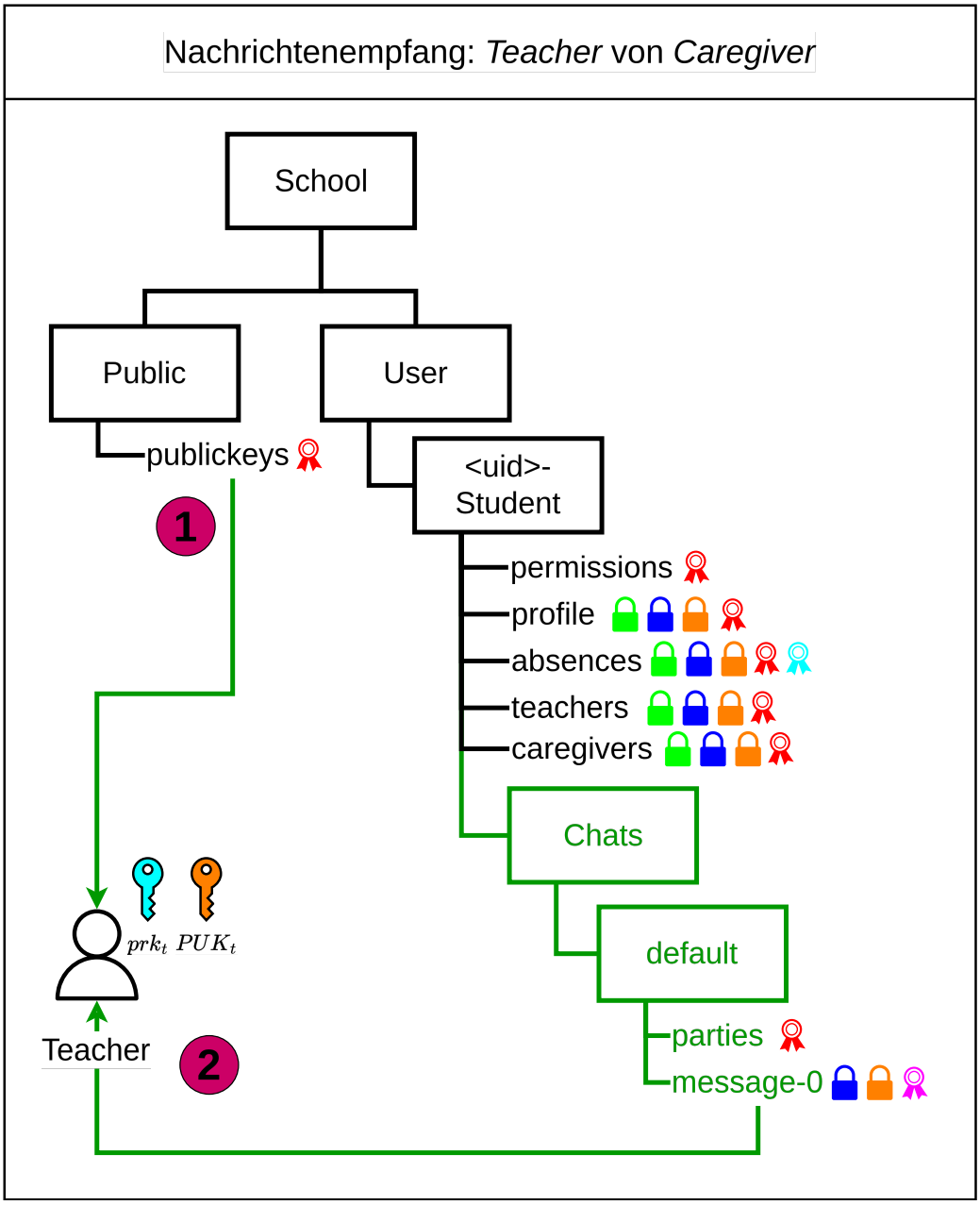


Abbildung 4.9.: Die empfangende Partei lädt die Nachrichtendatei sowie den öffentlichen Schlüssel des Senders, prüft die Signatur und entschlüsselt die Nachricht.

4.4. Abgrenzung zu bestehenden Plattformen

Im Gegensatz zu kommerziellen Schulkommunikationslösungen, wie sie in Abschnitt 1.2 analysiert wurden, wurde die hier beschriebene *Platform* von Anfang an unter Berücksichtigung der Prinzipien von *PbD* konzipiert. Ziel ist es nicht, eine neue Software zu vermarkten, sondern aufzuzeigen, dass datenschutzfreundliche Kommunikation im schulischen Umfeld technisch realisierbar ist.

4.4.1. Technische Absicherung statt rein organisatorischer Kontrolle

Während andere Schulkommunikationslösungen Daten zentral speichern und dabei häufig lediglich auf eine *at-rest*-Verschlüsselung setzen, also eine Absicherung der Festplatte oder Datenbank, schützt die in dieser Arbeit beschriebene Schulkommunikationslösung sämtliche Inhalte konsequent durch *E2EE* (vgl. Abschnitt 2.1.2). Bei der *at-rest*-Verschlüsselung liegen die Daten zwar verschlüsselt auf dem Speichermedium, sind jedoch prinzipiell für den Plattformbetreiber zugänglich, da dieser in der Regel auch die Schlüsselverwaltung kontrolliert.

Im Gegensatz dazu verarbeitet der Plattformbetreiber der hier beschriebenen Schulkommunikationslösung ausschliesslich verschlüsselte Inhalte und stellt die Infrastruktur für deren Speicherung bereit. Ihm sind einzig der Name der Schule sowie die Identität des *Admin* bekannt. Darüber hinaus hat er Zugriff auf wenige unverschlüsselte Dateien (vgl. Abschnitt 4.1.3), die jedoch keine personenbezogenen Informationen enthalten. Alle übrigen Daten liegen ihm ausschliesslich in nicht lesbarer, verschlüsselter Form vor.

4.4.2. Vermeidung zentraler Schwachstellen

Der Ansatz vermeidet, dass die *Platform* selbst zur Sicherheitslücke wird. Selbst bei einem Angriff auf die *Platform* bzw. den Plattformbetreiber bleiben die Inhalte für Unbefugte unlesbar. Dies schützt die Privatsphäre der Beteiligten selbst dann, wenn andere Schutzmechanismen versagen sollten.

4.4.3. Offenheit und Nachvollziehbarkeit

Die Spezifikation sowie der entwickelte *PoC* sind öffentlich zugänglich. So kann die Lösung unabhängig geprüft, verbessert und erweitert werden. Dies fördert Transparenz und Vertrauen in die technische Umsetzung.

4.5. Fazit

Die in diesem Kapitel präsentierte Konzeption einer datenschutzfreundlichen Schulkommunikationslösung demonstriert eindrücklich die praktische Umsetzbarkeit digitaler Kommunikation im Schulumfeld unter Einhaltung strenger Datenschutzanforderungen. Die realisierte Architektur stellt durch den konsequenten Einsatz von Verschlüsselung, digitalen Signaturen und klar definierten Zugriffsberechtigungen sicher, dass sensible Informationen auch bei einer Kompromittierung zentraler *Platform*-Komponenten geschützt bleiben und der Plattformbetreiber keinen Zugriff auf personenbezogene Inhalte erhält. Die Ausführung aller kryptographischen Operationen auf den Endgeräten der Nutzenden, bei gleichzeitigem Erhalt der Benutzerfreundlichkeit, unterstreicht die Machbarkeit. Die zentrale Erkenntnis dieses Kapitels ist somit, dass Sicherheit, Transparenz, Datenschutz und Alltagstauglichkeit in einer Schulkommunikationslösung vereinbar sind und das Prinzip von *PbD* im schulischen Kontext nicht nur ein theoretischer Anspruch, sondern eine technisch realisierbare Lösung darstellt, deren detaillierte Protokollspezifikation in Kapitel 5 folgt.

5. Protokoll Spezifikation

Dieses Kapitel beschreibt die detaillierte technische Spezifikation des Kommunikationsprotokolls, das der in Kapitel 4 vorgestellten *Platform* zugrunde liegt. Dabei wird auf die in Abschnitt 4.1.1 eingeführten Parteien Bezug genommen. Zunächst werden allgemeine Annahmen, das Bedrohungsmodell sowie die zugrunde liegenden Vertrauensannahmen erläutert. Anschliessend folgt die formale Definition der verwendeten kryptographischen Primitive, der Notation und der Datenstrukturen. Den Kern des Kapitels bilden die einzelnen Protokollabläufe, insbesondere für die Registrierung, das *Bootstrapping* und den Nachrichtenaustausch, die in Form von Protokollschritten und Pseudo-Code-Algorithmen dargestellt werden.

5.1. Protokoll-Design

Der Abschnitt zum Protokoll-Design bildet die Grundlage der Spezifikation. Er definiert das angenommene Angreifermodell sowie die daraus abgeleiteten Vertrauensannahmen, welche die sicherheitsrelevanten Rahmenbedingungen festlegen. Diese bilden die Basis für die spätere Ausgestaltung der kryptographischen Mechanismen und Abläufe.

5.1.1. Initialer Status der Plattform

Die nachfolgende Spezifikation geht davon aus, dass zwischen einer Schule und dem Betreibenden der *Platform* bereits ein Dienstleistungsvertrag unterzeichnet wurde. Der *Platform*-Betreibende hat die Ablagestruktur für die Schule wie in Abb. 4.2 initialisiert und dem *Admin* der Schule über einen sicheren Kanal (z. B. persönliche Übergabe oder eingeschriebener Brief) folgende Registrierungsinformationen zur Verfügung gestellt:

- Der *User Identifier Admin* (uid_a). Diese wird als einzige uid von der *Platform* generiert. Dies ist unproblematisch, da der Betreibende die Identität des *Admins* aufgrund des Vertragsabschlusses bereits kennt.
- Ein otp, das sicherstellt, dass sich nur der vorgesehene *Admin* registrieren kann – und dies nur einmalig.

Zur Überprüfung des otp bei der Registrierung speichert die *Platform* ausschliesslich dessen kryptographischen Hashwert. Der Klartext des otp wird nicht gespeichert,

um zu verhindern, dass sich eine Person mit Zugriff auf die *Platform* – unabhängig davon, ob dieser Zugriff legitim ist oder nicht – unrechtmässig als *Admin* einer Schule registrieren kann.

Diese Informationen werden dem *Admin* in einem geeigneten Format übermittelt. Vorzugsweise werden sie in einem QR-Code kodiert, sodass der *Admin* sie mithilfe des $client_a$ einlesen kann.

5.1.2. Angreifermodell

Ziel dieses Angreifermodells ist es, Bedrohungen zu klassifizieren, die auf die *Confidentiality*, *Integrity* und *Authenticity* der auf der *Platform* gespeicherten Daten abzielen. Angriffe durch *Social Engineering* auf am Schulalltag beteiligte Personen (*Admins*, *Teachers*, *Caregivers* und *Students*) werden in diesem Modell bewusst ausgeschlossen, um den Fokus auf technische Bedrohungen zu legen. Weiter wird von einem realistischen Angreifermodell ausgegangen, das ausschliesslich probabilistisch polynomiell beschränkte Angreifende berücksichtigt, also solche mit begrenzten Rechenressourcen. Diese sind nicht in der Lage, als hart geltende Probleme wie das *DL*- oder *DDH*-Problem effizient zu lösen.

In dieser Arbeit werden drei Typen von Angreifenden unterschieden, die sich durch unterschiedliche Motivationen und Fähigkeiten auszeichnen:

1. **Interne Angreifende:** Darunter fallen Personen, die aktiv am Schulalltag beteiligt sind, wie *Caregivers* und *Teachers*. *Admins* werden als Angreifende ausgeschlossen, da sie unabhängig von einer Schulkommunikationslösung umfassenden Zugriff auf schulinterne Daten besitzen. Interne Angreifende verfügen über bestimmte Rechte innerhalb der *Platform* und könnten versuchen, diese auszuweiten oder Schwachstellen im Systemdesign auszunutzen, um unbefugt auf Informationen zuzugreifen oder solche zu manipulieren. Die Motivationen dieser Angreifenden sind vielfältig, im Folgenden werden exemplarisch einige mögliche Szenarien dargestellt:

   - Ein *Caregiver* könnte an Informationen wie Absenzen, Chatverläufen oder Noten eines benachbarten *Student* interessiert sein.
   - Ein *Caregiver* könnte ein Zurücksetzen des Absenzenzählers eines *Student* anstreben.
   - Ein *Caregiver* könnte versuchen, Nachrichten eines *Teacher* zu manipulieren, um sich selbst oder einem *Student* einen Vorteil zu verschaffen.
   - Analog könnte ein *Teacher* versuchen, eine Nachricht eines *Caregivers* zu verändern.

   Diese Beispiele verdeutlichen, dass interne Angreifende trotz eingeschränkter Rechte ein erhebliches Bedrohungspotenzial darstellen.

2. **Administrative Angreifende:** Diese Gruppe umfasst Personen mit weitreichenden administrativen Rechten auf der *Platform*, etwa Systemadministrierende oder andere technisch befugte Mitarbeitende des Plattformbetreibenden. Sie verfügen in der Regel über uneingeschränkten Zugriff auf sämtliche auf der *Platform* gespeicherten Daten. Ein potenzieller Missbrauch dieser privilegierten Stellung könnte darin bestehen, personenbezogene Daten unbefugt auszulesen und etwa zu kommerziellen Zwecken an Dritte weiterzugeben. Im Fall einer Doppelrolle als interne Angreifende könnten sie zudem gezielt Informationen über andere *Students* oder *Caregivers* abgreifen und ausnutzen.

3. **Externe Angreifende:** Diese Kategorie umfasst Personen ausserhalb des schulischen Kontexts sowie ausserhalb der *Platform*, die daran interessiert sind, unbefugt Zugang zu schulischen Daten zu erlangen. Externe Angreifende verfügen typischerweise über keine internen Berechtigungen, sind aber oft besonders motiviert und kreativ in der Ausnutzung von Fehlkonfigurationen, Schwachstellen in der Softwarearchitektur oder menschlichem Fehlverhalten. Sie können verschiedene Angriffsvektoren ausnutzen, etwa schwach gesicherte Administrationskonten, Sicherheitslücken in der Software- oder Hardwarearchitektur der *Platform* oder unzureichend geschützte, öffentlich zugängliche Cloud-Speicher[1]. Zudem könnten Einladungsschreiben beim Versand abgefangen und manipuliert werden, wodurch ein *Man-in-the-Middle*-Angriff ermöglicht wird. Ebenso können Datenlecks aus früheren Angriffen durch externe oder interne Angreifende genutzt werden. Die kompromittierten Daten könnten zur Erstellung detaillierter Persönlichkeitsprofile betroffener Personen verwendet werden, die wiederum als Grundlage für *Social Engineering*, Identitätsmissbrauch oder gezielte Rufschädigung dienen können.

5.1.3. Sicherheitsziele

Das zuvor beschriebene Angreifermodell zeigt, dass die auf der *Platform* gespeicherten Daten vielfältigen Bedrohungen ausgesetzt sind, sowohl durch interne, externe als auch administrative Angreifende. Daraus ergibt sich die Notwendigkeit, zentrale Schutzziele wie *Confidentiality*, *Integrity* und *Authenticity* konsequent zu adressieren. Diese klassischen Ziele werden durch die ergänzenden Schutzziele *Privacy* und *Accountability* erweitert, um auch dem Missbrauch durch privilegierte Instanzen sowie datengetriebenen Angriffen (z. B. Profilbildung) angemessen zu begegnen.

Auf Grundlage dieser Bedrohungslage gelten für die *Platform* die folgenden Sicherheitsziele:

1. *Confidentiality*: Nur autorisierte Parteien dürfen Zugriff auf Daten erhalten, die auf der *Platform* gespeichert sind.

[1] Ein Beispiel für ein solches Datenleck stellt der Vorfall bei Volkswagen dar, siehe [30].

2. *Integrity*: Änderungen an diesen Daten dürfen ausschliesslich durch berechtigte Parteien erfolgen. Manipulationen müssen verhindert oder zuverlässig erkannt werden.
3. *Authenticity*: Die Herkunft und Unverfälschtheit gespeicherter Daten müssen überprüfbar und nachvollziehbar sein.
4. *Privacy*: Die Erhebung und Verarbeitung personenbezogener Daten ist auf das erforderliche Minimum zu beschränken. Die technische Erschwerung oder Verhinderung von Profilbildung ist explizites Ziel.
5. *Accountability*: Alle sicherheitsrelevanten Vorgänge müssen protokolliert werden, um Missbrauch aufzudecken und Verantwortlichkeiten eindeutig zuordnen zu können.

5.1.4. Vertrauensannahmen

Zur Einordnung der Bedrohungslage müssen bestimmte Voraussetzungen über die Umwelt und die Nutzung der *Platform* getroffen werden. Diese betreffen insbesondere schulinterne Rollen, Endgeräte, Kommunikationskanäle sowie infrastrukturelle Komponenten. Sie sind notwendig, um die zuvor definierten Sicherheitsziele realistisch erreichen zu können, und markieren zugleich die Grenzen des hier betrachteten Bedrohungsmodells:

- **Verhalten schulinterner Rollen:** Zwar wird berücksichtigt, dass schulinterne Rollen wie *Caregivers* oder *Teachers* versuchen könnten, unberechtigten Zugriff auf Daten zu erlangen, solche Fälle sind explizit Teil des Angreifermodells (Abschnitt 5.1.2). Es wird jedoch davon ausgegangen, dass sie ihre legitimen Zugriffsrechte nicht vorsätzlich missbrauchen, etwa zur Veröffentlichung oder Weitergabe sensibler Informationen. Solche Verstösse gegen berufliche oder gesetzliche Pflichten können durch eine Schulkommunikationslösung nicht allein technisch verhindert werden.
- **Integrity der Endgeräte:** Die verwendeten Endgeräte der Nutzenden (z. B. Laptops, Smartphones) gelten als nicht kompromittiert. Ist dies nicht der Fall, kann analog zu hochsicherheitskritischen Anwendungen wie E-Voting [31, Kap. 6.2], keine *E2EE* und damit auch keine *Confidentiality* gewährleistet werden.
- **Vertrauenswürdige Initialkommunikation:** Für die Übermittlung der Einladungsschreiben an *Caregivers* und *Teachers* wird ein sicherer Übertragungskanal vorausgesetzt (z. B. persönliche Übergabe). So soll verhindert werden, dass Angreifende bereits im ersten Schritt durch gezielte Manipulation, etwa die Verteilung falscher Schlüssel, die spätere Registrierung kompromittieren.
- **Unverfälschte Druckinfrastruktur:** Die Systeme zur Erzeugung und zum Druck der Einladungsschreiben (einschliesslich Drucker und angeschlossener Rechner) gelten als integer. Eine Manipulation in diesem frühen Schritt könnte da-

zu führen, dass gefälschte oder abgefangene Zugangsinformationen verteilt werden. Da diese Informationen die Grundlage für die sichere Registrierung bilden, ist die *Integrity* der Druckinfrastruktur von besonderer Bedeutung.

- **Verlässlichkeit des Entwicklungsprozesses:** Es wird angenommen, dass der Quellcode öffentlich zugänglich ist und dass gezielte Manipulationen, etwa das heimliche Deaktivieren von Verschlüsselung, dadurch auffallen würden. Angriffe auf den Build-Prozess oder gezielte Supply-Chain-Angriffe werden in diesem Modell nicht berücksichtigt.
- **Abgrenzung zu operativen Sicherheitsmassnahmen:** Die Arbeit fokussiert sich auf konzeptionelle und kryptographische Schutzmechanismen. Operative Massnahmen zur Sicherstellung der Verfügbarkeit (z. B. Schutz vor DoS-Angriffen, Redundanz) werden nicht betrachtet und liegen ausserhalb des gewählten Bedrohungsmodells.

5.1.5. Kryptographische Primitive

Basierend auf den definierten Sicherheitszielen werden in diesem Abschnitt die kryptographischen Primitive spezifiziert. Dazu gehören Verfahren zur symmetrischen und asymmetrischen Verschlüsselung, digitale Signaturen sowie kryptographische Hashfunktionen. Die konkreten Schematas zu den folgenden Primitven werden im Abschnitt 5.1.7 beschrieben.

Symmetrische Verschlüsselung

Bei der symmetrischen Verschlüsselung wird ein geheimer Schlüssel sowohl zur Ver- als auch zur Entschlüsselung verwendet. Dieser wird in der Regel als *Secret Key* (sk) bezeichnet. Um neben der *Confidentiality* auch die *Integrity* und die *Authenticity* sicherzustellen, wird der Einsatz eines *Authenticated Encryption with Associated Data* (*AEAD*)-Verfahrens gefordert.

Ein solches Verfahren erzeugt beim Verschlüsseln zusätzlich einen *Tag*, der bei der Entschlüsselung erneut berechnet und mit dem übermittelten Wert verglichen wird. Stimmen die Werte überein, kann der Empfänger sicher sein, dass die Nachricht nicht manipuliert wurde, dies entspricht dem *Authenticated Encryption* (*AE*)-Anteil des *AEAD*-Konzepts und sorgt für die *Integrity*. Ausserdem bietet *AEAD* die Möglichkeit, zusätzliche, unverschlüsselte Daten (*Associated Data* (*AD*)) in den *Tag* zu integrieren. Dabei handelt es sich häufig um Metadaten wie sendende oder empfangende Partei. Diese Informationen werden zwar nicht verschlüsselt, aber mitauthentifiziert und dienen dazu, die *Authenticity* einer verschlüsselten Nachricht sicherzustellen [22, Kap. 5.2].

Asymmetrische Verschlüsselung

Bei der asymmetrischen Verschlüsselung kommen ein *Public Key* (PUK) und ein *Private Key* (prk) zum Einsatz. Nachrichten werden mit dem PUK verschlüsselt und können nur mit dem zugehörigen prk entschlüsselt werden.

Als Sicherheitsanforderung genügt im vorliegenden Protokoll *Indistinguishability under Chosen Plaintext Attack* (*IND-CPA*)-Sicherheit. Sie garantiert, dass ein Angreifender - selbst wenn er beliebige Klartexte verschlüsseln kann - nicht zwischen zwei ausgewählten Klartexten anhand ihrer Chiffrate unterscheiden kann. Formal bedeutet dies, dass ein solcher Angreifer in einem *IND-CPA*-Experiment mit einer Wahrscheinlichkeit von höchstens 50 % erraten kann, welcher der beiden Klartexte verschlüsselt wurde. Praktisch entspricht dies einer rein zufälligen Auswahl, sodass keine verwertbare Information über den verschlüsselten Inhalt gewonnen werden kann, womit die *Confidentiality* sichergestellt ist.

Ein stärkeres Schutzniveau wie *Indistinguishability under Adaptive Chosen Ciphertext Attack* (*IND-CCA2*) ist im vorliegenden Fall nicht erforderlich, da nahezu alle verschlüsselten Nachrichten zusätzlich digital signiert sind und dadurch bereits vor Manipulation und aktiven Angriffen geschützt werden.

Einzige Ausnahme bildet das Registrierungspaket, das von den *Caregivers* auf der *Platform* abgelegt wird. Zwar könnte ein potenzieller Angreifender hier auf ein Entschlüsselungsorakel in Form des $client_a$ zugreifen, jedoch bietet dies keinen praktischen Vorteil, denn wird das Paket verändert, wird es vom $client_a$ abgewiesen, ohne verwertbare Rückmeldung zu geben. Für weiterführende Informationen zur Sicherheit gegenüber Ver- und Entschlüsselungsorakeln sei auf [22, Kap. 12.2] verwiesen.

Digitale Signaturen

Digitale Signaturen ermöglichen es, die *Authenticity* und *Integrity* von Nachrichten zu prüfen. Dazu wird eine Nachricht mit dem *Signing Key* (sik) signiert und kann mit dem zugehörigen *Verification Key* (VK) überprüft werden.

Um die *Authenticity* von Nachrichten und Identitäten sicherzustellen, wird ein Signaturverfahren mit *Existential Unforgeability under Chosen Message Attacks* (*EUF-CMA*)-Sicherheit vorausgesetzt. Diese garantiert, dass es selbst dann nicht möglich ist, eine gültige Signatur für eine neue Nachricht zu erzeugen, wenn ein Angreifender zuvor Signaturen zu beliebigen anderen Nachrichten erhalten hat [22, Definition 13.2].

Kryptographische Hashfunktionen

Hashfunktionen werden eingesetzt, um beliebig grosse Eingaben in kurze, eindeutige Ausgaben (*digest*s) zu überführen. Sie dienen unter anderem zur Integritätsprü-

fung, zur Ableitung von Schlüsseln und zur Signaturerstellung. Damit diese Aufgaben sicher erfüllt werden können, muss eine Kryptographische Hashfunktion insbesondere die Folgenden Sicherheitsziele erfüllen.

Erstens ist *Preimage-resistance* erforderlich. Für einen gegebenen Hashwert h soll es praktisch unmöglich sein, eine Eingabe x zu finden, sodass $hash(x) = h$ gilt. Zweitens muss die *Second-preimage-resistance* gewährleistet sein. Für eine gegebene Eingabe x soll es praktisch unmöglich sein, eine zweite Eingabe $x' \neq x$ zu finden, sodass $hash(x') = hash(x)$. Drittens ist die *Collision-resistance* entscheidend. Es soll praktisch unmöglich sein, zwei beliebige Eingaben $x \neq x'$ zu finden, für die $hash(x) = hash(x')$ gilt [22, Kap. 6].

Key Derivation Function (KDF)

In verschiedenen Protokollschritten müssen sichere Schlüssel aus Passwörtern, zufälligen Seeds oder gemeinsamen Geheimnissen abgeleitet werden - etwa zur Erzeugung des usk aus dem us. Dazu wird eine *Key Derivation Function* (*KDF*) eingesetzt, die sicherstellt, dass selbst bei schwachen Eingaben keine Rückschlüsse auf den Ursprung möglich sind und unterschiedliche Kontexte unterschiedliche Schlüssel liefern.

Damit eine *KDF* als sicher gilt, muss sie insbesondere drei Anforderungen erfüllen. Erstens muss die *Entropy preservation* gewährleistet sein, das heisst, die resultierenden Schlüssel sollten mindestens die Entropie des Eingabematerials widerspiegeln und dürfen nicht schwächer sein. Zweitens ist eine *Domain separation* erforderlich - unterschiedliche Anwendungen oder Protokolle sollten bei identischem Eingabematerial unterschiedliche Schlüssel ableiten. Drittens muss die *Brute-force resistance* gegeben sein, sodass es selbst bei Kenntnis eines abgeleiteten Schlüssels schwierig ist, auf den ursprünglichen *Seed* oder das Passwort zu schliessen, insbesondere durch den Einsatz von *Salt* oder hohen Iterationszahlen [22, Kap. 6.6.4].

5.1.6. Notation und Variablen

Dieser Abschnitt definiert die in der vorliegenden Spezifikation verwendete Notation, Symbolik sowie konventionelle Schreibweisen.

Typografische Konventionen

Funktionen und Algorithmen (beispielsweise GenKeyPairs()) werden in serifenloser Schrift dargestellt und bestehen aus einem oder mehreren Wörtern, die jeweils mit einem Grossbuchstaben beginnen (CamelCase). Eine Ausnahme bilden sogenannte Black-Box-Algorithmen (beispielsweise enc_a() oder store_file()). Diese werden zwar

eingeführt, jedoch nicht im Pseudocode definiert, da ihre konkrete Implementierung je nach verwendeter Programmiersprache und primitiver Konstruktion (z. B. *RSA*, *AES-GCM*, *PSS*) variiert. Um sie visuell von anderen Algorithmen abzuheben, werden sie durchgehend kleingeschrieben und Wörter mit einem Unterstrich (_) verbunden dargestellt (snake_case). Variablennamen wie uid, PUK und prk werden hingegen in Serifenschrift geschrieben, um eine klare Abgrenzung zwischen Funktionen und Variablen zu gewährleisten.

Allgemeine Tupel werden durch kleingeschriebene lateinische oder griechische Buchstaben in Serifenschrift dargestellt ($x = (y, z)$ für ein Tupel aus $Y \times Z$). In bestimmten Fällen kommen Grossbuchstabenpaare wie RP zum Einsatz, z.B. wenn das Tupel semantische Informationen wie Nutzerdaten enthält.

Zur Darstellung von Listen gleichartig strukturierter Tupel wird die folgende generische Notation verwendet:

$$x = ((t_{1,1}, \ldots, t_{1,k}), \ldots, (t_{n,1}, \ldots, t_{n,k})) \in (T_1 \times \cdots \times T_k)^n,$$

wobei n die Länge der Liste und $k \geq 2$ die Anzahl der Einträge pro Tupel bezeichnet. Jedes Tupel $(t_{i,1}, \ldots, t_{i,k})$ ist dabei ein Element des kartesischen Produkts $T_1 \times \cdots \times T_k$ für $1 \leq i \leq n$.

Diese Notation macht explizit, dass es sich um eine geordnete Liste (oder Sequenz) von n Tupeln identischer Struktur handelt, unabhängig davon, wie viele Komponenten ein einzelnes Tupel enthält.

Zur besseren Lesbarkeit wird insbesondere bei kurzen Tupeln (z. B. $k = 2$ oder $k = 3$) die explizite Ausformulierung

$$x = ((a_1, b_1), \ldots, (a_n, b_n))$$

verwendet, wobei $x \in (A \times B)^n$.

Analog dazu bezeichnet die Notation T^n die Menge aller geordneten Listen der Länge n, deren Einträge jeweils Elemente aus T sind. Beispielsweise steht $(\mathsf{uid})^n \in (\mathcal{U})^n$ für eine Liste von n Benutzeridentifikationsnummern, wobei $\mathcal{U}$ den Raum gültiger uids beschreibt.

Kryptographische Ergebnisse

In der vorliegenden Spezifikation wird das Präfix c verwendet, um anzuzeigen, dass ein *String* asymmetrisch verschlüsselt wurde. Die Notation $\langle\ \rangle$ kennzeichnet eine digitale Signatur über einen *String*, während [ ] das Ergebnis einer *AEAD*-Verschlüsselung darstellt. Die Kombination $\langle[\textit{String}(str)]\rangle$ beschreibt einen *String*, der mittels *AEAD* verschlüsselt und anschliessend signiert wurde. Tabelle 5.1 veranschaulicht diese Notationen anhand entsprechender Beispiele.

Zusätzlich gilt, dass $\langle str \rangle$ ein Tupel bestehend aus einem *String* (*Plaintext*) und der zugehörigen Signatur sig repräsentiert. Die Notation $[str]$ steht für ein *AEAD*-Verschlüsselungsergebnis, d. h. ein Tupel bestehend aus *Nonce*, *Associated Data*, *Ciphertext* und *Tag*. Die geschachtelte Darstellung $\langle [str] \rangle$ steht entsprechend für ein signiertes *AEAD*-Ergebnis:

$$((\textit{Ciphertext},\ \textit{Tag},\ \textit{Nonce},\ \textit{AD}),\ sig)\,,$$

wobei sig die Signatur über das *AEAD*-Tupel darstellt.

Symbol	Beispiel	Beschreibung
c	$cString$	Präfix für einen *String*, der mit einem öffentlichen Schlüssel asymmetrisch verschlüsselt wurde
$\langle\ \rangle$	$\langle str \rangle$	Digitale Signatur über einen *String*. Ergebnis ist das Tupel $(str,\ sig)$
$[\ ]$	$[str]$	Ergebnis der *AEAD*-Verschlüsselung: Tupel aus *Nonce*, *Associated Data*, *Ciphertext* und *Tag*
$\langle [\ ] \rangle$	$\langle [str] \rangle$	Geschachtelte Operation: str wurde *AEAD*-verschlüsselt und anschliessend signiert

Tabelle 5.1.: Notation kryptographischer Ergebnisse

Innerhalb der Algorithmen folgen die Variablennamen einer pragmatischen Konvention. So wird beispielsweise der *AEAD-Tag* einer Variable x als tag_x bezeichnet, das *Ciphertext*-Äquivalent einer Klartextvariablen x als ct_x sowie entsprechend $nonce_x$, ad_x usw. Diese Konvention dient der Klarheit, insbesondere bei mehrfacher Verwendung von *AEAD*-Verschlüsselungen innerhalb eines Algorithmus. In Fällen, in denen nur eine einzelne *AEAD*-Verschlüsselung auftritt und keine Verwechslungsgefahr besteht, kann auf das Subscript verzichtet werden. Sobald Variablen in die Notation $[\ ]$ eingebettet werden, gelten die oben spezifizierten kryptographischen Bedeutungen.

Schlüssel

In der vorliegenden Spezifikation werden unterschiedliche Schlüssel zur asymmetrischen (prk, PUK) und symmetrischen Verschlüsselung (usk, dsk, fsk) sowie zum Erstellen und Verifizieren von Signaturen (sik, VK) verwendet. Grundsätzlich werden öffentliche Schlüssel in Grossbuchstaben und private in Kleinbuchstaben dargestellt. Tabelle 5.2 listet alle in dieser Spezifikation verwendeten Schlüssel auf und beschreibt ihre jeweilige Funktion. Diese Schlüssel sind zudem im Symbolverzeichnis aufgeführt.

Sofern der genaue Typ des symmetrischen Schlüssels (usk, dsk oder fsk) für das Verständnis nicht entscheidend ist, wird in dieser Spezifikation der Schlüssel sk als Platzhalter verwendet.

Kürzel	Beschreibung
$PUK_i,\ i \in \{a,t,c\}$	Öffentlicher Schlüssel einer spezifischen Person i. Wird verwendet, um *File Secret Key* (fsk) zu verschlüsseln.
$prk_i,\ i \in \{a,t,c\}$	Privater Schlüssel einer spezifischen Person i. Wird verwendet, um fsk zu entschlüsseln.
$VK_i,\ i \in \{a,t,c\}$	Verifikationsschlüssel einer spezifischen Person i. Dient zur Überprüfung der Signatur einer Datei.
$sik_i,\ i \in \{a,t,c\}$	Signaturschlüssel einer spezifischen Person i. Wird verwendet, um eine Datei zu signieren.
usk	*User Secret Key*, abgeleitet vom us. Wird verwendet, um den prk zu verschlüsseln.
dsk	*Device Secret Key* ist ein gerätespezifischer Schlüssel (z. B. ein *Passkey* im Sinne von WebAuthn/FIDO2). Kann verwendet werden, um den prk zu verschlüsseln.
fsk	*File Secret Key* wird verwendet, um einzelne Dateien zu verschlüsseln.
msk	*Message Secret Key* wird verwendet, um einzelne Nachricht zu verschlüsseln.

Tabelle 5.2.: In der Spezifikation verwendete Schlüssel

Zufallswerte

Die Auswahl eines Wertes r als gleichverteilten Bitstring der Länge ℓ aus dem *Space* $\{0,1\}^\ell$ wird mit der Notation

$$r \xleftarrow{\$} \{0,1\}^\ell$$

ausgedrückt. Die konkrete Quelle der Zufallswerte (z. B. ein *Pseudorandom Number Generator* (*PRNG*)) wird im Kontext des *PoC* spezifiziert.

Sicherheitsparameter

Der Sicherheitsparameter $(\lambda) \in \mathbb{N}$ definiert das angestrebte Sicherheitsniveau eines kryptographischen Verfahrens. Er gibt an, wie viele Bit ein Angreifer mindestens erraten oder berechnen müsste, um das Verfahren erfolgreich zu brechen - typischerweise durch Brute-Force-Angriffe. Ein Verfahren mit Sicherheitsniveau λ soll daher mindestens 2^λ Operationen erfordern, um gebrochen zu werden.

Die konkrete Bitlänge kryptographischer Elemente wie Schlüssel, Hashwerte, Nonces oder Zufallswerte hängt vom verwendeten kryptographischen Primitive ab: So

benötigt etwa eine kollisionsresistente Hashfunktion eine Ausgabelänge von mindestens $2 \cdot \lambda$ Bit, während bei symmetrischer Verschlüsselung bereits Schlüssel der Länge λ genügen.

In dieser Spezifikation wird λ als globale Konstante verwendet und auf

$$\lambda = 128$$

gesetzt. Dies entspricht einem Sicherheitsniveau von 2^{128} und folgt damit gängigen kryptographischen Empfehlungen (z. B. [32]).

Strings

Diese Spezifikation verwendet Strings, beispielsweise als Eingabe für Verschlüsselungsfunktionen, Signaturalgorithmen oder Hashfunktionen. Ein String wird als eine beliebig lange Aneinanderreihung von Zeichen aus dem UTF-8-Zeichensatz verstanden, wobei das zugrunde liegende Alphabet mit Σ bezeichnet wird. Die Menge aller möglichen Strings ist damit $\Sigma^\star$.

Das Zeichenalphabet Σ kann kontextabhängig eingeschränkt sein, etwa auf Base64- oder Hex-Zeichen, wenn Strings zur Übertragung oder Kodierung verwendet werden. Zeichenketten, die in Anführungs- und Schlusszeichen stehen, gelten in dieser Spezifikation als Strings im oben beschriebenen Sinne.

User Identifier

Der *User Identifier* ist ein pseudorandom erzeugter, gleichverteilter Identifier, dessen Entropiegehalt dem gewählten Sicherheitsniveau λ entspricht. Formal wird die uid als Zeichenkette aus dem Raum

$$\mathcal{U} := \Sigma^\star$$

modelliert. Das verwendete Alphabet Σ ist in diesem Fall typischerweise auf ein Kodierungsalphabet (z. B. Base64 oder Hex) beschränkt. Die uid dient als eindeutiger Bezeichner eines Nutzenden innerhalb des Systems.

One Time Password

Das *One Time Password* (otp) ist ein pseudorandom erzeugter, gleichverteilter Code, dessen Entropiegehalt dem gewählten Sicherheitsniveau λ entspricht. Formal wird das otp als Zeichenkette aus dem Raum

$$\mathcal{O} := \Sigma^\star$$

modelliert. Das verwendete Alphabet Σ ist in diesem Fall typischerweise auf ein Kodierungsalphabet (z. B. Base64 oder Hex) beschränkt. Das otp dient als einmalige eindeutige Authentifizierung eines Nutzenden innerhalb des Systems.

Keystore Space

Der in Abschnitt 5.1.8 im Detail beschriebene *Keystore* (KS) wird in dieser Arbeit durch den sogenannten *Keystore-Space* ($\mathcal{KS}$) formalisiert. Dieser beschreibt den Raum aller möglichen Keystore-Daten eines Nutzenden. Der $\mathcal{KS}$ ist definiert als:

$$\mathcal{KS} := \mathcal{U} \times (\Sigma^\star \times \mathcal{R} \times (\mathcal{N} \times \mathcal{A} \times \mathcal{C} \times \mathcal{T}) \times (\mathcal{N} \times \mathcal{A} \times \mathcal{C} \times \mathcal{T}))^n$$

Jedes Element des $\mathcal{KS}$ repräsentiert einen vollständigen Keystore-Eintrag eines Nutzenden, wie in Tabelle 5.5 strukturell beschrieben.

Message Identifier

Der *Message Identifier* ist ein pseudorandom erzeugter, gleichverteilter Identifier, dessen Entropiegehalt dem gewählten Sicherheitsniveau λ entspricht. Formal wird die *Message Identifier* (mid) als Zeichenkette aus dem Raum

$$\mathcal{M} := \Sigma^\star$$

modelliert. Das verwendete Alphabet Σ ist in diesem Fall typischerweise auf ein Kodierungsalphabet (z. B. Base64 oder Hex) beschränkt. Die mid dient als eindeutiger Bezeichner einer Nachricht innerhalb des Systems.

5.1.7. Kryptographische Schemata

Dieser Abschnitt definiert verschiedene kryptographische Schemata in formaler Weise. Dazu zählen symmetrische und asymmetrische Verschlüsselungsschemata, digitale Signaturschemata, Kryptographische Hashfunktionen sowie *KDF*s. Für jedes Schema werden die relevanten *Space*s sowie die zugehörigen Operationen präzise beschrieben.

Unabhängig vom konkreten Schema gelten die folgenden allgemeinen *Space*s.

$$\begin{aligned} \mathcal{P} &:= \Sigma^\star \quad \textit{plaintext space} \\ \mathcal{C} &:= \{0,1\}^\star \quad \textit{ciphertext space} \end{aligned}$$

Schemabezogene *Space*s und Parameter werden jeweils lokal innerhalb der entsprechenden Definition eingeführt.

Die Sicherheitsanforderungen an die hier beschriebenen Schemata wurden in Abschnitt 5.1.5 definiert. Die Struktur und Terminologie, der folgenden Schematas orientieren sich, sofern nicht anders vermerkt, an [33], [34] sowie [35].

Symmetrisches AEAD-Verschlüsselungsschema

Ein *AEAD*-fähiges symmetrisches Verschlüsselungsschema definiert, nebst $\mathcal{P}$ und $\mathcal{C}$ die folgenden *Space*s:

$$\begin{aligned} \mathcal{K} &:= \{0,1\}^\lambda && \textit{key space} \\ \mathcal{T} &:= \{0,1\}^\lambda && \textit{tag space} \\ \mathcal{N} &:= \{0,1\}^\lambda && \textit{nonce space} \\ \mathcal{A} &:= \Sigma^\star && \textit{associated data space} \end{aligned}$$

Bei der Entschlüsselung steht das Symbol *Perp* ($\perp$) für ein negatives Verifikationsergebnis, d. h. eine abgelehnte oder ungültige Eingabe. Aufbauend auf den oben definierten *Space*s umfasst das Schema drei randomisierte, in polynomieller Zeit berechenbare Algorithmen:

- **Schlüsselgenerierung[2]:**
$$sk \leftarrow \text{gen\_key}()$$

- **Verschlüsselung:**
$$(c, t) := \text{enc\_s}(sk, n, a, p)$$

- **Entschlüsselung:**
$$\text{dec\_s}(sk, n, a, c, t) = \begin{cases} p, & \text{falls } t \text{ gültig ist,} \\ \perp, & \text{sonst.} \end{cases}$$

Dabei gelten folgende Definitionsbereiche:

$$sk \in \mathcal{K}, \quad p \in \mathcal{P}, \quad n \in \mathcal{N}, \quad a \in \mathcal{A}, \quad c \in \mathcal{C}, \quad t \in \mathcal{T}$$

Für die Korrektheit des Schemas muss ausserdem gelten:

$$\text{dec\_s}(sk, n, a, \text{enc\_s}(sk, n, a, p), t) = p$$

für alle Eingaben aus den zuvor angegebenen Definitionsbereichen.

Es muss insbesondere sichergestellt werden, dass sich die verwendete *Nonce* bei einem AEAD-Schema unter keinen Umständen wiederholt. Eine Wiederverwendung derselben Nonce mit dem gleichen Schlüssel führt zu einer deterministischen Verschlüsselung sowie Tag-Generierung. Dies verletzt die geforderte *IND-CPA*-Sicherheit, da ein Angreifer dadurch Rückschlüsse auf den verwendeten *Plaintext* ziehen kann.

[2] In diesem Abschnitt steht sk stellvertretend für einen symmetrischen Schlüssel aus der Tabelle 5.2, beispielsweise den fsk.

In vielen Implementierungen (z. B. *Advanced Encryption Standard – Galois/Counter Mode* (*AES-GCM*) in pyca/cryptography) wird der *Tag* implizit an den *Ciphertext* angehängt. Zur Verbesserung der Klarheit und Modularität wird im vorliegenden Schema jedoch explizit zwischen *Ciphertext* und *Tag* unterschieden.

Die Definition des *AEAD*-Schemas wurde bewusst an standardisierte Schemata angepasst. Eine detaillierte theoretische Beschreibung findet sich in [22, Kap. 5.3].

Asymmetrisches Verschlüsselungsschema

Ein asymmetrisches Verschlüsselungsschema definiert, nebst $\mathcal{P}$ und $\mathcal{C}$ die folgenden *Spaces*[3]:

$$\begin{aligned} \mathcal{K}_{PUK} &:= \{0,1\}^{\lambda} && \textit{public key space} \\ \mathcal{K}_{prk} &:= \{0,1\}^{\lambda} && \textit{private key space} \\ \mathcal{K} &\subset \mathcal{K}_{PUK} \times \mathcal{K}_{prk} && \textit{key space} \end{aligned}$$

Darauf aufbauend umfasst das Schema drei randomisierte, in polynomieller Zeit berechenbare Algorithmen:

- **Schlüsselgenerierung:**

$$(PUK, prk) \leftarrow \text{gen\_key\_pair\_enc()}$$

- **Verschlüsselung:**

$$c := \text{enc\_a}(PUK, p)$$

- **Entschlüsselung:**

$$p := \text{dec\_a}(prk, c)$$

Dabei gelten folgende Definitionsbereiche:

$$PUK \in \mathcal{K}_{PUK}, \quad prk \in \mathcal{K}_{prk}, \quad (PUK, prk) \in \mathcal{K}, \quad p \in \mathcal{P}, \quad c \in \mathcal{C}$$

Für die Korrektheit des Schemas muss gelten:

$$\text{dec\_a}(prk, \text{enc\_a}(PUK, p)) = p$$

für alle Eingaben aus den zuvor angegebenen Definitionsbereichen.

[3] Der *key space* ($\mathcal{K}$) umfasst nur solche Schlüsselpaare (PUK, prk), die gemeinsam durch KeyGen() erzeugt wurden. Beliebige Kombinationen aus $\mathcal{K}_{PUK}$ und $\mathcal{K}_{prk}$ sind in der Regel ungültig.

Digitales Signaturschema

Ein digitales Signaturschema definiert die folgenden *Spaces*[4]:

$$\begin{aligned} \mathcal{K}_{VK} &:= \{0,1\}^\lambda && \textit{verification key space} \\ \mathcal{K}_{sik} &:= \{0,1\}^\lambda && \textit{signing key space} \\ \mathcal{K} &\subset \mathcal{K}_{VK} \times \mathcal{K}_{sik} && \textit{key space} \\ \mathcal{S} &:= \{0,1\}^\lambda && \textit{signature space} \\ \mathcal{D} &:= \Sigma^\star && \textit{data space} \end{aligned}$$

Darauf aufbauend umfasst das Schema drei deterministische, in polynomieller Zeit berechenbare Algorithmen:

- **Schlüsselgenerierung:**

$$(VK, sik) \leftarrow \text{gen\_key\_pair\_sig()}$$

- **Signaturerzeugung:**

$$sig := \text{sign}(sik, d)$$

- **Verifikation:**

$$\text{verify}(VK, d, sig) = \begin{cases} \text{true}, & \text{falls } sig \text{ eine gültige Signatur für } d \text{ ist,} \\ \text{false}, & \text{sonst.} \end{cases}$$

Dabei gelten folgende Definitionsbereiche:

$$VK \in \mathcal{K}_{VK}, \quad sik \in \mathcal{K}_{sik}, \quad (VK, sik) \in \mathcal{K}, \quad d \in \mathcal{D}, \quad sig \in \mathcal{S}$$

Für die Korrektheit des Schemas muss gelten:

$$\text{verify}(VK, d, \text{sign}(sik, d)) = 1$$

für alle Eingaben aus den zuvor angegebenen Definitionsbereichen.

Kryptographische Hashfunktion

Eine kryptographische Hashfunktion definiert die folgenden *Spaces*:

$$\begin{aligned} \mathcal{H}_{in} &:= \Sigma^\star && \textit{hash input space} \\ \mathcal{H}_{out} &:= \{0,1\}^\lambda && \textit{digest space} \end{aligned}$$

[4] Wie bei asymmetrischer Verschlüsselung umfasst der *key space* ($\mathcal{K}$) nur gültige Schlüsselpaare (VK, sik), die durch KeyGen() erzeugt wurden.

Darauf aufbauend definiert das Schema den folgenden Hashberechnungsalgorithmus, der deterministisch und in polynomieller Zeit berechenbar ist:

$$d \leftarrow \text{hash}(x)$$

Dabei gelten folgende Definitionsbereiche:

$$x \in \mathcal{H}_{in}, \quad d \in \mathcal{H}_{out}$$

Key Derivation Function (KDF)

Eine *Key Derivation Function* definiert, die folgenden *Space*s:

$$\begin{aligned} \mathcal{X} &:= \{0,1\}^\star && \textit{KDF input space} \\ \mathcal{R} &:= \{0,1\}^\star && \textit{salt space} \\ \mathcal{I} &:= \Sigma^\star && \textit{context space} \\ \mathcal{K}_{out} &:= \{0,1\}^\lambda && \textit{KDF output space} \end{aligned}$$

Darauf aufbauend definiert das Schema den folgenden Schlüsselableitungsalgorithmus, der deterministisch und in polynomieller Zeit berechenbar ist:

$$k \leftarrow \text{kdf}(x, r, i)$$

Dabei gelten folgende Definitionsbereiche:

$$x \in \mathcal{X}, \quad r \in \mathcal{R}, \quad i \in \mathcal{I}, \quad k \in \mathcal{K}_{out}$$

Hierbei bezeichnet x einen geheimen Eingabewert (z. B. ein Passwort oder einen *Seed*), r einen optionalen *Salt*-Wert zur Erhöhung der Sicherheit gegenüber Wörterbuchangriffe, und i einen optionalen Kontextparameter (auch *label* genannt), der die Ableitung an eine bestimmte Anwendung binden kann. Die gewünschte Ausgabelänge wird durch λ angegeben.

In dieser Arbeit werden hauptsächlich *Password-Based Key Derivation Function*s (*PBKDF*s) verwendet. Diese stellen eine spezialisierte Form von *KDF*s dar, die insbesondere auf Eingaben mit geringer Entropie (z. B. Passwörter) ausgelegt sind. Dabei spielt der *Salt* eine zentrale Rolle. Je nach konkreter Implementierung, wie etwa bei *PBKDF2*, kommen weitere Parameter hinzu, beispielsweise die Anzahl der Iterationen der *KDF*-internen Kryptographische Hashfunktion, um Brute-Force-Angriffe und Wörterbuchangriffe zu erschweren [36, 37].

5.1.8. Definition Datenstrukturen

In diesem Kapitel werden die zentralen Datenstrukturen spezifiziert, welche die Grundlage für die Funktionalität und Sicherheitsarchitektur der *Platform* bilden. Jede Datenstruktur ist als Tupel modelliert, wodurch sich eine klare und formale Beschreibung ergibt. Diese Modellierungsform erlaubt die freie Wahl des Persistenzformats, beispielsweise JSON oder YAML, in der konkreten Implementierung.

Sofern nicht ausdrücklich anders angegeben, werden die beschriebenen Datenstrukturen weder verschlüsselt noch signiert gespeichert. Für Strukturen, die signiert oder verschlüsselt abgelegt werden, ist jeweils spezifiziert, in welcher Form dies erfolgt und unter welchem Dateinamen die Persistenz vorgenommen wird. Der konkrete Ablageort der jeweiligen Dateien innerhalb der Verzeichnisstruktur kann Abbildung 4.7 entnommen werden. Dateien mit verschlüsseltem Inhalt erhalten grundsätzlich die Endung .enc, während Signaturen in separaten Dateien mit der Endung .sig gespeichert werden.

Die Datenstrukturen umfassen sowohl im Klartext gespeicherte als auch verschlüsselte Informationen und decken zentrale Anwendungsfälle wie Registrierung, Authentifizierung, Schlüsselverwaltung, Berechtigungssteuerung und Informationsaustausch ab. Für jede Struktur werden Zweck, Aufbau, enthaltene Werte, deren Bedeutung sowie das jeweilige Datenformat in tabellarischer Form dargestellt.

Der Invitation Letter Caregiver

Beim *Invitation Letter Caregviver* (ILC) handelt es sich um ein Tupel, das alle Informationen enthält, die ein *Caregiver* für die Registrierung auf der *Platform* benötigt. Der ILC wird in Form eines QR-Codes auf dem *Physical / Printed Invitation Letter* (PIL) kodiert. Die inhaltliche Struktur des ILC ist in Tabelle 5.3 dargestellt. Das ILC-Tupel wird lediglich temporär zwischengespeichert und unmittelbar nach der erfolgreichen Kodierung auf den PIL verworfen. Der PIL wird physisch erzeugt und über einen authentischen Kanal an den jeweiligen *Caregiver* übermittelt. Nach dem Druck wird der PIL aus dem System entfernt.

Wert	Typ	Beschreibung
uid_c	string	uid des *Caregiver*
$student$	tupel	Informationen zum zugeordneten *Student*:
uid_s	string	uid des *Student*
$firstName$	string	Vorname des *Student*
$lastName$	string	Nachname des *Student*
otp	string	Einmalig verwendbares Passwort für die Registrierung
uid_a	string	uid des *Admin*
PUK_a	string	PUK des *Admin*
VK_a	string	VK des *Admin*

Tabelle 5.3.: Inhalt und Struktur vom *Invitation Letter Caregviver* (ILC)

Der Invitation Letter Teacher

Beim *Invitation Letter Teacher* (ILT) handelt es sich um ein Tupel, das alle Informationen enthält, die ein *Teacher* für die Registrierung auf der *Platform* benötigt. Der ILT wird in Form eines QR-Codes auf dem PIL kodiert. Die inhaltliche Struktur des ILT ist in Tabelle 5.4 dargestellt. Das ILT-Tupel wird lediglich temporär zwischengespeichert und unmittelbar nach der erfolgreichen Kodierung auf den PIL verworfen. Der PIL wird physisch erzeugt und über einen authentischen Kanal an den jeweiligen *Teacher* übermittelt. Nach dem Druck wird der PIL aus dem System entfernt.

Wert	Typ	Beschreibung
uid_t	string	uid des *Teacher*
otp	string	*One Time Password* für die Registrierung
uid_a	string	uid des *Admin*
PUK_a	string	PUK des *Admin*
VK_a	string	VK des *Admin*

Tabelle 5.4.: Inhalt und Struktur vom *Invitation Letter Teacher* (ILT)

Der Keystore

Der KS stellt ein Tupel dar, das aus der uid sowie einer Liste von *Keystore*-Einträgen besteht. Jeder Keystore-Eintrag ist ein Tupel, das den durch den *Secret Key Identi-*

fier ($skid$) identifizierten sk sowie die verschlüsselten Formen des prk und des sik enthält. Darüber hinaus umfasst er die zur Ableitung und Verschlüsselung erforderlichen Parameter. Die genaue Struktur des KS ist in Tabelle 5.5 dargestellt. Der KS wird signiert, in einem geeigneten Persistenzformat serialisiert und unter dem Namen *keystore* auf der *Platform* beim jeweiligen *User* gespeichert.

Wert	Typ	Beschreibung
uid	string	*User Identifier* (uid), zu welcher der KS gehört
$keystore$	list	Eine Liste von Tupel mit folgendem Inhalt:
$skid$	string	*Secret Key Identifier* ($skid$), identifiziert den verwendeten sk, z. B. usk oder $Passkey\_Device2$
$salt$	string	Salt für die Schlüsselableitung bei passwortbasierten sk (codiert)
$[prk]$	tupel	*AEAD*-Verschlüsselungsergebnis aus der Verschlüsselung des *Private Key* (prk), bestehend aus *Nonce*, *AD*, *Ciphertext*, *Tag*
$nonce_{prk}$	string	*Nonce* für das *AEAD*-Vefahren zur Verschlüsselung des prk (codiert)
ad_{prk}	string	*AD* für das *AEAD*-Vefahren zur Verschlüsselung des prk (Siehe Tabelle 5.22)
ct_{prk}	string	*Ciphertext* aus der *AEAD*-Verschlüsselung des prk (codiert)
tag_{prk}	string	*Tag* aus der *AEAD*-Verschlüsselung des prk (codiert)
$[sik]$	tupel	*AEAD*-Verschlüsselungsergebnis aus der Verschlüsselung des *Signing Key* (sik), bestehend aus *Nonce*, *AD*, *Ciphertext*, *Tag*
$nonce_{sik}$	string	*Nonce* für das *AEAD*-Vefahren zur Verschlüsselung des sik (codiert)
ad_{sik}	string	*AD* für das *AEAD*-Vefahren zur Verschlüsselung des sik (Siehe Tabelle 5.22)
ct_{sik}	string	*Ciphertext* aus der *AEAD*-Verschlüsselung des sik (codiert)
tag_{sik}	string	*Tag* aus der *AEAD*-Verschlüsselung des sik (codiert)

Tabelle 5.5.: Inhalt und Struktur vom *Keystore* (KS)

Das Registrationpackage Caregiver

Das *Registrationpackage Caregiver* (RPC) stellt ein Tupel dar, das die in Tabelle 5.6 aufgeführten Informationen zur Registrierung eines *Caregiver* umfasst. Diese wird für den *Admin* verschlüsselt, jedoch nicht signiert, in einem geeigneten Persistenzformat serialisiert und unter dem Dateinamen *rpc-‹uid›.enc* auf der *Platform* gespeichert. Das RPC dient ausschliesslich dem einmaligen Übertrag der Registrierungsinformationen und wird nach erfolgter Verarbeitung gelöscht. Im Gegensatz dazu enthält das *Userprofile* (PF) dauerhafte Stammdaten eines *Users*.

Wert	**Typ**	**Beschreibung**
uid_c	string	*User Identifier Caregiver* (uid_c)
uid_s	string	*User Identifier Student* (uid_s)
otp	string	*One Time Password* für die Sicherstellung der *Authenticity*
$firstName$	string	Vorname des *Caregiver*
$lastName$	string	Nachname des *Caregiver*
$phone$	string	Telefonnummer des *Caregiver*
$email$	string	E-Mail-Adresse des *Caregiver*
PUK_c	string	*Public Key Caregiver* (PUK_c)
VK_c	string	*Verification Key Caregiver* (VK_c)

Tabelle 5.6.: Inhalt und Struktur eines RPC Eintrags

Das Registrationpackage Teacher

Das *Registrationpackage Teacher* (RPT) stellt ein Tupel dar, das die in Tabelle 5.7 aufgeführten Informationen zur Registrierung eines *Teacher* umfasst. Diese wird für den *Admin* verschlüsselt, jedoch nicht signiert, in einem geeigneten Persistenzformat serialisiert und unter dem Dateinamen *rpt-‹uid›.enc* auf der *Platform* gespeichert. Das RPT dient ausschliesslich dem einmaligen Übertrag der Registrierungsinformationen und wird nach erfolgter Verarbeitung gelöscht. Im Gegensatz dazu enthält das PF dauerhafte Stammdaten eines *Users*.

Wert	Typ	Beschreibung
uid_t	string	*User Identifier Teacher* (uid_t)
otp	string	*One Time Password* für die Sicherstellung der *Authenticity*
$firstName$	string	Vorname des *Caregiver*
$lastName$	string	Nachname des *Caregiver*
$phone$	string	Telefonnummer des *Caregiver*
$email$	string	E-Mail-Adresse des *Caregiver*
PUK_t	string	*Public Key Teacher* (PUK_t)
VK_t	string	*Verification Key Teacher* (VK_t)

Tabelle 5.7.: Inhalt und Struktur eines RPT Eintrags

Das Userprofil

Das PF stellt ein Tupel dar, das die zentralen Stammdaten eines *Users* umfasst. Diese Informationen sind erforderlich, um *Admins*, *Teachers* und *Caregivers* gegenseitigen Zugriff auf ihre Kontaktdaten zu ermöglichen und eine adressierbare Kommunikation zwischen den beteiligten Instanzen sicherzustellen. Das PF wird für jeden *User* verschlüsselt und signiert in einem geeigneten Persistenzformat unter den Dateinamen *profile.enc* und *profile.enc.sig* abgelegt. Die enthaltenen Werte sind in Tabelle 5.8 aufgeführt.

Wert	Typ	Beschreibung
uid	string	*User Identifier* (uid)
$salutation$	string	Anrede
$firstName$	string	Vorname
$lastName$	string	Nachname
$birthday$	string	Geburtsdatum
$street$	string	Strasse und Hausnummer
zip	string	Postleitzahl
$city$	string	Ort
$email$	string	E-Mail-Adresse
$phone$	string	Telefonnummer

Tabelle 5.8.: Inhalt und Struktur eines *Userprofile* (PF) Eintrags

Die Permissions-Datei

Die *Permissions* Datei ($PERF$) ist eine signierte Liste von Tupeln, die Lese- und Schreibrechte für verschlüsselte Dateien auf der *Platform* definiert. Sie wird auf dieser mit dem Dateinamen *permissions* an verschiedenen Orten (siehe 4.7) gespeichert. Für jede Datei wird angegeben, welche *User* (uids) Lese- oder Schreibzugriff besitzen. Im Fall von Leseberechtigungen enthält die $PERF$ zusätzlich für jeden berechtigten *User* den jeweils individuell verschlüsselten symmetrischen Dateischlüssel (fsk). Die zur Entschlüsselung erforderlichen *AEAD*-Metadaten (*Nonce*, *Tag*, *AD*) werden nicht in der $PERF$ gespeichert, sondern sind direkt in der jeweiligen verschlüsselten Datei eingebettet. Der genaue Aufbau eines $PERF$ Eintrags ist in Tabelle 5.9 dargestellt.

Wert	**Typ**	**Beschreibung**
$fileName$	string	Dateiname (z. B. $profile.enc$)
$read$	tupel	Liste von Tupeln bestehend aus uid und verschlüsseltem *File Secret Key* (fsk)
uid	string	*User Identifier* (uid)
$cfsk$	string	Für die uid verschlüsselter fsk (serialisiert)
$write$	list	Liste der uid, die Schreibzugriff besitzen
uid	string	uid

Tabelle 5.9.: Inhalt und Struktur eines *Permissions* Datei ($PERF$) Eintrags

Die Signature-Datei

Zu jeder Datei $file$ existiert eine zugehörige *Signature* Datei ($sigf$) $file.sig$, die eine digitale Signatur enthält (siehe Tabelle 5.10). Signaturen dürfen ausschliesslich durch berechtigte *Users* mit Schreibrechten erstellt werden. Die Berechtigung wird über die $PERF$ anhand des Abgleichs mit der jeweiligen uid überprüft. Wird eine Datei verändert, muss sie erneut signiert werden. Andernfalls gilt die neue Version als ungültig und wird von der *Platform* nicht akzeptiert. Daraus folgt, dass zu jedem Zeitpunkt eine gültige Signatur vorhanden sein muss.

Wert	Typ	Beschreibung
uid	string	uid des *User*, welcher die Signatur erstellt hat
sig	string	Digitale Signatur (serialisiert)
$algorithm$	string	Verwendeter Signaturalgorithmus
$timestamp$	integer	Zeitstempel der Erstellung

Tabelle 5.10.: Inhalt und Struktur eines *Signature* Datei ($sigf$) Eintrags

Die Publickeys-Datei

Die *Publickeys* Datei ($PUKS$) ist eine strukturierte Liste von Tupeln, welche die öffentlichen Schlüsseln sämtlicher *User* enthält. Sie wird ausschliesslich vom *Admin* signiert und unter dem Dateinamen *publickeys* auf der *Platform* abgelegt. Diese Datei ermöglicht es, verschlüsselte Nachrichten zu versenden oder digitale Signaturen zu verifizieren, ohne dass zuvor ein direkter Schlüsselaustausch über einen sicheren Kanal erforderlich ist. Somit benötigt jeder *User* Lesezugriff auf diese Datei. Der genaue Aufbau eines Eintrags in der $PUKS$ ist in Tabelle 5.11 dargestellt.

Wert	Typ	Beschreibung
uid	string	*User Identifier* (uid) des Nutzenden
PUK	string	*Public Key* (PUK) des Nutzenden
VK	string	*Verification Key* (VK) des Nutzenden

Tabelle 5.11.: Inhalt und Struktur eines *Publickeys* Datei ($PUKS$) Eintrags

Die Caregiver OTPS-Datei

Die $COTPS$ ist eine strukturierte Liste, in der für jeden *Student* sämtliche zugehörigen *Caregivers* zusammen mit dem jeweils zugehörigen, vom $client_a$ generierten otp, gespeichert sind. Das jeweilige otp entspricht jenem Wert, der im ILC an die *Caregivers* übermittelt wird und welches im *Registrationpackage Caregiver* (RPC) von den *Caregivers* bei der Registrierung angegeben werden muss, um die eigene *Authenticity* gegenüber der *Platform* zu belegen. Da der $client_a$ sich das otp nicht lokal speichern kann, wird dieses bis zur einmaligen Verwendung dauerhaft in der $COTPS$ abgelegt. Nach erfolgreichem Einsatz wird das betreffende otp aus der $COTPS$ entfernt. Jedes otp wird mit dem PUK_a verschlüsselt gespeichert. Die Datei selbst wird unter dem Dateinamen *OTPS* im Klartext auf der *Platform* abgelegt, da sie keine direkt personenbezogenen Informationen enthält. Sie wird ausschliesslich vom *Admin* signiert und genutzt. Die genaue Struktur eines Eintrags in der $COTPS$ ist in Tabelle 5.12 dargestellt.

Wert	Typ	Beschreibung
uid_s	string	*User Identifier Student* (uid_s) des betreffenden *Student*
$caregivers$	list	Liste von Tupeln, bestehend aus einer uid_c und dem verschlüsselten otp
uid_c	string	*User Identifier Caregiver* (uid_c) der jeweiligen *Caregiver*
$cotp$	string	Für die entsprechende uid_c verschlüsseltes otp

Tabelle 5.12.: Inhalt und Struktur eines *COTPS* Datei ($COTPS$) Eintrags

Die Teacher OTPS-Datei

Die *TOTPS* Datei ($TOTPS$) ist eine Liste, in der für jeden *Teacher*, ein vom $client_a$ generierten otp, gespeichert sind. Der jeweilige otp entspricht jenem Wert, der im ILC an den *Teacher* übermittelt wird und welches im RPT vom *Teacher* bei der Registrierung angegeben werden muss, um die eigene *Authenticity* gegenüber der *Platform* zu belegen. Da der $client_a$ sich den otp nicht lokal vorhalten kann, wird dieser bis zur einmaligen Verwendung in der $TOTPS$ abgelegt. Nach erfolgreichem Einsatz wird der betreffende otp aus der $TOTPS$ entfernt. Jedes otp wird mit dem PUK_a verschlüsselt gespeichert. Die Datei selbst wird unter dem Dateinamen $TOTPS$ im Klartext auf der *Platform* abgelegt, da sie keine direkt personenbezogenen Informationen enthält. Sie wird ausschliesslich vom *Admin* signiert und genutzt. Die genaue Struktur eines Eintrags in der $TOTPS$ ist in Tabelle 5.13 dargestellt.

Wert	Typ	Beschreibung
uid_t	string	uid_t des betreffenden *Teacher*
$cotp$	string	Durch *Public Key Admin* (PUK_a) verschlüsseltes otp

Tabelle 5.13.: Inhalt und Struktur eines *TOTPS* Datei ($TOTPS$) Eintrags

Die Hash OTPS-Datei

Die *HOTPS* Datei ($HOTPS$) ist eine Liste, in der für jeden *Admin* der *Hash* seines otp gespeichert wird, welcher von der *Platform* generiert wurde. Der *Hash* wird aus dem otp gebildet, das dem *Admin* bei der initialen Registrierung (siehe Abs. 5.1.1) übermittelt wird und das der *Admin* bei der Registrierung angeben muss, um seine *Authenticity* gegenüber der *Platform* nachzuweisen. Der *Hash* des otp wird bis zu dessen einmaliger Verwendung in der $HOTPS$ abgelegt. Nach erfolgreichem Einsatz wird der entsprechende Eintrag aus der $HOTPS$ entfernt. Die Datei selbst wird unter dem Dateinamen $HOTPS$ im Klartext auf der *Platform* gespeichert, da sie keine direkt personenbezogenen Informationen enthält. Sie wird ausschliesslich von der

Platform verwendet. Die genaue Struktur eines Eintrags in der $HOTPS$ ist in Tabelle 5.14 dargestellt.

Wert	Typ	Beschreibung
uid_a	string	*User Identifier Admin* (uid_a) des betreffenden *Admin*
$hash_{otp}$	string	Hash eines otp

Tabelle 5.14.: Inhalt und Struktur eines *HOTPS* Datei ($HOTPS$) Eintrags

Aufbau verschlüsselte Datei

Eine verschlüsselte Datei besteht aus einem Tupel, welches alle für die *AEAD*-Entschlüsselung notwendigen Informationen enthält. Diese Informationen sind in ein geeignetes Persistenzformat serialisiert worden. Diese Struktur ermöglicht eine sichere, selbstbeschreibende Speicherung, bei der keine externen Metadaten notwendig sind. Ein typischer Eintrag ist in der Tabelle 5.15 dargestellt.

Wert	Typ	Beschreibung
$nonce$	string	Initialisierungsvektor für die *AEAD*-Verschlüsselung
AD	string	*Associated Data* (*AD*), welche mitverifiziert, aber nicht verschlüsselt werden
$ciphertext$	string	Der verschlüsselte Inhalt der Datei
tag	string	Authentifizierungstag der *AEAD*-Verschlüsselung

Tabelle 5.15.: Struktur einer verschlüsselten Datei

Diese Informationen werden durch den Besitzer der Datei im Zuge der Verschlüsselung mit dem fsk generiert und zusammen mit dem *Ciphertext* in der Datei abgelegt. Nur *User* mit gültigem Zugriff auf den entschlüsselten fsk (gemäss $PERF$, Abs. 5.1.8) können diese Datei erfolgreich entschlüsseln.

Die Students-Datei

Die *Students* Datei (ST) enthält eine Liste aller *Students*, die einem bestimmten *Caregiver* zugeordnet sind. Sie wird im Ordner des jeweiligen *Caregiver* mit dem Namen *students* verschlüsselt abgelegt und dient der Herstellung notwendiger Verknüpfungen innerhalb des Systems.

Der *Caregiver* erhält ausschliesslich Lesezugriff auf diese Datei, um die ihm zugewiesenen *Students* einsehen zu können. Für jeden *Student* wird lediglich die zugehörige *User Identifier Student* (uid_s) gespeichert. Dadurch bleibt die Datei bewusst minimal

gehalten und wird aus datenschutzrechtlicher Sicht als unkritisch eingestuft. Die Datei wird ausschliesslich durch den *Admin* signiert.

Die Struktur eines Eintrags der ST ist in Tabelle 5.16 dargestellt.

Wert	Typ	Beschreibung
uid_s	string	*User Identifier* (uid) von *Student*, die dem *Caregiver* zugewiesenen wurde

Tabelle 5.16.: Entspricht einem Eintrag in der *Students* Datei (ST)

Die Caregivers-Datei

Die *Caregivers* Datei (CG) enthält eine Liste aller *Caregivers*, die einem bestimmten *Student* zugeordnet sind. Sie wird im Ordner des jeweiligen *Student* mit dem Namen *caregivers* verschlüsselt abgelegt und dient der Herstellung notwendiger Verknüpfungen innerhalb des Systems.

Der *Caregiver* besitzt ausschliesslich Lesezugriff auf diese Datei. Die Pflege und Verwaltung obliegt ausschliesslich dem *Admin*. Die Datei wird somit ausschliesslich durch den *Admin* signiert.

Für jeden *Caregiver* wird lediglich die zugehörige *User Identifier Caregiver* (uid_c) gespeichert. Dadurch bleibt die Datei bewusst minimal gehalten und wird aus datenschutzrechtlicher Sicht als unkritisch eingestuft.

Die Struktur eines Eintrags der CG ist in Tabelle 5.17 dargestellt.

Wert	Typ	Beschreibung
uid_c	string	*User Identifier* (uid) des jeweiligen *Caregiver*, die dem *Student* zugewiesenen wurde

Tabelle 5.17.: Entspricht einem Eintrag in der *Caregivers* Datei (CG)

Die Teachers-Datei

Die *Teachers* Datei (TA) enthält eine Liste aller *Teachers*, die einem bestimmten *Student* zugeordnet sind. Sie wird im Ordner des jeweiligen *Student* mit dem Namen *teachers* verschlüsselt abgelegt und dient der Herstellung notwendiger Verknüpfungen innerhalb des Systems.

Die zugehörigen *Caregivers* des *Student* haben lediglich Lesezugriff, um die dem *Student* zugeordneten *Teachers* einsehen zu können. Die *Teachers* besitzen keinen Zugriff auf diese Datei. Die Erstellung, Pflege und das Signieren der Datei obliegen ausschliesslich dem *Admin*.

Für jeden *Teacher* wird lediglich die zugehörige uid_t gespeichert. Dadurch bleibt die Datei bewusst minimal gehalten und wird aus datenschutzrechtlicher Sicht als unkritisch eingestuft.

Die Struktur eines Eintrags der TA ist in Tabelle 5.18 dargestellt.

Wert	**Typ**	**Beschreibung**
uid_t	string	*User Identifier* (uid) des jeweiligen *Teacher*, die dem *Student* zugewiesenen wurde

Tabelle 5.18.: Entspricht einem Eintrag in der *Teachers* Datei (TA)

Die Absences-Datei

Die *Absences* Datei (AB) enthält eine Liste von Absenzeinträgen eines bestimmten *Student*. Jeder Eintrag dokumentiert eine Abwesenheitsperiode und deren Begründung. Die Datei wird im Ordner des jeweiligen *Student* verschlüsselt unter dem Namen *absences* abgelegt.

Nur berechtigte Parteien können die Datei entschlüsseln und einsehen. Dazu gehören der *Admin*, die für den *Student* zuständigen *Teachers* sowie die verantwortlichen *Caregivers*.

Die Leserechte sind auf die genannten Rollen beschränkt. Schreibrechte besitzen ausschliesslich die *Teachers*, die den *Student* unterrichten, sowie der *Admin*. Dadurch wird sichergestellt, dass nur autorisierte Personen Absenzeinträge erfassen oder bearbeiten können.

Die Struktur eines Eintrags in der AB ist in Tabelle 5.19 dargestellt.

Wert	**Typ**	**Beschreibung**
$startDate$	integer	Datum, an dem die Absenz beginnt, als Unix-Zeitstempel (Sekunden seit dem 1. Januar 1970)
$endDate$	integer	Datum, an dem die Absenz endet, als Unix-Zeitstempel (Sekunden seit dem 1. Januar 1970)
$reason$	string	Freitextfeld zur Begründung der Absenz, z. B. «Krankheit» oder «Jokertag»

Tabelle 5.19.: Entspricht einem Eintrag in der *Absences* Datei (AB)

Die Nachrichtendatei (MF)

Eine Nachrichtendatei (*Message* Datei (MF)) repräsentiert eine einzelne, ausgetauschte Nachricht auf der *Platform*. Die MF ist ein Tupel bestehend aus den vier Hauptkomponenten *header*, *sigInfo*, *rcpInfo* und *[msg]*. Sie enthält alle notwendigen Informationen, um die Nachricht zu verarbeiten, ihre *Authenticity* und *Integrity* zu überprüfen und den Inhalt für berechtigte Parteien zugänglich zu machen. Die Datei selbst wird, wie in Abschnitt 4.3 beschrieben, unter einem dynamisch generierten Dateinamen (z.B. $<timeStamp>-<mid>$) im jeweiligen Chat-Ordner des *Student* oder der Klasse auf der *Platform* gespeichert. Der eigentliche Nachrichteninhalt ist dabei *AEAD*-verschlüsselt, während die anderen Komponenten der MF im Klartext vorliegen, um die notwendige Verarbeitung und Weiterleitung zu ermöglichen.

Wert	Typ	Beschreibung
$header$	tupel	Metadaten der Nachricht:
uid_{sender}	string	uid des Senders.
$recipients$	list	Liste der uids der Empfänger im Chat-Kontext.
mid	string	Eindeutige Nachrichten-ID im Chat-Kontext.
$timestamp$	integer	Zeitstempel der Nachrichtenerstellung.
$sigInfo$	tupel	Signaturinformationen:
sig	string	Digitale Signatur über den *Hash* von $(header, msg)$).
$hash\_algorithm$	string	Verwendeter Hash-Algorithmus (z.B. SSHA256").
$rcpInfo$	list	Liste mit Einträgen für jede Partei zum Entschlüsseln des *Message Secret Key* (msk):
uid_p	string	uid der Partei (Sender/Empfänger).
$cmsk$	string	Asymmetrisch mit dem PUK von uid_p verschlüsselter Session Key (msk).
$[msg]$	tupel	*AEAD*-verschlüsselter Nachrichteninhalt und Metadaten:
$nonce_{msg}$	string	*Nonce* für die *AEAD*-Verschlüsselung.
ad_{msg}	string	*AD* für die *AEAD*-Verschlüsselung (Siehe Tabelle 5.22).
ct_{msg}	string	Verschlüsselter Nachrichteninhalt (*Ciphertext*).
tag_{msg}	string	Authentifizierungs-*Tag* der *AEAD*-Verschlüsselung.

Tabelle 5.20.: Inhalt und Struktur einer Nachrichtendatei (*Message* Datei (MF))

Die Parties-Datei

Die *parties*-Datei ist ein Tupel, das die Lese- und Schreibrechte innerhalb eines bestimmten Chat-Kontexts auf der *Platform* definiert. Sie enthält zwei explizite Listen von uids, die jeweils den Zugriff auf Nachrichten in diesem Chat steuern. Die Liste $read$ spezifiziert alle *Users*, welche an diesem Chat beteiligt sind, während $write$ jene *Users* aufführt, die Nachrichten in diesem Chat verfassen dürfen. Die *parties*-Datei wird im Klartext und entweder durch den Admin oder durch den *User* signiert, welcher den Chat erstellt hat.

Die Datei wird typischerweise unter dem Namen *parties* im jeweiligen Chat-Ordner einer Klasse oder oder eines *Students* gespeichert.

Wert	**Typ**	**Beschreibung**
$read$	list	Liste von *User Identifier*s (uids), welche berechtigt sind, Nachrichten dieses Chats zu lesen
$uid_{recipient}$	string	uid Leser
$write$	list	Liste von uids, welche berechtigt sind, Nachrichten in diesem Chat zu verfassen
uid_{sender}	string	uid Sender

Tabelle 5.21.: Inhalt und Struktur der *parties*-Datei

5.2. Protokoll Beschreibung

Dieses Kapitel beschreibt das kryptographische Protokoll im Detail. Im Zentrum stehen zwei Ziele. Erstens soll die Darstellung klar und konsistent erfolgen und damit eine fundierte Grundlage für künftige Implementierungen schaffen. Zweitens wird die Spezifikation bewusst abstrakt gehalten, sodass die zugrunde liegenden kryptographischen Primitive austauschbar bleiben, solange die wesentlichen Sicherheitsanforderungen wie *Confidentiality*, *Integrity* und *Authenticity* erfüllt sind.

Zu Beginn jedes Protokollschritts wird angegeben, welche Variablen wie Schlüssel, Identifikatoren oder Klartextdaten der jeweiligen Partei bereits bekannt sein müssen, damit sie die beschriebenen Operationen korrekt ausführen kann. Diese Angaben erfolgen in folgender Notation:

$$\text{Knows } var_1, var_2, ...$$

Die Herkunft dieser Werte wird im jeweiligen Schritt nicht nochmals erläutert. Sie ergibt sich entweder aus vorhergehenden Protokollschritten oder wird aus dem Kontext wie etwa der Initialisierungsphase vorausgesetzt.

Aus Gründen der Lesbarkeit wird in den folgenden Protokollbeschreibungen auf die explizite Konvertierung von Strings in Bytefolgen und umgekehrt verzichtet, beispielsweise vor der Verschlüsselung mittels enc_c oder nach der Entschlüsselung durch dec_s.

5.2.1. Variablen

Zur besseren Lesbarkeit der Protokollschritte und des Pseudocodes werden bestimmte Variablen und Platzhalter verwendet. Dazu gehören insbesondere vordefinierte *AD*-Objekte, welche im Folgenden spezifiziert werden.

Darüber hinaus wird im Rahmen verschiedener Protokollschritte, insbesondere bei der Registrierung von *Users* eine generische Variable $usrdata$ verwendet. Diese repräsentiert eine Sammlung von personenbezogenen Profildaten, die für die jeweilige Person erfasst werden. Typische Bestandteile von $usrdata$ können Anrede, Vorname, Nachname, Geburtsdatum, Adresse, E-Mail-Adresse und Telefonnummer sein. Die genaue Definition, welche dieser Profildaten für den Schulalltag und die spezifische Rolle des *User* zwingend erforderlich sind, ist im jeweiligen Anwendungskontext festzulegen, in der Regel in Abstimmung mit dem Schulpersonal, und liegt ausserhalb des detaillierten Umfangs der kryptographischen Protokollspezifikation dieser Arbeit. Die Variable $usrdata$ wird beispielsweise im Protokoll zur Administratorregistrierung (siehe Abschnitt 1) sowie in den Registrierungsprozessen für andere *User* verwendet, um die notwendigen Stammdaten zu übermitteln.

Associated Data (AD) Variablen

Tabelle 5.22 zeigt die im Protokoll und Pseudocode verwendeten *AD*-Objekte. Diese definieren den unveränderlichen Kontext symmetrischer Verschlüsselungsoperationen im Sinne eines *AEAD*-Schemas.

Alle *AD*-Variablen folgen demselben strukturellen Schema:

$$creator : uid_i \mid target : (uid_i)^n \ \vee \ "public" \mid label : \ "prk" \mid algorithm : \ "x" \mid version : \ "v1"$$

Dabei bezeichnet $creator$ die uid des *Users*, der den Inhalt verschlüsselt hat. $target$ steht für die Zielgruppe der Entschlüsselung. Dies kann entweder eine Liste von $n > 0$ uids oder das Schlüsselwort $public$ sein, falls die Zielgruppe nicht weiter eingeschränkt ist.

Das Feld $label$ beschreibt, welcher Inhalt verschlüsselt wurde, etwa ein Dateiname oder ein anderer semantischer Bezeichner (z. B. $student\_key$ oder $profile$), $algorithm$ gibt den verwendeten *AEAD*-Algorithmus an (z. B. *AES-GCM*) und dient dazu, die korrekte Verarbeitung bei zukünftigen Änderungen am Verschlüsselungsverfahren sicherzustellen.

Die Angabe $version$ beschreibt die Version der zugrunde liegenden Klartextstruktur. Sie wird relevant, wenn sich das Datenformat ändert, beispielsweise durch einen Wechsel des asymmetrischen Verschlüsselungsschemas, der Auswirkungen auf die Struktur gespeicherter Schlüssel oder anderer Objekte haben kann.

Dasselbe Schema wird auch beim Versenden von Nachrichten verwendet. In diesem Fall sind die Felder wie folgt belegt:

- $creator$: Die uid der Person, die die Nachricht gesendet hat.
- $target$: Eine Liste von uids der Empfängerinnen und Empfänger.
- $label$: Die mid der Nachricht.
- $algorithm$: Verwendeter AEAD-Algorithmus (z. B. *AES-GCM*)
- $version$: Die Version des Nachrichtenformats, standardmässig $''v1''$.

Variable	**Wert**	**Beschreibung**	**Prot./Alg.**
ad_{prk}	$creator : uid_i \mid target : uid_i \mid label : ''prk'' \mid algorithm : ''AES-GCM'' \mid version : ''v1''$	*AD* für einen prk einer uid_i, $i \in \{a,t,c,s\}$, der durch einen sk (identifiziert durch $skid$) verschlüsselt wurde.	Alg. 48
ad_{sik}	$creator : uid_i \mid target : uid_i \mid label : ''sik'' \mid algorithm : ''AES-GCM'' \mid version : ''v1''$	*AD* für einen sik einer uid_i, $i \in \{a,t,c,s\}$, der durch einen sk (identifiziert durch $skid$) verschlüsselt wurde.	Alg. 48
ad_{pf}	$creator : uid_i \mid target : ''public'' \mid label : ''profile'' \mid algorithm : ''AES-GCM'' \mid version : ''v1''$	*AD* für das PF.	Prot. 2, 9, 1, 4, 12
ad_{st}	$creator : uid_i \mid target : ''public'' \mid label : ''students'' \mid algorithm : ''AES-GCM'' \mid version : ''v1''$	*AD* für das ST.	Prot. 2, 12, Alg. 42, 45

Variable	Wert	Beschreibung	Prot./Alg.
ad_{ab}	$creator : uid_i \mid target : "public" \mid label : "absences" \mid algorithm : "AES-GCM" \mid version : "v1"$	*AD* für das AB.	Prot. 9
ad_{ta}	$creator : uid_i \mid target : "public" \mid label : "teachers" \mid algorithm : "AES-GCM" \mid version : "v1"$	*AD* für das TA.	Prot. 9, 2, Alg. 43, 44
ad_{cg}	$creator : uid_i \mid target : "public" \mid label : "caregivers" \mid algorithm : "AES-GCM" \mid version : "v1"$	*AD* für das CG.	Prot. 2, 12
ad_{rp}	$creator : uid_i \mid target : "public" \mid label : "registrationpackage" \mid algorithm : "AES-GCM" \mid version : "v1"$	*AD* für das RPC und RPT.	Prot. 4, 12, Alg. 7
ad_{msg}	$creator : uid_{sender} \mid target : \{uid_{recipient}\}^n \mid label : MID \mid algorithm : "AES-GCM\|SHA-256\|RSA-PSS" \mid version : "v1"$	*AD* für das MF.	Alg. 59, 60

Tabelle 5.22.: Definition von *Associated Data*-Variablen, welche sowohl in den Protokollen als auch in den Algorithmen verwendet werden

Notation von Signaturen

In den nachfolgenden Protokollschritten und Pseudocode-Algorithmen wird zur Vereinfachung die Variable *sig* verwendet. Es ist wichtig zu verstehen, dass diese Variable je nach Kontext zwei unterschiedliche, aber eng miteinander verbundene Konzepte repräsentieren kann:

- **Die digitale Signatur selbst:** In vielen kryptographischen Operationen, insbesondere bei der Erzeugung oder Verifizierung, steht *sig* für den eigentlichen kryptographischen Wert der digitalen Signatur, also das Ergebnis des Signatur-Algorithmus.
- **Die Signaturdatei:** Wenn es um die Persistierung oder den Austausch von Signaturen im Dateisystem der *Platform* geht, kann *sig* auch die Datei repräsentieren, welche die digitale Signatur enthält (typischerweise eine Datei mit der Endung *.sig*, wie in Abschnitt 5.1.8 beschrieben). In den Protokollschritten wird dies oft impliziert, wenn *sig* als Ergebnis einer *load_file*-Operation oder

als Eingabe für eine *store_file*-Operation auftaucht, die eine Signatur betrifft.

Diese kontextabhängige Interpretation der Variable *sig* dient der Kompaktheit der Darstellung in den Protokollschritten und Pseudocodes. Diese Vorgehensweise wurde gewählt, um die Lesbarkeit der komplexen Protokollschritten zu erhöhen, ohne die Eindeutigkeit im jeweiligen Anwendungskontext zu beeinträchtigen.

5.2.2. Registrierung des Admins

Gemäss den in Abschnitt 5.1.1 dargelegten initialen Annahmen existiert die grundlegende Ablagestruktur der Schule auf der *Platform* bereits und der designierte *Admin* verfügt über alle erforderlichen Informationen für seine erstmalige Registrierung, einschliesslich eines einmaligen Passworts (otp). Die Registrierung des *Admin* ist ein kritischer und fundamentaler Protokollschritt, da hierdurch der primäre Vertrauensanker für alle nachfolgenden Sicherheitsoperationen und administrativen Prozesse innerhalb der Schulkommunikationslösung etabliert wird. Das Hauptaugenmerk dieses Protokolls, dargestellt in Protokoll 1, liegt darauf, die einmalige und authentisierte Registrierung des vorgesehenen *Admin* sicherzustellen. Erst nach erfolgreicher Verifikation des otp durch die *Platform* werden dessen *Publickeys* Datei sowie die zugehörigen Profildaten, bestehend aus dem verschlüsselten Keystore (KS), dem Benutzerprofil (PF) und der initialen Berechtigungsdatei ($PERF$), auf der *Platform* persistiert.

Das Ziel dieses Protokolls ist somit die sichere Initialisierung der $PUKS$-Datei auf der *Platform* durch den *Admin*, deren Signierung, sowie die Hinterlegung seines verschlüsselten Profils und Keystores.

In diesem Protokollschritt werden die Algorithmen DeriveUserSecretKey() (46), GenKeyPairs() (47), EncPrivateKeys() (48), EncSign() (8), VerifyAdmOTP() (52), StoreInitPUKs() (14), StoreInitEncFile() (15) und StoreInitSignFile() (16) verwendet. Diese sind in Abschnitt 5.3 zu finden.

Initialisierung von Klassen durch den Admin

Nach der erfolgreichen Registrierung des *Admin* und bevor spezifische *Users* wie *Teachers* oder *Caregivers* dem System hinzugefügt werden, ist ein wesentlicher vorbereitender Schritt die Initialisierung der Klassenstrukturen auf der *Platform*. Dieser Prozess wird exklusiv vom *Admin* durchgeführt und legt die Grundlage für die spätere Zuweisung von *Students* und *Teachers* sowie für die klassenspezifische Kommunikation. Für jede zu erstellende Klasse führt der $client_a$ die im Prot. 2 dargestellten Aktionen aus.

Ziel dieses Protokollschritts ist es, für jede Klasse eine sichere und klar definierte Grundstruktur auf der *Platform* zu etablieren. Dies umfasst leere, aber korrekt

verschlüsselte und signierte *Students* Datei und *Teachers* Datei, eine initiale *Permissions* Datei, die dem *Admin* die Verwaltung dieser Dateien ermöglicht, sowie eine initiale Konfiguration für den Klassenchat. Alle Operationen werden lokal auf dem $client_a$ durchgeführt, und nur die verschlüsselten bzw. signierten Ergebnisse werden an die *Platform* übertragen.

Die in diesem Protokollschritt verwendeten Algorithmen umfassen EncSign() (8), StoreInitEncFile() (15) und StoreInitSignFile() (16) verwendet. Diese sind in Abschnitt 5.3 zu finden.

5.2.3. Bootstrapping

Die *Bootstrapping* Phase bereitet das System vor. Sie wird typischerweise zu Beginn eines jeden Schuljahres durchgefürt. Der *Admin* hat sich wie in Abschnitt 5.2.2 beschrieben registriert. Ausserdem hat er die Liste aller *Students* bereits erhalten. Die *Bootstrapping* Phase hat die folgenden Ziele:

1. **Etablierung aller Benutzerkonten:** Alle notwendigen Benutzerrollen (*Admin*, *Teacher*, *Student*, *Caregiver*) sind auf der *Platform* registriert und verfügen über eine eindeutige Identifikation (uid) sowie ein initiales, gesichertes Profil.
2. **Initialisierung der kryptographischen Basis:** Für jeden registrierten Benutzer sind die erforderlichen kryptographischen Schlüsselpaare generiert, die öffentlichen Schlüssel sicher in der globalen $PUKS$ hinterlegt und geheime Schlüsselkomponenten geschützt abgelegt.
3. **Sicherstellung der grundlegenden Datenintegrität und Zugriffskontrolle:** Alle initialen Datenstrukturen und Benutzerdaten sind gemäss den definierten Sicherheitsrichtlinien (Verschlüsselung, Signatur) geschützt und die initialen Zugriffsberechtigungen (z.B. Klassenzuordnungen, Chat-Teilnahmen) sind konfiguriert.

Registrierung und Verifizierung eines Teacher

Für die Registrierung eines *Teacher* ist es essenziell, dass der PIL über einen authentischen und vertrauenswürdigen Kanal vom *Admin* an den *Teacher* übermittelt wird. Nach Erhalt des PIL scannt der *Caregiver* den enthaltenen QR-Code und wird zur Eingabe persönlicher Daten aufgefordert.

Analog zur Registrierung des *Admin* (vgl. Abschnitt 5.2.2) erhält der *Teacher* ein im Prot. 3 generiertes otp. Dieses dient im Anschluss dazu, die *Integrity* und *Authenticity* des vom *Teacher* erzeugten RPT zu gewährleisten. Der *Teacher* erstellt das RPT im Rahmen von Prot. 4. Dabei werden unter anderem kryptographische Schlüssel generiert und Metadaten verschlüsselt im RPT abgelegt, bevor dieses auf die *Platform*

hochgeladen wird. Die Verifikation dieses RPT durch den *Admin* erfolgt in Prot. 5 mithilfe des otp.

Nach erfolgreicher Verifikation gemäss Prot. 6 werden die öffentlichen Schlüssel des *Teacher* auf der *Platform* hinterlegt. Im Anschluss daran erfolgt gemäss Prot. 7 die Speicherung der Profildaten sowie des zugehörigen KS im persönlichen Ordner des *Teacher*. Schliesslich wird dem *Teacher* in Prot. 8 die Berechtigung für eine spezifische Klasse und den zugehörigen Klassenchat erteilt. Dies umfasst die Fähigkeit, Nachrichten zu senden und die uids der *Students* dieser Klasse einzusehen. Nach erfolgreicher Verifikation gemäss Prot. 6 werden die öffentlichen Schlüssel des *Teacher* auf der *Platform* hinterlegt. Im Anschluss daran erfolgt gemäss Prot. 7 die Speicherung der Profildaten sowie des zugehörigen KS im persönlichen Ordner des *Teacher*. Schliesslich wird dem *Teacher* in Prot. 8 die Berechtigung für eine spezifische Klasse und den zugehörigen Klassenchat erteilt. Dies umfasst die Fähigkeit, Nachrichten zu senden und die uids der *Students* dieser Klasse einzusehen.

Ziel der Protokollschritte 3 bis 7 ist es, den *Teacher* vollständig für die Nutzung der *Platform* zu initialisieren und ihm den sicheren Zugriff auf plattformspezifische Funktionen zu ermöglichen.

In diesen Protokollschritten werden die Algorithmen GenUID() (2), GenOTP() (3), StoreTeaOTPS() (17), StoreRootPERF() (21), DeriveUserSecretKey() (46), GenKeyPairs() (47), EncPrivateKeys() (48), EncRP() (7), StoreRPTea() (22), VerifyTeaOTP() (54), StorePUKS() (28), EncSign() (8), CreateUserDir() (37), StoreInitSignFile() (16), StoreInitEncFile() (15), DeleteRPTea() (24), GiveReadAccessToFile() (41), AddTeaToClass() (43), AddUIDToChat() (40) verwendet. Diese sind in Abschnitt 5.3 zu finden.

Initialisierung des Student-Ordners

Im Prot. 9 wird der *Student* inklusive seines Ordners und der darin abgelegten Dateien durch den *Admin* erstellt. Der *Admin* besitzt zu diesem Zeitpunkt alle nötigen Informationen über den *Student* und weiss, welche *Teachers* ihn unterrichten werden. Er importiert diese Informationen in seinen $client_a$. Anschliessend erstellt dieser alle benötigten Ordner und Dateien auf der *Platform*. Zudem werden bereits die zuständigen *Teacher* auf die Dateien des *Student* berechtigt.

Ziel dieses Protokollschritts ist es, den *Student* auf der *Platform* zu erfassen, seine Informationen sicher abzulegen und den bereits bestehenden Parteien gezielte Zugriffsrechte auf diese Informationen zu geben.

In diesem Protokollschritt werden die Algorithmen GenUID() (2), GenReadList() (58), EncSign() (8), AddStuToClass() (42), CreateUserDir() (37), StoreInitSignFile() (16) und StoreInitEncFile() (15) verwendet. Diese sind in Abschnitt 5.3 beschrieben.

Registrierung und Verifizierung eines Caregivers

Für die Registrierung eines *Caregiver* ist es essenziell, dass der PIL über einen authentischen und vertrauenswürdigen Kanal vom *Admin* an den *Caregiver* übermittelt wird. Nach Erhalt des PIL scannt der *Caregiver* den enthaltenen QR-Code und wird zur Eingabe persönlicher Daten aufgefordert, nachdem bestätigt wurde, dass der Name vom eigenen Kind angezeigt wurde.

Analog zur Registrierung des *Teacher* (vgl. Abschnitt 5.2.3) erhält der *Caregiver* ein im Prot. 10 generiertes otp. Dieses dient im Anschluss dazu, die *Integrity* und *Authenticity* des vom *Caregiver* erzeugten RPC zu gewährleisten. Der *Caregiver* erstellt das RPC im Rahmen von Prot. 11. Dabei werden unter anderem kryptographische Schlüssel generiert und Metadaten verschlüsselt im RPC abgelegt, bevor dieses auf die *Platform* hochgeladen wird. Die Verifikation dieses RPC durch den *Admin* erfolgt in Prot. 12 mithilfe des auf der *Platform* liegenden otp.

Nach erfolgreicher Verifikation gemäss Prot. 13 werden die öffentlichen Schlüssel des *Caregiver* auf der *Platform* hinterlegt. Anschliessend erfolgt in Prot. 14 die Speicherung der Profildaten und des zugehörigen KS. Neben dem *Caregiver* selbst erhält auch der *Admin* Zugriffsberechtigungen auf diese Dateien. Obwohl hier nicht explizit aufgeführt, erhalten weitere Parteien (wie z.B. der *Teacher*) basierend auf demselben Berechtigungsprinzip ebenfalls ausgewählte Zugriffsrechte auf das Profil des *Caregiver*. Dies wird jedoch zugunsten der Übersichtlichkeit im Protokoll nicht weiter detailliert. Nach erfolgreicher Verifikation gemäss Prot. 13 werden die öffentlichen Schlüssel des *Caregiver* auf der *Platform* hinterlegt. Anschliessend erfolgt in Prot. 14 die Speicherung der Profildaten und des zugehörigen KS. Neben dem *Caregiver* selbst erhält auch der *Admin* Zugriffsberechtigungen auf diese Dateien. Obwohl hier nicht explizit aufgeführt, erhalten weitere Parteien (wie z.B. der *Teacher*) basierend auf demselben Berechtigungsprinzip ebenfalls ausgewählte Zugriffsrechte auf das Profil des *Caregiver*. Dies wird jedoch zugunsten der Übersichtlichkeit im Protokoll nicht weiter detailliert.

Zusätzlich werden daraufhin im Prot. 15 dem *Caregiver* im Ordner des *Student* die notwendigen Zugriffsberechtigungen erteilt und die Verknüpfung zwischen *Student* und *Caregiver* erstellt. Weiter erhält der *Caregiver* im Prot. 16 Leseberechtigungen auf einen Klassenchat seines *Student*, um die Nachrichten verfolgen zu können. Ziel der Protokollschritte 10 bis 15 ist es, den *Caregiver* vollständig für die Nutzung der *Platform* zu initialisieren und ihm den sicheren Zugriff auf plattformspezifische Funktionen zu ermöglichen. Diese Initialisierung gewährleistet die reibungslose Durchführung aller zukünftigen Verschlüsselungs-, Entschlüsselungs-, Signatur- und Verifizierungsprozesse vom *Caregiver*.

In diesen Protokollschritten werden die Algorithmen GenUID() (2), GenCOTPs() (55), (2), GetVerifiedPermissions() (32), StoreCarOTPS() (19), StorePermissions() (31), GetEncFileKey() (13), GetEncFileKey() DecVerify() (10), DeriveUserSecretKey() (46), GenKeyPairs() (47), EncPrivateKeys() (48), EncRP() (7), StoreRPC() (25), VerifyCarOTP() (57),

StorePUKS() (28), EncSign() (8), CreateUserDir() (37), StoreInitSignFile() (16), StoreInitEncFile() (15), DeleteRPCar() (27) und AddUIDToChat() (40) verwendet. Diese sind in Abschnitt 5.3 zu finden.

5.2.4. Protokolle für den sicheren Nachrichtenaustausch

Der Austausch von Direkt- und Gruppennachrichten bildet eine Kernfunktionalität der Schulkommunikationslösung und erfordert robuste Sicherheitsmechanismen, um die *Confidentiality*, *Integrity* und *Authenticity* der sensiblen schulischen Kommunikation zu gewährleisten. Dieser Abschnitt beschreibt die Protokolle, die diesen sicheren Austausch ermöglichen und baut auf den in Kapitel 4 konzeptionell dargestellten Mechanismen auf. Die detaillierten Abläufe werden in den Prot. 17 und Prot. 18 visualisiert.

Die hier spezifizierten Protokolle zielen darauf ab, eine *E2EE* für alle Nachrichten zu realisieren, bei der die *Platform* selbst zu keinem Zeitpunkt Zugriff auf die Klartextinhalte hat. Dies wird durch eine Kombination aus symmetrischer und asymmetrischer Kryptographie erreicht, die im Folgenden näher erläutert wird.

In diesen Protokollschritten kommen die Algorithmen SendMessage() (59), ReceiveMessage() (60) sowie GetStudentsTeachers() (44) zur Anwendung. Sie umfassen alle wesentlichen Operationen, die für das Senden und Empfangen von Nachrichten erforderlich sind.

Sicherheitsüberlegungen für den Nachrichtenversand

Das Versenden einer Nachricht durch einen *User* an einen oder mehrere Empfänger initiiert einen mehrstufigen kryptographischen Prozess, der sicherstellt, dass die Nachricht auf ihrem Weg und bei ihrer Speicherung auf der *Platform* geschützt ist. Der Prozess stützt sich massgeblich auf den Algorithmus SendMessage() (59).

Ein zentrales Designelement für die Schlüsselerzeugung und -verteilung ist die Verwendung eines symmetrischen Nachrichtenschlüssels (msk) für jede einzelne Nachricht. Dieser Ansatz bietet mehrere Vorteile:

- **Nachrichten-spezifische Sicherheit:** Eine Kompromittierung eines einzelnen msk betrifft nur die zugehörige Nachricht und nicht die gesamte Kommunikationshistorie.
- **Effizienz für Gruppenchats:** Der Nachrichteninhalt muss nur einmal mit dem msk symmetrisch verschlüsselt werden. Anschliessend wird lediglich der msk für jeden berechtigten Empfänger (einschliesslich des Senders selbst, um gesendete Nachrichten lesen zu können) individuell mit dessen öffentlichem Schlüssel (PUK) asymmetrisch verschlüsselt. Diese verschlüsselten msks ($cmsk$) werden Teil der Nachrichtendatei (MF).

Die für die Verschlüsselung der msks benötigten PUKs der Empfänger werden aus der globalen, vom *Admin* signierten $PUKS$-Datei bezogen.

Zur Gewährleistung von *Integrity* und *Authenticity* wird in diesem Protokoll der *Sign-and-Encrypt* Ansatz verfolgt. Hierbei wird eine digitale Signatur über einen kryptographischen *Hash* der Nachricht, bestehend aus dem Header (mit Metadaten wie Sender, Empfängerliste, mid, Zeitstempel) und dem Klartext Inhalt, gebildet. Durch die Einbeziehung des eindeutigen Zeitstempels und der mid in die Hash-Berechnung wird zudem sichergestellt, dass selbst für semantisch identische Nachrichten, die zu unterschiedlichen Zeitpunkten gesendet werden, jeweils ein einzigartiger Hash und somit eine einzigartige Signatur entsteht. Diese Signatur wird mit dem privaten Signierschlüssel (sik) des Senders erstellt und wird Teil der *sigInfo*-Komponente der MF. Der eigentliche Nachrichteninhalt wird *AEAD*-verschlüsselt.

Die bewusste Entscheidung für *Sign-and-Encrypt*, spezifisch mit einer Signatur über den *Hash* des Klartextes und Headers, bietet entscheidende Vorteile:

- **Direkte Authenticity des semantischen Inhalts:** Die Signatur ist untrennbar mit dem ursprünglichen, unveränderten Klartext und dessen Kontext (Header) verbunden. Dies ermöglicht es dem Empfänger, nach der Entschlüsselung unmittelbar die *Integrity* und *Authenticity* der ursprünglichen, unverschlüsselten Daten und Metadaten zu verifizieren. Die Signatur bestätigt somit, was der Sender tatsächlich intendiert hat zu kommunizieren.
- **Kein Informationsleck über Klartext durch die Signatur:** Da die Signatur über den *Hash* und nicht über den Klartext direkt erstellt wird, offenbart die Signatur an sich keine direkten Informationen über den Nachrichteninhalt. Der *Hash* ist das Ergebnis einer Einwegfunktion und lässt keine Rückschlüsse auf die ursprünglichen Daten zu. Folglich erfordert die Signatur selbst keine separate Verschlüsselung, um Informationslecks über den Klartext zu verhindern. Ihre Sicherheit in dieser Hinsicht wird durch das vorherige *Hashing* gewährleistet. Dies ist ein wichtiger Aspekt, um die *Confidentiality* auch gegenüber Parteien zu wahren, die die Signatur, aber nicht den verschlüsselten Inhalt einsehen können.
- **Robustheit gegenüber Ciphertext-Manipulationen:** Die Signatur über den Hash des ursprünglichen Klartextes und Headers, bevor dieser verschlüsselt wird, bietet eine robuste Verteidigung gegen bestimmte Angriffe auf den *Ciphertext*. Selbst wenn der Schutz der nachfolgenden *AEAD*-Verschlüsselung kompromittiert würde und ein Angreifer den *Ciphertext* verändern könnte, würde der entschlüsselte (manipulierte) Inhalt nicht mehr mit der ursprünglichen Signatur übereinstimmen, wodurch die Manipulation bei der finalen Signaturprüfung durch den Empfänger aufgedeckt würde.

Die nachfolgende *AEAD*-Verschlüsselung des Nachrichteninhalts gewährleistet dessen *Confidentiality*. Zusätzlich bietet das *AEAD*-Verfahren durch den *Tag* eine weitere, unabhängige *Integrity*-Prüfung spezifisch für den *Ciphertext* und die assoziierten

Daten (*AD*). Es ist von entscheidender Bedeutung, dass der Header mit seinen Metadaten sowohl in die Berechnung des *Hashes* für die äussere Signatur als auch als *AD* in die *AEAD*-Verschlüsselung einfliesst. Diese doppelte Bindung des Kontexts erschwert das böswillige Umleiten von Nachrichten in falsche Kontexte erheblich.

Die Nachrichtendatei (MF) wird auf der *Platform* gespeichert und enthält alle notwendigen Komponenten wie den unverschlüsselten Header, die Signaturinformation, die Liste der verschlüsselten msks und den *AEAD*-verschlüsselten Nachrichteninhalt. Die *Platform* agiert hier als reiner Speicher- und Verteildienst, ohne die Fähigkeit, die verschlüsselten Teile zu interpretieren. Der Dateiname wird aus dem Zeitstempel und der mid generiert, um Eindeutigkeit und eine chronologische Sortierung zu ermöglichen.

Sicherheitsaspekte des Nachrichtenempfangs

Der Empfang einer Nachricht erfordert, dass der *Client* des Empfängers die MF von der *Platform* abruft, ihre *Authenticity* und *Integrity* überprüft und den Inhalt entschlüsselt. Dieser Prozess wird durch den Algorithmus ReceiveMessage() (60) gesteuert.

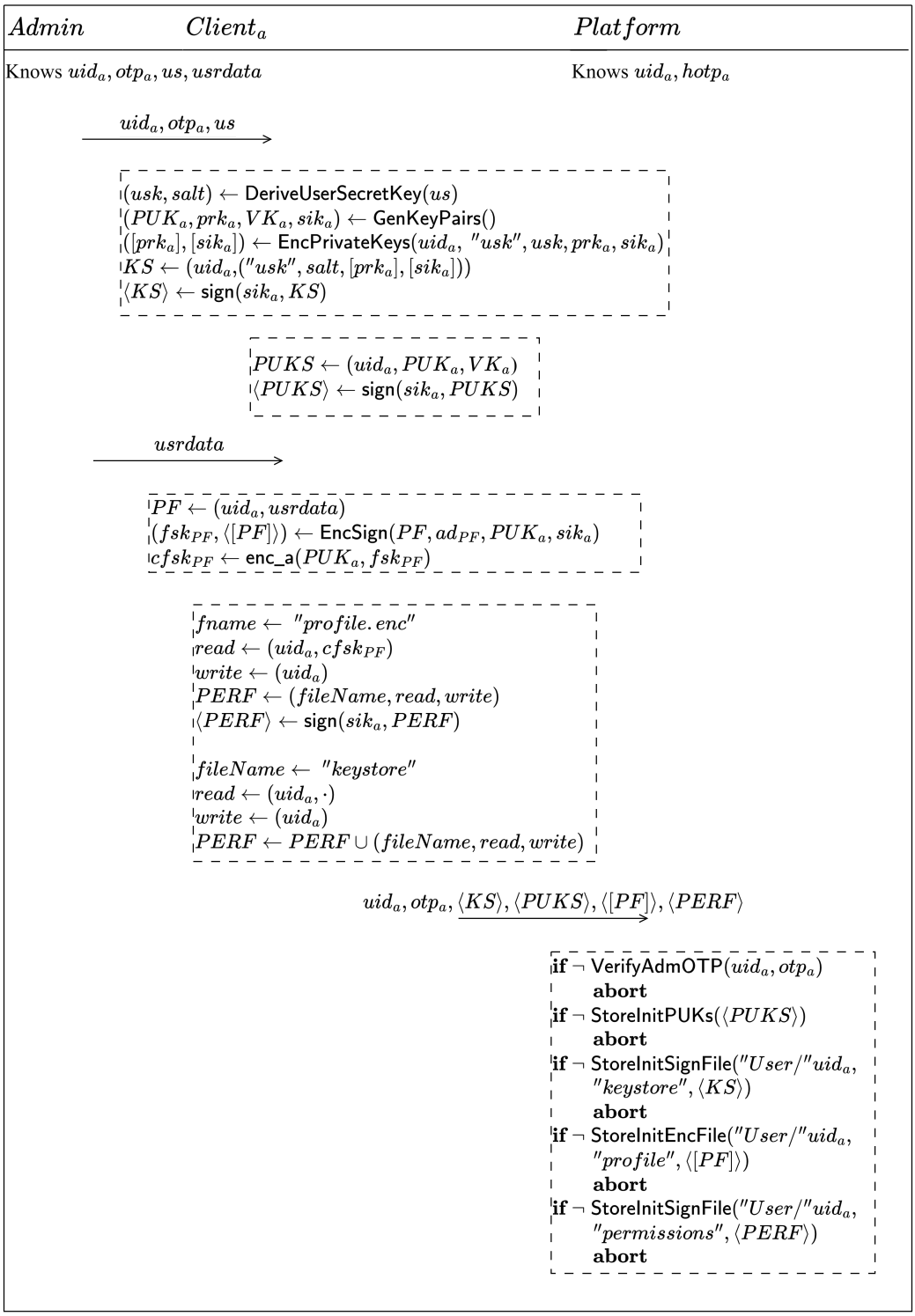
Für den Abruf und die Identifizierung des relevanten Nachrichtenschlüssels lädt der *Client* des Empfängers neue MFs aus dem entsprechenden Chat-Verzeichnis (z.B. mittels load_next_message() (5.23)). Aus der $rcpInfo$-Liste innerhalb der MF extrahiert der Empfänger den spezifisch für seine uid verschlüsselten msk ($cmsk_{recipient}$). Dieser wird mit dem privaten Entschlüsselungsschlüssel (prk) des Empfängers asymmetrisch entschlüsselt, um den Klartext-msk zu erhalten. Scheitert dieser Schritt, kann die Nachricht nicht weiterverarbeitet werden, was darauf hindeutet, dass der *User* nicht der intendierte Empfänger ist oder der $cmsk$ manipuliert wurde.

Die Entschlüsselung und Verifizierung des Inhalts erfolgt mit dem nun vorliegenden msk. Der *AEAD*-verschlüsselte Nachrichteninhalt wird entschlüsselt, wobei das *AEAD*-Verfahren gleichzeitig die *Integrity* des *Ciphertext*s und der *AD* mittels des Authentifizierungs-*Tag* prüft. Ein Fehlschlag hier deutet auf eine Manipulation des verschlüsselten Teils der Nachricht hin.

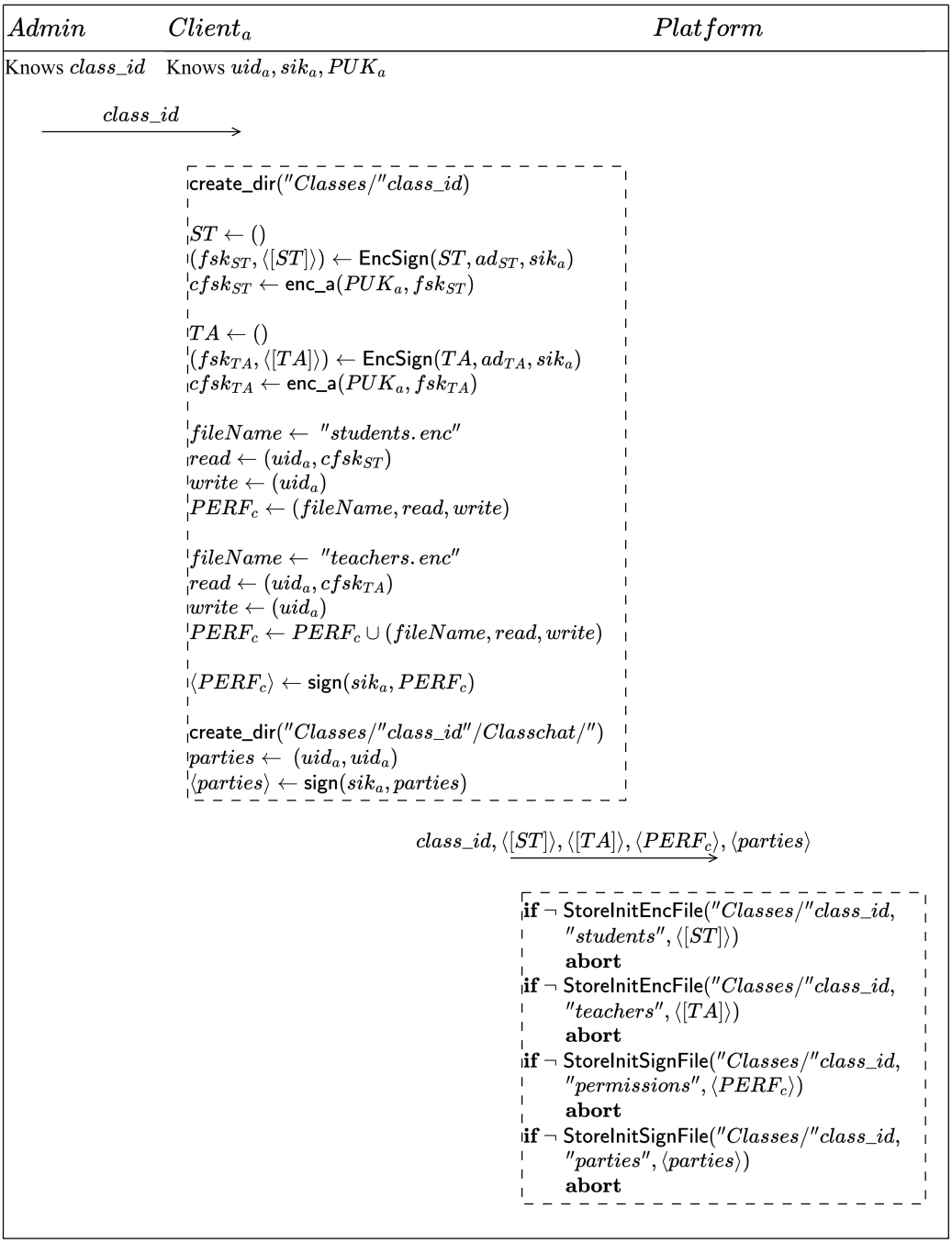
Als finale Signaturprüfung wird nach erfolgreicher Entschlüsselung des Inhalts die digitale Signatur aus der $sigInfo$-Komponente der MF überprüft. Hierzu wird der VK des Senders von der *Platform* abgerufen. Die Signatur wird über den Header und den nun entschlüsselten Nachrichteninhalt geprüft. Diese letzte Prüfung bestätigt endgültig die *Authenticity* des Senders und die *Integrity* der gesamten Nachricht. Ist eine dieser Prüfungen nicht erfolgreich, wird die Nachricht verworfen.

Schliesslich ist eine korrekte Zustandsverwaltung beim Empfänger notwendig. Um Duplikate oder das wiederholte Verarbeiten bereits gelesener Nachrichten zu vermeiden, muss der *Client* des Empfängers den Zeitstempel der zuletzt erfolgreich

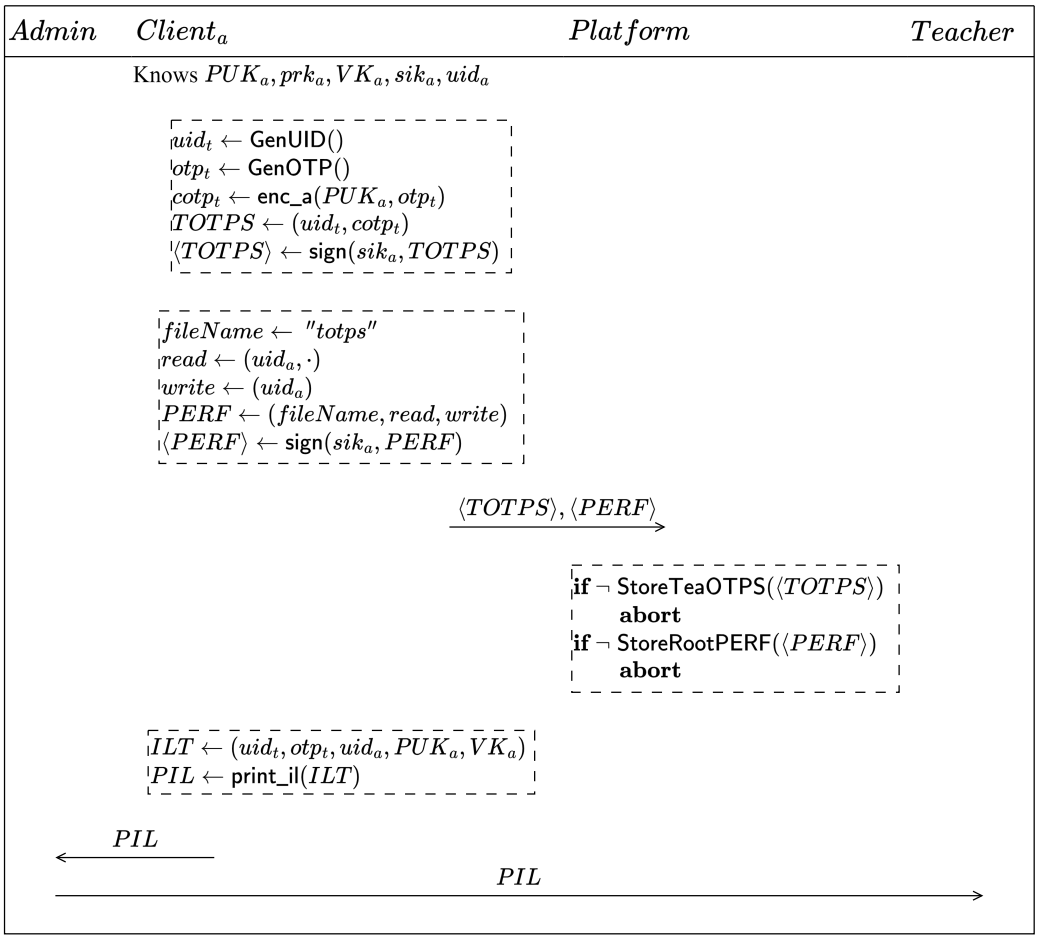
verarbeiteten Nachricht speichern und diesen Wert ($lastMsgTime$) beim nächsten Abruf neuer Nachrichten verwenden.



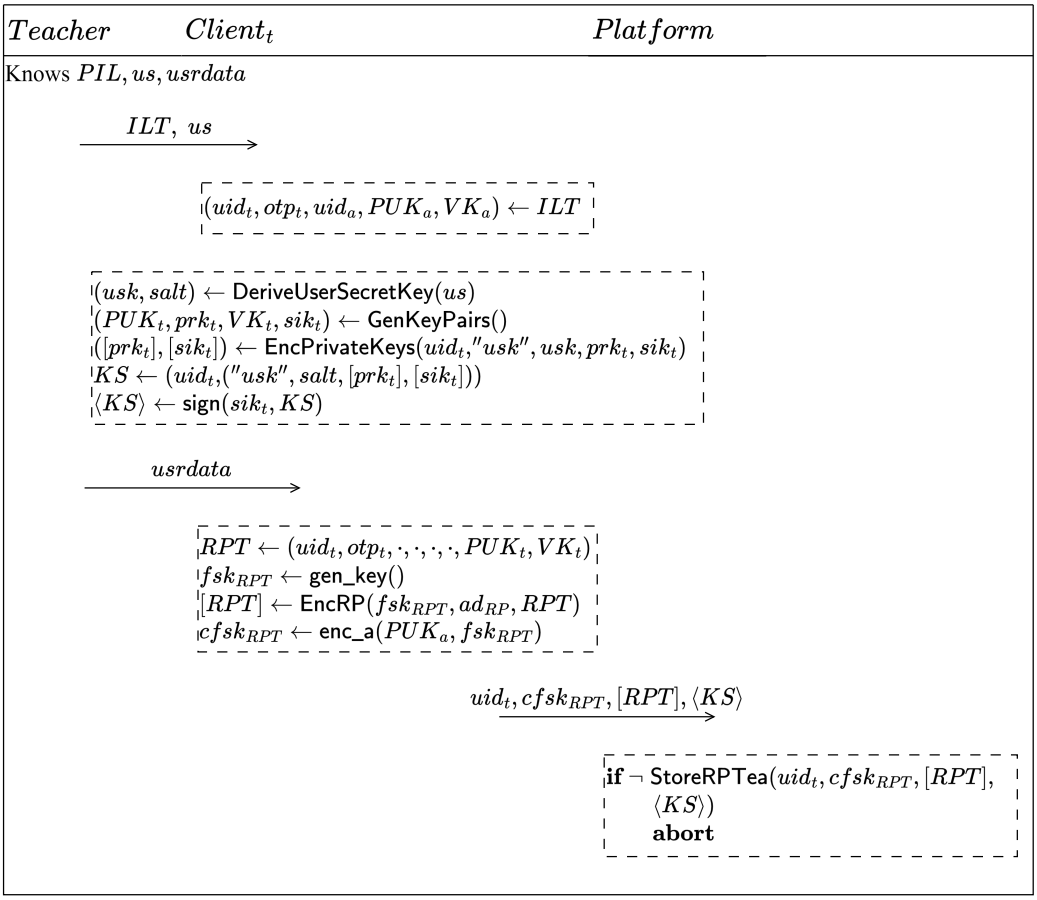
Protokoll 1: Registrierung des *Admin* auf der *Platform* durch Eingabe des *User Secret* sowie Benutzerinformationen, gefolgt von der Erstellung kryptografischer Schlüssel und der *Users*-Dateien.



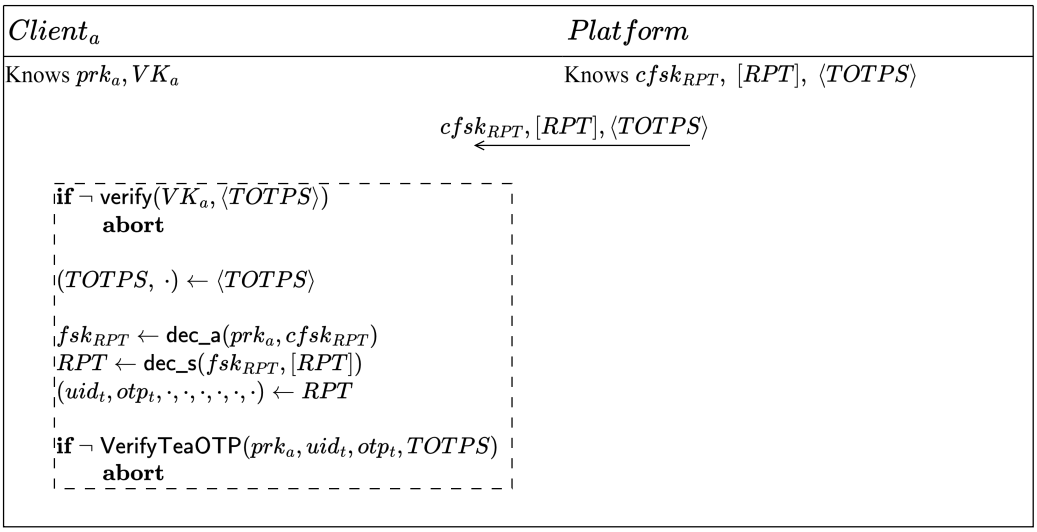
Protokoll 2: Erstellung einer Klasse auf der *Platform*, inklusive Erstellung, Verschlüsselung und Signierung der relevanten Dateien.



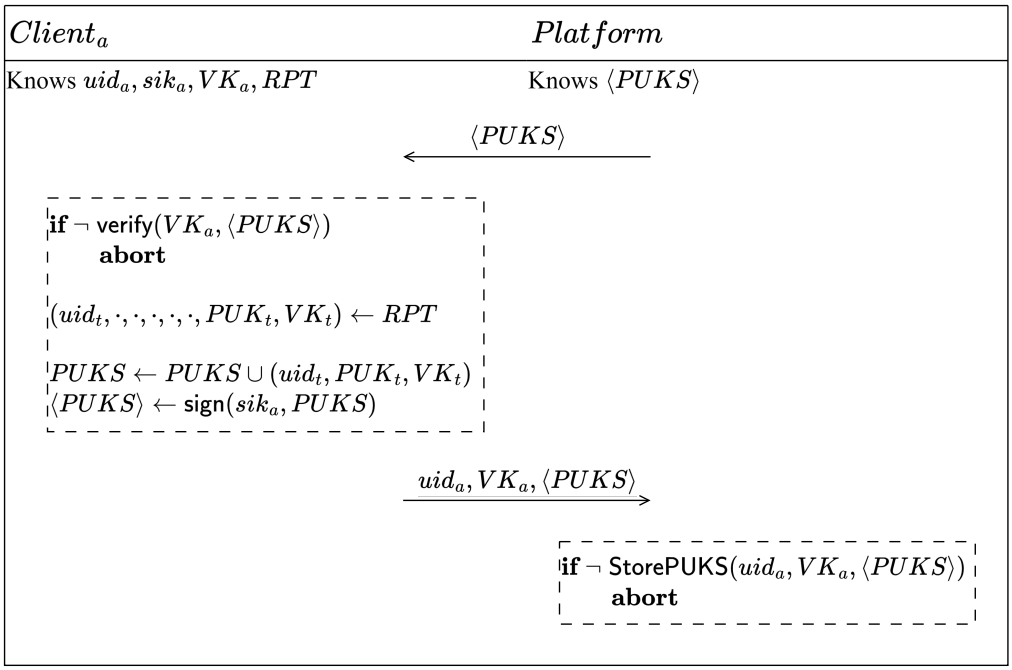
Protokoll 3: Erzeugung und Übermittlung des PIL an einen *Teacher* durch den *Admin*.



Protokoll 4: Erstellung des RPT durch den *Teacher* und dessen Ablage auf der *Platform*.

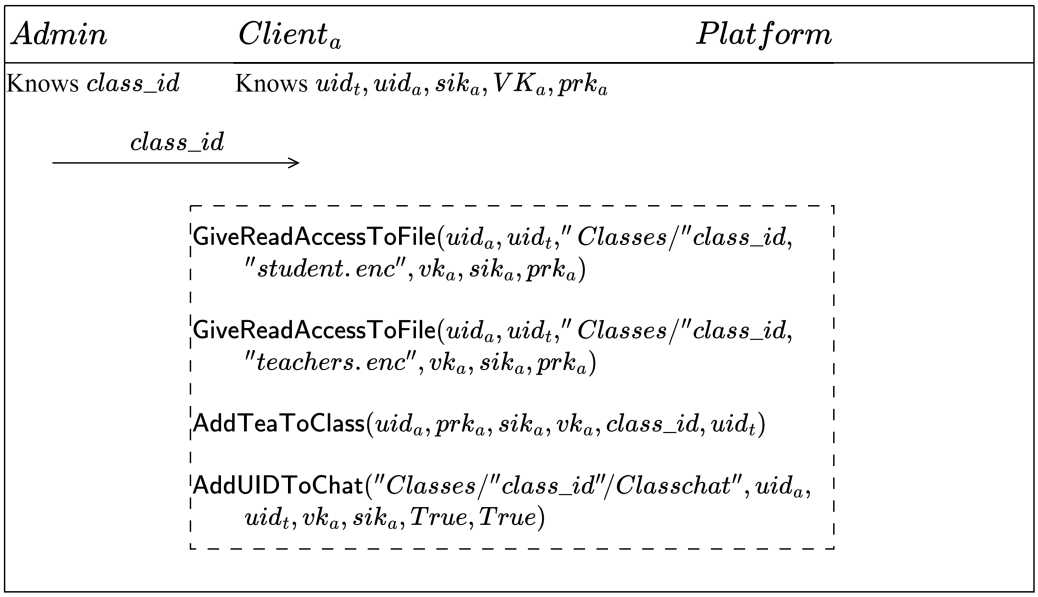


Protokoll 5: Verifikation des RPT eines *Teacher* durch den *Admin* mittels otp.



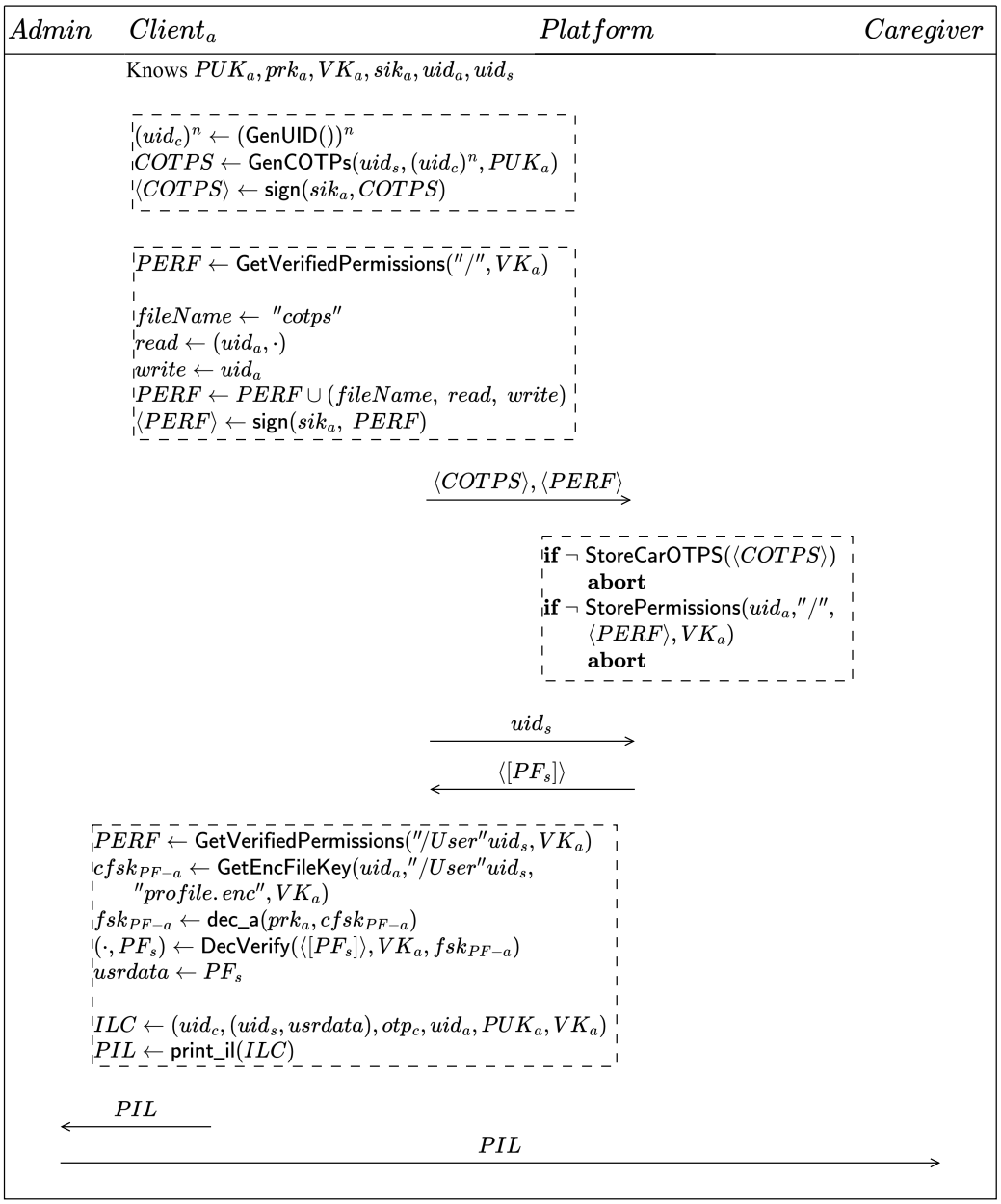
Protokoll 6: Ablage der öffentlichen Schlüssel (VK_t, PUK_t) des *Teacher* in der $PUKS$-Datei.

Protokoll 7: Ablage der Profildaten und des KS des *Teacher* auf der *Platform*.

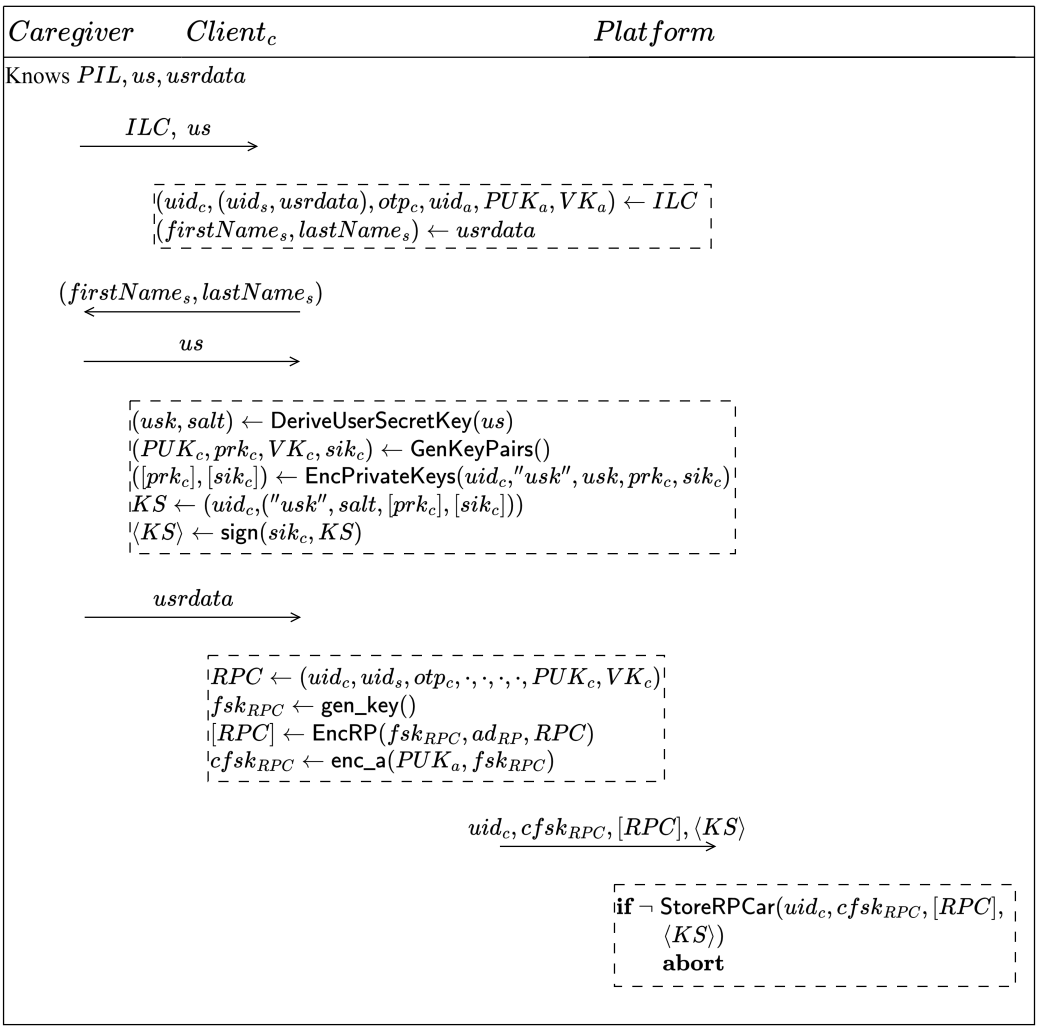


Protokoll 8: Erteilung Zugriffsberechtigungen für den *Teacher* auf eine bestimmte Klasse auf der *Platform*.

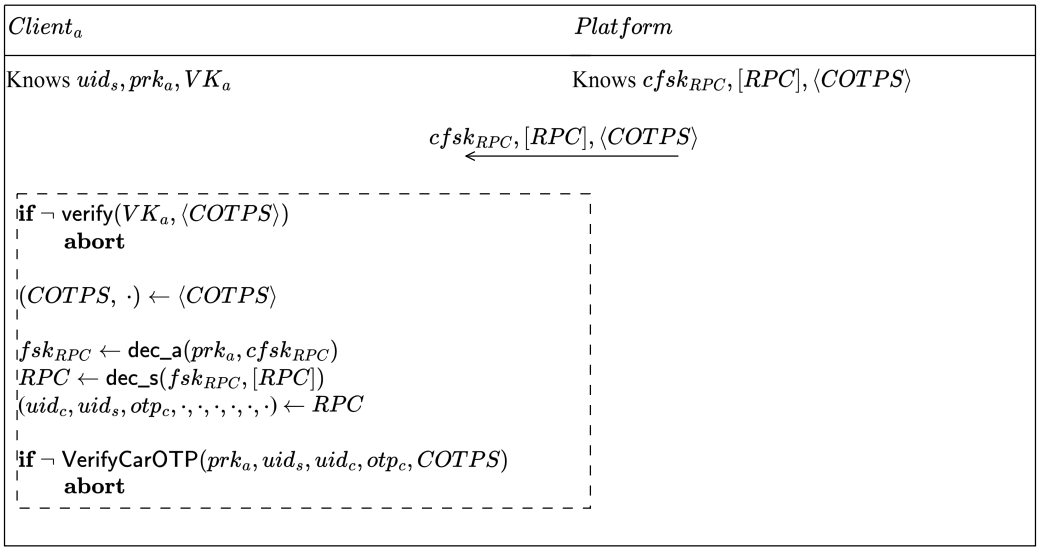
Protokoll 9: Erstellung und Speicherung der Daten eines *Student* sowie Berechtigungsvergabe an bestehende Parteien



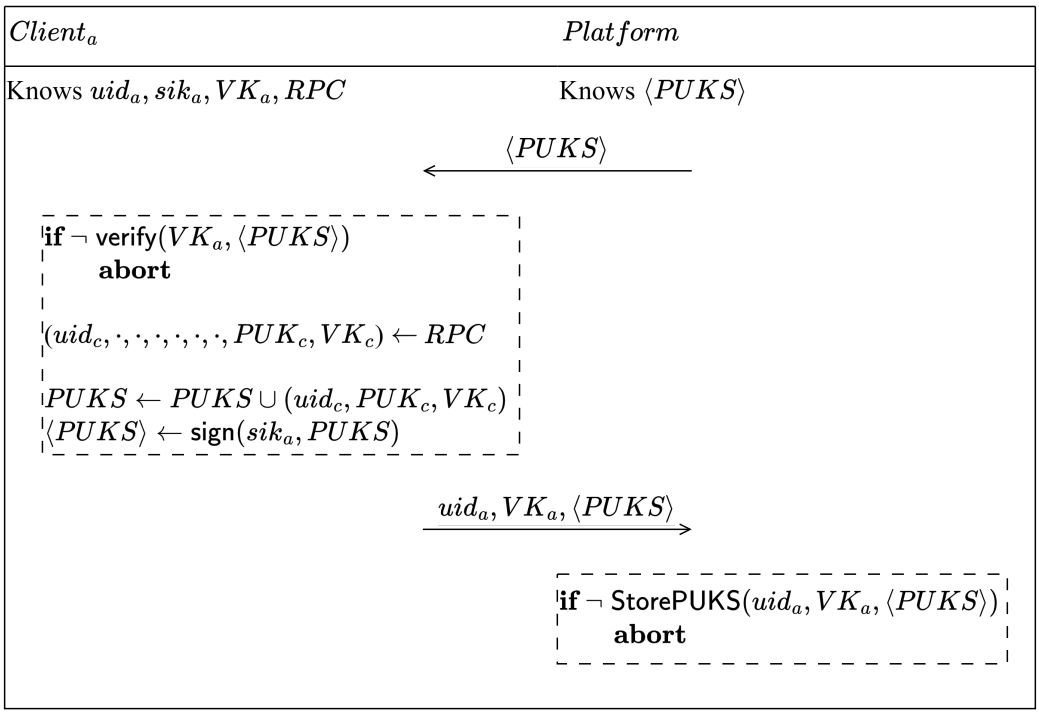
Protokoll 10: Erzeugung und Übermittlung des PIL an einen *Caregiver* durch den *Admin*.



Protokoll 11: Erstellung des RPC durch den *Caregiver* und dessen Ablage auf der *Platform*.



Protokoll 12: Verifikation des RPC eines *Caregiver* durch den *Admin* mittels otp.



Protokoll 13: Ablage der öffentlichen Schlüssel (VK_c, PUK_c) des *Caregiver* in der $PUKS$-Datei.

Protokoll 14: Ablage der Profildaten und des KS des *Caregiver* auf der *Platform*.

$Client_a$	$Platform$
Knows $uid_a, prk_a, sik_a, PUK_a, VK_a, RPC$	Knows $\langle PERF_s \rangle$

$(uid_c, uid_s, \cdot, \cdot, \cdot, \cdot, \cdot, PUK_c, \cdot) \leftarrow RPC$

$PERF_s \leftarrow$ GetVerifiedPermissions$("/User"uid_s, VK_a)$

$CG \leftarrow (uid_c)$

$(fsk_{CG}, \langle [CG] \rangle) \leftarrow$ EncSign(CG, ad_{CG}, sik_a)
$cfsk_{CG-a} \leftarrow$ enc_a(PUK_a, fsk_{CG})
$cfsk_{CG-c} \leftarrow$ enc_a(PUK_c, fsk_{CG})

$cfsk_{PF-a} \leftarrow$ GetEncFileKey$(uid_a,"/User"uid_s,"profile.enc", VK_a)$
$fsk_{PF} \leftarrow$ dec_a$(prk_a, cfsk_{PF-a})$
$cfsk_{PF-c} \leftarrow$ enc_a(PUK_c, fsk_{PF})

$cfsk_{TA-a} \leftarrow$ GetEncFileKey$(uid_a,"/User"uid_s,"teachers.enc", VK_a)$
$fsk_{TA} \leftarrow$ dec_a$(prk_a, cfsk_{TA-a})$
$cfsk_{TA-c} \leftarrow$ enc_a(PUK_c, fsk_{TA})

$fileName \leftarrow$ $"caregivers.enc"$
$read \leftarrow ((uid_a, cfsk_{CG-a}), (uid_c, cfsk_{CG-c}))$
$write \leftarrow (uid_a)$
$PERF_s \leftarrow PERF_s \cup (fileName, read, write)$

$fileName \leftarrow$ $"profile.enc"$
$read \leftarrow (uid_c, cfsk_{PF-c})$
$write \leftarrow (uid_c)$
$PERF_s \leftarrow PERF_s \cup (fileName, read, write)$

$fileName \leftarrow$ $"teachers"$
$read \leftarrow (uid_c, cfsk_{TA-c})$
$PERF_s \leftarrow PERF_s \cup (fileName, read, \cdot)$

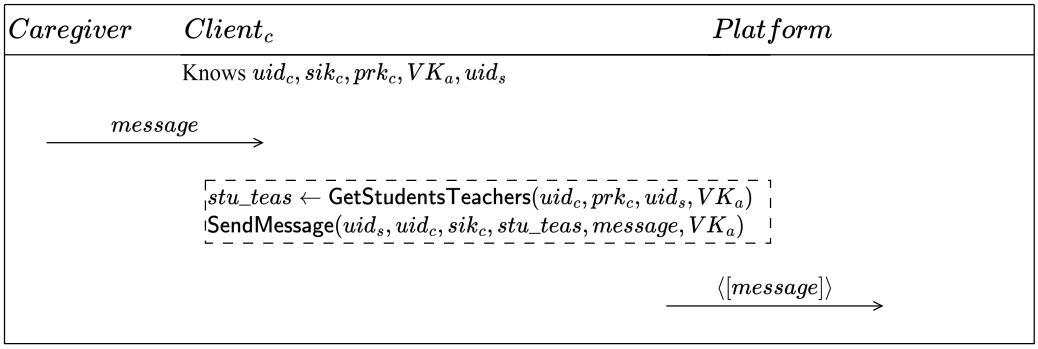
$\langle PERF_s \rangle \leftarrow$ sign$(sik_a, PERF_s)$

$\xrightarrow{uid_s, \langle [CG] \rangle, \langle PERF_s \rangle}$

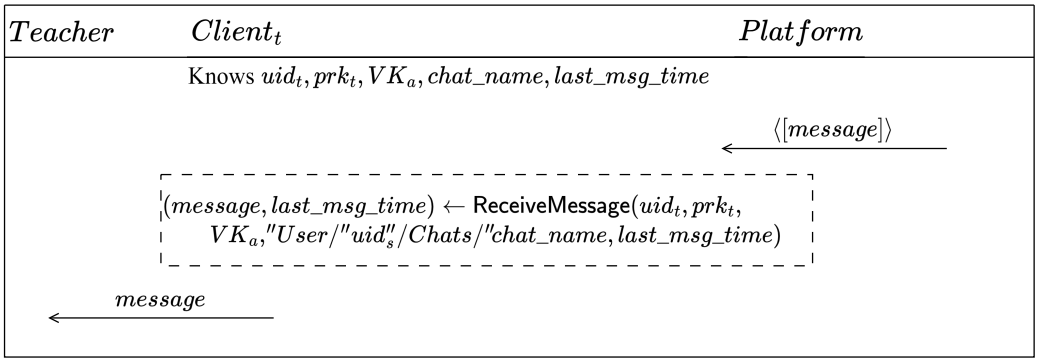
if $\neg$ StorePermissions$(uid_a,"User/"uid_s, \langle PERF_s \rangle, VK_a)$
abort
if $\neg$ StoreInitEncFile$("User/"uid_s, "caregivers", \langle [CG] \rangle)$
abort

Protokoll 15: Etablierung der Beziehung zwischen *Caregiver* und *Student* durch Aktualisierung der $PERF$-Datei des *Students* und Erstellung der CG-Datei.

Protokoll 16: Erteilung Zugriffsberechtigungen für den *Caregiver* auf eine bestimmte Klasse auf der *Platform*.



Protokoll 17: Ein *Caregiver* versendet eine Nachricht an einen *Teacher*. Die Nachricht wird verschlüsselt sowie signiert auf der *Platform* abgelegt.



Protokoll 18: Ein *Teacher* erhält eine für ihn von *Caregiver* erstellte Nachricht aus der *Platform*.

5.3. Pseudo-Code-Algorithmen

In diesem Kapitel sind alle Algorithmen aufgeführt, die für das Funktionieren der in Kapitel 5.2 beschriebenen Protokollschritte erforderlich sind. Sie sind als modulare Bausteine konzipiert und dienen der präzisen, aber implementierungsunabhängigen Beschreibung einzelner Funktionen und Operationen.

Jeder Algorithmus ist so formuliert, dass er sich direkt in modernen Programmiersprachen abbilden lässt. Der Fokus liegt dabei auf Klarheit, Konsistenz und formaler Typisierung.

In Tabelle 5.23 sind ergänzend zu den kryptographischen Black-Box-Algorithmen (vgl. 5.1.7) weitere Black-Box-Algorithmen aufgeführt, die erst im Rahmen einer konkreten Implementierung definiert werden können.

Algorithmus	**Beschreibung**
encode_uid()	Nimmt ein Byte-Array r als Eingabe und kodiert dieses als Base32-Zeichenkette ohne Padding. Das Ergebnis dient als menschenlesbare, plattformkompatible Benutzerkennung.
encode_otp()	Nimmt ein Byte-Array r als Eingabe und kodiert dieses als Base32-Zeichenkette ohne Padding. Das Ergebnis dient als menschenlesbarer Einmalcode zur Authentifizierung.
encode_mid()	Kodiert ein Byte-Array r als URL-sichere Base32-Zeichenkette ohne Padding. Dient als menschenlesbarer, dateinamen-geeigneter mid.
store_file()	Nimmt den Pfad ($folderPath$), den Dateinamen ($fileName$) sowie einen String ($data$), welcher die zu speichernden Daten enthält, als Eingabe und speichert die resultierende Datei auf der *Platform*.
load_file()	Lädt eine Datei mit dem Namen ($fileName$) aus dem Pfad ($folderPath$) von der *Platform* herunter. Wird für das Einlesen strukturierter Protokolldaten verwendet, die mit store_file gespeichert wurden.
delete_file()	Löscht eine Datei mit dem Namen ($fileName$) aus dem Pfad ($folderPath$) von der *Platform*.
create_dir()	Erstellt unter dem Pfad $folderPath$ den gegebenen Ordner auf der *Platform*.

Algorithmus	**Beschreibung**
print_il()	Formatiert die *Invitation Letter Caregviver*- und *Invitation Letter Teacher*-Datenstrukturen in ein druckbares Format und druckt diese.
extract_substring_from_filename()	Extrahiert und retourniert eine uid oder eine andere spezifische Information aus einem gegebenen Dateinamen ($fileName$), der zwischen dem gegebenen Präfix ($prefix$) und Suffix ($suffix$) liegt.
find_arbitrary_rpt_filename()	Gibt den $fileName$ ein beliebiges RPT zurück, wenn auf der *Platform* eines vorhanden ist.
find_arbitrary_rpc_filename()	Gibt den $fileName$ ein beliebiges RPC zurück, wenn auf der *Platform* eines vorhanden ist.
gen_timestamp()	Liefert einen aktuellen Zeitstempel als Integer (z.B. Unix-Timestamp in Sekunden).
load_all_file_names()	Lädt die Namen aller regulären Dateien im angegebenen $folderPath$. Gibt eine Liste von Dateinamen zurück.
load_next_message()	Lädt den Inhalt der nächsten Nachrichtendatei aus $folderPath$, deren Zeitstempel im Dateinamen grösser als $lastMsgTime$ ist und den kleinsten solchen Zeitstempel hat.
get_chat_folder()	Bestimmt den Pfad zum Chat-Ordner basierend auf der uid des *Student* und dem Nachrichten-Header $header$.

Tabelle 5.23.: Beschreibung der in den folgenden Algorithmen verwendeten Black-Box-Algorithmen

Die Pseudo-Code-Algorithmen in diesem Abschnitt werden in den Beschreibungen zu den Protokollschritten in Kapitel 5.2 referenziert. Zu Beginn jedes Unterabschnitts befindet sich eine einleitende Tabelle, welche alle in diesem Abschnitt definierten Algorithmen auflistet, inklusive kurzer Beschreibung und Angabe der zugehörigen Protokollschritte oder verwendenden Algorithmen.

An dieser Stelle sei angemerkt, dass der Sicherheitsparameter λ als globale Konstante betrachtet wird und somit allen Algorithmen implizit zur Verfügung steht. Das heisst, er muss nicht explizit als Parameter übergeben werden.

5.3.1. Allgemeine Algorithmen

Die in diesem Abschnitt spezifizierten allgemeinen Algorithmen bilden fundamentale Bausteine für die im Rahmen dieser Arbeit entwickelten Protokolle. Sie umfassen grundlegende Operationen wie die Generierung von Zufallswerten, eindeutigen Identifikatoren (uids, otps, mids), kryptographischen *Nonce*s und *Salt*s. Des Weiteren werden Algorithmen für die symmetrische Verschlüsselung von Registrierungspaketen sowie die kombinierte Verschlüsselung und Signierung von Daten definiert, einschliesslich der zugehörigen Verifikations- und Entschlüsselungsprozesse. Diese Operationen werden unterstützt durch Verfahren zur Überprüfung von Lese- und Schreibzugriffen, welche die in $PERF$-Dateien hinterlegten Berechtigungen validieren. Tabelle 5.24 gibt einen umfassenden Überblick über diese Algorithmen, ihre spezifischen Funktionen und ihre Verwendung innerhalb der definierten Protokolle und anderer Algorithmen.

Nr.	Algorithmus	Beschreibung	Use
1	GenRandomBytes()	Generiert einen gleichverteilten Bytestring der spezifizierten Länge ℓ.	Alg. 2, 3, 4, 5 , 6
2	GenUID()	Generiert eine zufällige, eindeutige uid.	Prot. 3, 9, 10
3	GenOTP()	Generiert ein zufälliges, einmaliges otp.	Prot. 3, Alg. 55
4	GenNonce()	Generiert eine zufällige, einmaliges *Nonce* für AEAD-Operationen.	Alg. 7, 8, 9, 59, 48
5	GenSalt()	Generiert einen zufälligen *Salt* zur Verwendung in Schlüsselableitungsfunktionen.	Alg. 46
6	GenMID()	Generiert eine zufällige, eindeutige mid.	Alg. 59
7	EncRP()	Verschlüsselt ein RPC oder RPT symmetrisch (*AEAD*) mit einem gegebenen Dateischlüssel (fsk) und *AD*. Gibt das verschlüsselte RPC oder RPT als *AEAD*-Tupel zurück.	Prot. 4, 11

Nr.	Algorithmus	Beschreibung	Use
8	EncSign()	Verschlüsselt Daten symmetrisch (*AEAD*) und signiert das Ergebnis. Gibt den verwendeten Dateischlüssel (fsk) und den signierten *Ciphertext* zurück.	Prot. 1, 2, 9, 7, 14, 15
9	EncSignWithKey()	Verschlüsselt Daten symmetrisch (*AEAD*) mit gegebenem Schlüssel fsk und *AD* und signiert das Ergebnis. Gibt das *AEAD*-Tupel und die *signature* (sig) zurück.	Alg. 42, 43
10	DecVerify()	Verifiziert die sig von *AEAD*-verschlüsselten Daten und entschlüsselt diese bei Erfolg.	Prot. 10, Alg. 42, 43, 44, 45
11	VerifyWriteAccess()	Überprüft, ob eine angegebene uid Schreibrechte für eine Datei gemäss der $PERF$-Datei besitzt und ob die sig der zu schreibenden Daten gültig ist.	Alg. 33, 35
12	VerifyReadAccess()	Prüft, ob ein *User* gemäss der $PERF$-Datei Leserechte für eine bestimmte Datei besitzt.	Alg. 34, 36
13	GetEncFileKey()	Ermittelt den verschlüsselten Dateischlüssel (($cfsk$)) für einen leseberechtigten *User* aus der $PERF$-Datei.	Prot. 10, 15, 16, Alg. 41, 42, 43, 44, 45

Tabelle 5.24.: Übersicht der allgemeinen Algorithmen mit Beschreibung ihrer Funktionalität und Referenzierung ihrer Verwendung in Protokollen und anderen Algorithmen.

Generierung von Zufallsbytes

GenRandomBytes() ist eine grundlegende Hilfsfunktion zur Erzeugung eines Byte-Strings gegebener Länge mit zufälligen, gleichverteilten Werten.

Algorithm 1: GenRandomBytes(ℓ)

Input: Length $\ell \in \mathbb{N}^+$

$r \xleftarrow{\$} \{0,1\}^{8\ell}$

return r

Generierung einer User ID

GenUID() erzeugt einen eindeutigen *User Identifier*.

Algorithm 2: GenUID()

Constraints: $\lambda \bmod 8 = 0$

$r \leftarrow$ GenRandomBytes$(\lambda/8)$ `// see Alg. 1`
$uid \leftarrow$ encode_uid(r)

return uid `//` $uid \in \mathcal{U}$

Generierung eines One-Time Passwords

GenOTP() erzeugt ein *One Time Password*.

Algorithm 3: GenOTP()

Constraints: $\lambda \bmod 8 = 0$

$r \leftarrow$ GenRandomBytes$(\lambda/8)$ `// see Alg. 1`
$otp \leftarrow$ encode_otp(r)

return otp `//` $otp \in \mathcal{O}$

Generierung einer Nonce

GenNonce() generiert einen *Nonce* für *AEAD*-Verschlüsselung.

Algorithm 4: GenNonce()

Constraints: $\lambda \bmod 8 = 0$

$nonce \leftarrow$ GenRandomBytes$(\lambda/8)$ `// see Alg. 1`

return $nonce$ `//` $nonce \in \mathcal{N}$

Generierung eines Salt

GenSalt() generiert einen zufälligen *Salt*, beispielsweise zur Derivation von Schlüsseln.

Algorithm 5: GenSalt()

Constraints: $\lambda \bmod 8 = 0$

$salt \leftarrow$ GenRandomBytes$(\lambda/8)$ `// see Alg. 1`

return $salt$ `//` $salt \in \mathcal{R}$

Generierung einer Message ID

GenMID() erzeugt einen eindeutigen mid.

Algorithm 6: GenMID()

Constraints: $\lambda \bmod 8 = 0$

$r \leftarrow$ GenRandomBytes$(\lambda/8)$ `// see Alg. 1`
$mid \leftarrow$ encode_mid(r)

return mid `//` $mid \in \mathcal{M}$

Verschlüsseln eines Registrierungspakets

EncRP() verschlüsselt ein gegebenes RPC oder RPT symmetrisch mittels *AEAD*.

Algorithm 7: EncRP(fsk, ad_{RP}, RP)

Input: File secret key $fsk \in \mathcal{K}$
Associated data for the RP $ad_{RP} \in \mathcal{A}$
Registration package (Plaintext) $RP \in \mathcal{D}$

$nonce_{RP} \leftarrow$ GenNonce() `// see Alg. 4`
$(ct_{RP}, tag_{RP}) \leftarrow$ enc_s$(fsk, nonce_{RP}, ad_{RP}, RP)$
$[RP_{enc}] \leftarrow (nonce_{RP}, ad_{RP}, ct_{RP}, tag_{RP})$

return $[RP_{enc}]$ `//` $[RP_{enc}] \in \mathcal{N} \times \mathcal{A} \times \mathcal{C} \times \mathcal{T}$

Verschlüsseln und Signieren von Daten

EncSign() kombiniert symmetrische *AEAD*-Verschlüsselung mit einer digitalen Signatur, um *Confidentiality* und *Authenticity* sicherzustellen.

Algorithm 8: EncSign(p, ad_p, sik)

Input: Plaintext $p \in \mathcal{D}$
Associated data $ad_p \in \mathcal{A}$
Signing key $sik \in \mathcal{K}_{sik}$

$fsk \leftarrow$ gen_key()
$nonce_p \leftarrow$ GenNonce() `// see Alg. 4`
$(ct_p, tag_p) \leftarrow$ enc_s$(fsk, nonce_p, ad_p, p)$
$[d] \leftarrow (nonce_p, ad_p, ct_p, tag_p)$
$sig \leftarrow$ sign$(sik, [d])$
$\langle [d] \rangle \leftarrow ([d], sig)$

return $(fsk, \langle [d] \rangle)$ `//` $(fsk, \langle [d] \rangle) \in \mathcal{K} \times ((\mathcal{N} \times \mathcal{A} \times \mathcal{C} \times \mathcal{T}) \times \mathcal{S})$

Verschlüsseln mit gegebenem Schlüssel und Signieren von Daten

EncSignWithKey() kombiniert symmetrische *AEAD*-Verschlüsselung mit einer digitalen Signatur, wobei der symmetrische Schlüssel als Eingabeparameter dient. Dies unterscheidet ihn von EncSign(), welcher den Schlüssel intern generiert.

Algorithm 9: EncSignWithKey(p, ad_p, fsk, sik)

Input: Plaintext $p \in \mathcal{D}$
Associated data $ad_p \in \mathcal{A}$
File secret key $fsk \in \mathcal{K}$
Signing key $sik \in \mathcal{K}_{sik}$

$nonce_p \leftarrow$ GenNonce() `// see Alg. 4`
$(ct_p, tag_p) \leftarrow$ enc_s$(fsk, nonce_p, ad_p, p)$
$[d] \leftarrow (nonce_p, ad_p, ct_p, tag_p)$
$sig \leftarrow$ sign$(sik, [d])$
$\langle [d] \rangle \leftarrow ([d], sig)$

return $\langle [d] \rangle$ // $\langle [d] \rangle \in (\mathcal{N} \times \mathcal{A} \times \mathcal{C} \times \mathcal{T}) \times \mathcal{S}$

Entschlüsseln und Verifizieren von Daten

DecVerify() prüft die Signatur von verschlüsselten Daten und entschlüsselt diese bei erfolgreicher Verifikation.

Algorithm 10: DecVerify(c, sig, VK, fsk)

Input: Ciphertext $c = (n, ad, ct, tag) \in \mathcal{N} \times \mathcal{A} \times \mathcal{C} \times \mathcal{T}$
Signature $sig \in \mathcal{S}$
Verification key $VK \in \mathcal{K}_{VK}$
File secret key $fsk \in \mathcal{K}$

if $\neg$verify(VK, c, sig) **then**
  return $\perp$

$(nonce_p,\ ad_p,\ ct_p,\ tag_p) \leftarrow c$
$p \leftarrow$ dec_s$(fsk,\ nonce_p,\ ad_p,\ ct_p,\ tag_p)$

return (ad_p, p) // $(ad_p, p) \in \mathcal{A} \times \mathcal{D}$

Verifizieren des Schreibzugriffs

VerifyWriteAccess() prüft, ob ein *User* die Berechtigung hat, eine bestimmte Datei zu schreiben und ob die Signatur der Daten korrekt ist.

Algorithm 11: VerifyWriteAccess($uid, folderPath, fileName, \langle data \rangle, VK_a$)

Input: *User Identifier* of the writer $uid \in \mathcal{U}$
Path of the folder where the file is lying $folderPath \in \Sigma^*$
Filename $fileName \in \Sigma^*$
Signed plaintext or ciphertext
$\langle data \rangle = (data, sig) \in ((\mathcal{D}) \cup (\mathcal{N} \times \mathcal{A} \times \mathcal{C} \times \mathcal{T})) \times \mathcal{S}$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$

$PERF \leftarrow$ GetVerifiedPermissions($folderPath, VK_a$) `// see Alg. 32`
$valid \leftarrow false$

foreach $p \in PERF$ **do**
  $(p\_fileName, \cdot, (uid)^n) \leftarrow p$
  if $p\_fileName = fileName$ **then**
    foreach $u \in (uid)^n$ **do**
      if $u = uid$ **then**
        $valid \leftarrow true$

if $valid$ **then**
  $VK \leftarrow$ GetVK(uid, VK_a) `// see Alg. 50`
  $(data, sig) \leftarrow \langle data \rangle$
  $valid \leftarrow$ verify($VK, data, sig$)

return $valid$ `//` $valid \in \{$true, false$\}$

Verifizieren des Lesezugriffs

VerifyReadAccess() prüft, ob ein *User* die Berechtigung hat, eine bestimmte Datei zu lesen.

Algorithm 12: VerifyReadAccess($uid, folderPath, fileName, VK_a$)

Input: *User Identifier* of the reader $uid \in \mathcal{U}$
Path of the folder where the file is lying $folderPath \in \Sigma^*$
Filename to read $fileName \in \Sigma^*$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$

$PERF \leftarrow$ GetVerifiedPermissions($folderPath, VK_a$) `// see Alg. 32`
$valid \leftarrow false$

foreach $p \in PERF$ **do**
  $(p\_fileName, (uid, \cdot)^n, \cdot) \leftarrow p$
  if $p\_fileName = fileName$ **then**
    foreach $u \in (uid, \cdot)^n$ **do**
      $(u\_uid, \cdot) \leftarrow u$
      if $u\_uid = uid$ **then**
        $valid \leftarrow true$

return $valid$ `//` $valid \in \{$true, false$\}$

Abrufen eines verschlüsselten Dateischlüssels

GetEncFileKey() extrahiert den für einen bestimmten *User* verschlüsselten Dateischlüssel ($cfsk$) aus der $PERF$-Datei.

Algorithm 13: GetEncFileKey($uid, folderPath, fileName, VK_a$)

Input: *User Identifier* of the reader $uid \in \mathcal{U}$
Path of the folder where the file is lying $folderPath \in \Sigma^*$
Filename to read $fileName \in \Sigma^*$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$

$PERF \leftarrow$ GetVerifiedPermissions($folderPath, VK_a$) `// see Alg. 32`

foreach $p \in PERF$ **do**
  $(p\_fileName, (uid, cfsk)^n, \cdot) \leftarrow p$
  if $p\_fileName = fileName$ **then**
    foreach $u \in (uid, cfsk)^n$ **do**
      $(u\_uid, cfsk) \leftarrow u$
      if $u\_uid = uid$ **then**
        return $cfsk$ `//` $cfsk \in \mathcal{C}$

5.3.2. Algorithmen mit Seiteneffekten

Die in Tabelle 5.25 aufgeführten Verfahren unterscheiden sich von rein kryptographischen Algorithmen, da sie keine neuen kryptographischen Werte generieren, sondern primär durch ihre Interaktion mit dem Systemzustand charakterisiert sind. Sie besitzen Seiteneffekte, indem sie beispielsweise Dateioperationen wie das Speichern und Laden von globalen öffentlichen Schlüsseln ($PUKS$), Berechtigungsdateien ($PERF$) oder verschlüsselten sowie signierten Dateien ausführen. Die Ausführung solcher Operationen kann von externen Bedingungen abhängen und beispielsweise bei beschädigten Dateien oder unzureichenden Zugriffsrechten fehlschlagen, wodurch diese Abläufe nicht vollständig deterministisch sind.

Trotz dieser Unterschiede werden diese Verfahren hier als Algorithmen bezeichnet und formal spezifiziert, da sie eng mit den Sicherheitsmechanismen der *Platform* verwoben sind, indem sie beispielsweise die korrekte Verifizierung von Signaturen und Zugriffsberechtigungen vor der Ausführung von Dateioperationen sicherstellen. Ihre korrekte und sichere Implementierung ist daher fundamental für die *Integrity* und *Confidentiality* des Gesamtsystems.

Einige dieser Algorithmen werden derzeit von keinem Protokollschritt und keinem anderen Algorithmus verwendet. Sie bleiben dennoch in der Tabelle enthalten, da sie für zukünftige Operationen, etwa zum Lesen von Dateien, relevant sein werden.

Nr.	Algorithmus	Beschreibung	Prot./Alg.
14	StoreInitPUKs()	Initialisiert und speichert die $PUKS$-Datei. Geht von erfolgreicher *Admin*-otp-Verifikation aus.	Prot. 1
15	StoreInitEncFile()	Initialisiert und speichert eine verschlüsselte und signierte Datei für einen *User*.	Prot. 1, 2, 9, 7, 14, 15
16	StoreInitSignFile()	Initialisiert und speichert eine nicht verschlüsselte Datei für einen *User*.	Prot. 1, 2, 7, 9, 14
17	StoreTeaOTPS()	Speichert die $TOTPS$-Datei, signiert vom *Admin*.	Prot. 3
18	LoadTOTPS()	Lädt die $TOTPS$-Datei der *Teacher* und deren Signatur. Die Verifikation erfolgt separat.	Prot. 5
19	StoreCarOTPS()	Speichert die $COTPS$-Datei, signiert vom *Admin*.	Prot. 10
20	LoadCOTPS()	Lädt die $COTPS$-Datei der *Caregiver* und deren Signatur. Die Verifikation erfolgt separat.	Prot. 12
21	StoreRootPERF()	Speichert die $PERF$-Datei im Root-Verzeichnis, signiert vom *Admin*.	Prot. 3
22	StoreRPTea()	Speichert das Registrierungspaket (RPT) und den Keystore (KS) eines *Teachers*.	Prot. 4
23	LoadRPTea()	Lädt das verschlüsselte Registrierungspaket (RPT) und den signierten Keystore (KS) eines spezifischen *Teachers* von der *Platform*.	Prot. 5
24	DeleteRPTea()	Löscht das Registrierungspaket (RPT) und den zugehörigen Keystore (KS) eines *Teachers* von der *Platform*.	Prot. 7

Nr.	Algorithmus	Beschreibung	Prot./Alg.
25	StoreRPCar()	Speichert das Registrierungspaket (RPC) und den Keystore (KS) eines *Caregivers*.	Prot. 11
26	LoadRPCar()	Lädt das verschlüsselte Registrierungspaket (RPC) und den signierten Keystore (KS) eines spezifischen *Caregivers* von der *Platform*.	Prot. 12
27	DeleteRPCar()	Löscht das Registrierungspaket (RPC) und den zugehörigen Keystore (KS) eines *Caregivers* von der *Platform*.	Prot. 14
28	StorePUKS()	Speichert die globalen öffentlichen Schlüssel ($PUKS$) und deren Signatur, nachdem der Schreibzugriff des *Admins* verifiziert wurde.	Prot. 6, 13
29	LoadPUKS()	Lädt die globale $PUKS$-Datei und deren Signatur.	Prot. 6, 13
30	GetVerifiedPUKS()	Lädt die globale $PUKS$-Datei und deren Signatur und verifiziert sie mit dem *Admin*-VK.	Alg. 49, 50
31	StorePermissions()	Speichert eine $PERF$-Datei und deren Signatur, nachdem der *Admin*-Zugriff verifiziert wurde.	Prot. 10, 15, 16, Alg. 41
32	GetVerifiedPermissions()	Lädt eine $PERF$-Datei und deren Signatur, verifiziert die Signatur mit dem *Admin*-VK und gibt die $PERF$-Datei zurück.	Alg. 11, 12, 13, 41, 42, 43, 44, 45, Prot. 10, 15, 16
33	StoreEncFile()	Speichert eine verschlüsselte und signierte Datei nachdem der Schreibzugriff verifiziert wurde.	Alg. 42, 43
34	LoadEncFile()	Lädt eine verschlüsselte Datei und ihre Signatur aus dem Speicher, nachdem der Lesezugriff verifiziert wurde.	Alg. 42, 43, 44, 45

Nr.	Algorithmus	Beschreibung	Prot./Alg.
35	StoreSignFile()	Speichert eine nicht verschlüsselte Datei und deren Signatur nachdem der Schreibzugriff verifiziert wurde.	-
36	LoadSignFile()	Lädt eine nicht verschlüsselte Datei und ihre Signatur nachdem der Lesezugriff verifiziert wurde.	-
37	CreateUserDir()	Erstellt ein Benutzerverzeichnis auf der *Platform*.	Prot. 7, 9, 14
38	StoreParties()	Speichert die `parties`-Datei und deren Signatur in einem Chat-Ordner, nachdem der *Admin*-Zugriff verifiziert wurde.	Alg. 40
39	GetVerifiedParties()	Lädt und verifiziert die signierte `parties`-Datei aus einem Chat-Ordner.	Alg. 40
40	AddUIDToChat()	Fügt eine uid zur `parties`-Datei eines Chats hinzu (Lese-/Schreibrechte).	Prot. 8, 16
41	GiveReadAccessToFile()	Erteilt einem *User* Leserecht für eine Datei durch Modifikation der $PERF$-Datei.	Prot. 8
42	AddStuToClass()	Fügt einen *Student* zur `students`-Datei einer Klasse hinzu und aktualisiert die Berechtigungen.	Prot. 9
43	AddTeaToClass()	Fügt einen *Teacher* zur `teachers`-Datei einer Klasse hinzu und aktualisiert die Berechtigungen.	Prot. 8
44	GetStudentsTeachers()	Ruft die Liste der *Teacher* ab, die einem spezifischen *Student* zugeordnet sind.	Prot. 17
45	GetTeachersStudents()	Ruft die Liste aller *Students* ab, die von einem spezifischen *Teacher* in dessen Klassen unterrichtet werden.	-

Tabelle 5.25.: Auflistung der Algorithmen mit Seiteneffekten, ihrer spezifischen Beschreibung und den Protokollen, in denen sie Anwendung finden.

Initiales Speichern der globalen öffentlichen Schlüssel (nur Admin)

StoreInitPUKs() wird nur vom *Admin* bei der initialen Systemeinrichtung verwendet, um die erste Version der $PUKS$-Datei zu speichern.

Algorithm 14: StoreInitPUKs($\langle PUKS \rangle$)

Input : Signed public keys file
$\langle PUKS \rangle = ((uid, PUK, VK)^n, sig) \in (\mathcal{U} \times \mathcal{K}_{PUK} \times \mathcal{K}_{VK})^n \times \mathcal{S}$

Constraints: $n = 1$

$(PUKS, sig) \leftarrow \langle PUKS \rangle$
$valid \leftarrow true$

if store_file("$Public$", "$puks$", $PUKS$) = false **then**
  $valid \leftarrow false$

if store_file("$Public$", "$puks.sig$", sig) = false **then**
  $valid \leftarrow false$

return $valid$ // $valid \in \{\text{true}, \text{false}\}$

Initiales Speichern einer verschlüsselten Datei (nur Admin)

StoreInitEncFile() wird vom *Admin* verwendet, um eine verschlüsselte Datei initial im System anzulegen.

Algorithm 15: StoreInitEncFile($folderPath, fileName, \langle [d] \rangle$)

Input: Path to the folder where the file will be stored $folderPath \in \Sigma^*$
Filename $fileName \in \Sigma^\star$
Encrypted and signed data $\langle [d] \rangle = ((n, a, c, t), sig) \in (\mathcal{N} \times \mathcal{A} \times \mathcal{C} \times \mathcal{T}) \times \mathcal{S}$

$((n, a, c, t), sig) \leftarrow \langle [d] \rangle$
$valid \leftarrow true$

if $\neg$ store_file($folderPath,\ fileName +$ "$.enc$", (n, a, c, t)) **then**
  $valid \leftarrow false$

if $\neg$ store_file($folderPath,\ fileName +$ "$.enc.sig$", sig) **then**
  $valid \leftarrow false$

return $valid$ // $valid \in \{\text{true}, \text{false}\}$

Initiales Speichern einer signierten Datei (nur Admin)

StoreInitSignFile() wird vom *Admin* verwendet, um eine signierte, aber unverschlüsselte Datei initial anzulegen (z.B. *Admin* KS).

Algorithm 16: StoreInitSignFile($folderPath, fileName, \langle d \rangle$)

Input: Path to the folder where the file will be stored $folderPath \in \Sigma^*$
Filename $fileName \in \Sigma^\star$
Signed data $\langle d \rangle = (d, sig) \in \mathcal{D} \times \mathcal{S}$

$(d, sig) \leftarrow \langle d \rangle$
$valid \leftarrow true$

if $\neg$ store_file($folderPath,\ fileName,\ d$) **then**
  $valid \leftarrow false$

if $\neg$ store_file($folderPath,\ fileName +$ "$.sig$", sig) **then**
  $valid \leftarrow false$

return $valid$ // $valid \in \{\text{true}, \text{false}\}$

Speichern der Teacher OTPs (nur Admin)

StoreTeaOTPS() speichert die vom *Admin* signierte Liste der otps für die *Teacher*.

Algorithm 17: StoreTeaOTPS($\langle TOTPS \rangle$)

Input: Signed TOTPS file $\langle TOTPS \rangle = ((uid_t,\ cotp_t)^n,\ sig) \in (\mathcal{U} \times \mathcal{C})^n \times \mathcal{S}$

$(TOTPS,\ sig) \leftarrow \langle TOTPS \rangle$
$valid \leftarrow true$

if ¬store_file("/", "$totps$", $TOTPS$) **then**
  $valid \leftarrow false$

if ¬store_file("/", "$totps.sig$", sig) **then**
  $valid \leftarrow false$

return $valid$ // $valid \in \{\text{true}, \text{false}\}$

Laden der Teacher OTP-Datei (nur Admin)

LoadTOTPS() lädt die spezifizierte $TOTPS$-Datei und deren Signatur von der *Platform*.

Algorithm 18: LoadTOTPS()

$TOTPS \leftarrow$ load_file("/", "$totps$")
$sig \leftarrow$ load_file("/", "$totps.sig$")

return $(TOTPS, sig)$
// $(TOTPS, sig) = ((uid_t,\ \ cotp_t)^n,\ \ sig) \in (\mathcal{U} \times \mathcal{C})^n \times \mathcal{S}$

Speichern der *Caregiver* OTPs (nur Admin)

StoreCarOTPS() speichert die vom *Admin* signierte Liste der otps für die *Caregiver*.

Algorithm 19: StoreCarOTPS($\langle COTPS \rangle$)

Input: Signed COTPS file $\langle COTPS \rangle = ((uid_c,\ cotp_c)^n,\ sig) \in (\mathcal{U} \times \mathcal{C})^n \times \mathcal{S}$

$(COTPS,\ sig) \leftarrow \langle COTPS \rangle$
$valid \leftarrow true$

if ¬store_file("/", "$cotps$", $COTPS$) **then**
  $valid \leftarrow false$

if ¬store_file("/", "$cotps.sig$", sig) **then**
  $valid \leftarrow false$

return $valid$ // $valid \in \{\text{true}, \text{false}\}$

Laden der Caregiver OTP-Datei (nur Admin)

LoadCOTPS() lädt die spezifizierte $COTPS$-Datei und deren Signatur von der *Platform*.

Algorithm 20: LoadCOTPS()

$COTPS \leftarrow$ load_file("/", "$cotps$")
$sig \leftarrow$ load_file("/", "$cotps.sig$")

return $(COTPS, sig)$
// $(COTPS, sig) = ((uid_c,\ cotp_c)^n,\ sig) \in (\mathcal{U} \times \mathcal{C})^n \times \mathcal{S}$

Speichern der Root-Berechtigungsdatei (nur Admin)

StoreRootPERF() speichert die vom *Admin* signierte $PERF$-Datei für das Root-Verzeichnis der *Platform*.

Algorithm 21: StoreRootPERF($\langle PERF \rangle$)

Input: Signed permissions file
$\langle PERF \rangle = ((fileName,\ (uid,\ cfsk)^m,\ (uid)^k)^n,\ sig) \in$
$(\Sigma^\star \times (\mathcal{U} \times \mathcal{C})^m \times (\mathcal{U})^k)^n \times \mathcal{S}$

$(PERF,\ sig) \leftarrow \langle PERF \rangle$
$valid \leftarrow true$

if ¬store_file("/", "$permissions$", $PERF$) **then**
  $valid \leftarrow false$
if ¬store_file("/", "$permissions.sig$", sig) **then**
  $valid \leftarrow false$

return $valid$ // $valid \in \{\text{true}, \text{false}\}$

Speichern des Teacher Registrierungspakets

StoreRPTea() speichert das verschlüsselte Registrierungspaket (RPT) und den signierten Keystore (KS) eines *Teachers* im öffentlichen Registrierungsordner.

Algorithm 22: StoreRPTea($uid_t,\ cfsk_{RPT},\ [RPT],\ \langle KS_t \rangle$)

Input: *User Identifier* of the teacher $uid_t \in \mathcal{U}$
Encrypted $cfsk$ for the RPT $cfsk_{RPT} \in \mathcal{C}$
Encrypted RPT $[RPT] = (n, a, c, t) \in \mathcal{N} \times \mathcal{A} \times \mathcal{C} \times \mathcal{T}$
Signed *Keystore* of the teacher $\langle KS_t \rangle = (KS_t,\ sig) \in \mathcal{KS} \times \mathcal{S}$

$KRPT \leftarrow (cfsk_{RPT},\ [RPT])$
$(KS_t,\ sig) \leftarrow \langle KS_t \rangle$
$valid \leftarrow true$

if ¬store_file("$Public/Registrations$", "$rpt-$" + uid_t + "$.enc$", $KRPT$) **then**
  $valid \leftarrow false$

if ¬store_file("$Public/Registrations$", "$ks-$" + uid_t, KS_t) **then**
  $valid \leftarrow false$

if ¬store_file("$Public/Registrations$", "$ks-$" + uid_t + "$.sig$", sig) **then**
  $valid \leftarrow false$

return $valid$ // $valid \in \{\text{true}, \text{false}\}$

Laden des Teacher Registrierungspakets

LoadRPTea() dient zum Laden des verschlüsselten Registrierungspakets (RPT) und des zugehörigen signierten Keystores (KS) eines *Teachers* von der *Platform*. Wird keine spezifische uid_t angegeben, wird ein beliebiges vorhandenes RPT gesucht und geladen.

Algorithm 23: LoadRPTea(uid_t)

Input: Optional *User Identifier* of the teacher $uid_t \in \mathcal{U}$

if uid_t is not given **then**
  $fileName_{RPT} \leftarrow$ find_arbitrary_rpt_filename()
  $uid_t \leftarrow$ extract_substring_from_filename($fileName_{RPT}$, "$rpt-$", "$.enc$")

$KRPT \leftarrow$ load_file("$Public/Registrations$", "$rpt-$" + uid_t + "$.enc$")
$(cfsk_{RPT}, [RPT]) \leftarrow KRPT$

$KS_t \leftarrow$ load_file("$Public/Registrations$", "$ks-$" + uid_t)
$sig_{KS_t} \leftarrow$ load_file("$Public/Registrations$", "$ks-$" + uid_t + "$.sig$")
$\langle KS_t \rangle \leftarrow (KS_t, sig_{KS_t})$

return $(cfsk_{RPT}, [RPT], \langle KS_t \rangle)$
// $(cfsk_{RPT}, [RPT], \langle KS_t \rangle) \in \mathcal{C} \times (\mathcal{N} \times \mathcal{A} \times \mathcal{C} \times \mathcal{T}) \times (\mathcal{KS} \times \mathcal{S})$

Löschen des Teacher Registrierungspakets

DeleteRPTea() löscht das Registrierungspaket und die zugehörigen KS-Dateien eines *Teachers* von der *Platform*.

Algorithm 24: DeleteRPTea(uid_t)

Input: *User Identifier* of the teacher $uid_t \in \mathcal{U}$

$valid \leftarrow true$

if ¬ delete_file("$Public/Registrations$", "$rpt-$" + uid_t + "$.enc$") **then**
  $valid \leftarrow false$

if ¬ delete_file("$Public/Registrations$", "$ks-$" + uid_t) **then**
  $valid \leftarrow false$

if ¬ delete_file("$Public/Registrations$", "$ks-$" + uid_t + "$.sig$") **then**
  $valid \leftarrow false$

return $valid$ // $valid \in$ {true, false}

Speichern des Caregiver Registrierungspakets

StoreRPCar() speichert das verschlüsselte Registrierungspaket (RPC) und den signierten Keystore (KS) eines *Caregivers* im öffentlichen Registrierungsordner.

Algorithm 25: StoreRPCar(uid_c, $cfsk_{RPC}$, $[RPC]$, $\langle KS_c \rangle$)

Input: *User Identifier* of the caregiver $uid_c \in \mathcal{U}$
Encrypted $cfsk$ for the RPC $cfsk_{RPC} \in \mathcal{C}$
Encrypted RPC $[RPC] = (n, a, c, t) \in \mathcal{N} \times \mathcal{A} \times \mathcal{C} \times \mathcal{T}$
Signed *Keystore* of the caregiver $\langle KS_c \rangle = (KS_c, sig) \in \mathcal{KS} \times \mathcal{S}$

$KRPC \leftarrow (cfsk_{RPC}, [RPC])$
$(KS_c, sig) \leftarrow \langle KS_c \rangle$
$valid \leftarrow true$

if ¬store_file("$Public/Registrations$", "$rpc-$" + uid_c + "$.enc$", $KRPC$) **then**
  $valid \leftarrow false$

if ¬store_file("$Public/Registrations$", "$ks-$" + uid_c, KS_c) **then**
  $valid \leftarrow false$

if ¬store_file("$Public/Registrations$", "$ks-$" + uid_c + "$.sig$", sig) **then**
  $valid \leftarrow false$

return $valid$ // $valid \in$ {true, false}

Laden des Caregiver Registrierungspakets (optional mit spezifischer UID)

LoadRPCar() dient zum Laden des verschlüsselten Registrierungspakets (RPC) und des zugehörigen signierten Keystores (KS) eines *Caregivers* von der *Platform*. Wird keine spezifische uid_c angegeben, wird ein beliebiges vorhandenes RPC gesucht und geladen.

Algorithm 26: LoadRPCar(uid_c)

Input: Optional *User Identifier* of the caregiver $uid_c \in \mathcal{U}$

if uid_c is not given **then**
  $fileName_{RPC} \leftarrow$ find_arbitrary_rpc_filename()
  $uid_c \leftarrow$ extract_substring_from_filename($fileName_{RPC}$, "$rpc-$", "$.enc$")

$KRPC \leftarrow$ load_file("$Public/Registrations$", "$rpc-$" + uid_c + "$.enc$")

$(cfsk_{RPC}, [RPC]) \leftarrow KRPC$
$KS_c \leftarrow$ load_file("$Public/Registrations$", "$ks-$" + uid_c)
$sig_{KS_c} \leftarrow$ load_file("$Public/Registrations$", "$ks-$" + uid_c + "$.sig$")
$\langle KS_c \rangle \leftarrow (KS_c, sig_{KS_c})$

return $(cfsk_{RPC}, [RPC], \langle KS_c \rangle)$
// $(cfsk_{RPC}, [RPC], \langle KS_c \rangle) \in \mathcal{C} \times (\mathcal{N} \times \mathcal{A} \times \mathcal{C} \times \mathcal{T}) \times (\mathcal{KS} \times \mathcal{S})$

Löschen des Caregiver Registrierungspakets

DeleteRPCar() löscht das Registrierungspaket (RPC) und die zugehörigen KS-Dateien eines *Caregivers* von der *Platform*.

Algorithm 27: DeleteRPCar(uid_c)

Input: *User Identifier* of the caregiver $uid_c \in \mathcal{U}$

$valid \leftarrow true$

if ¬ delete_file("$Public/Registrations$", "$rpc-$" + uid_c + "$.enc$") **then**
  $valid \leftarrow false$

if ¬ delete_file("$Public/Registrations$", "$ks-$" + uid_c) **then**
  $valid \leftarrow false$

if ¬ delete_file("$Public/Registrations$", "$ks-$" + uid_c + "$.sig$") **then**
  $valid \leftarrow false$

return $valid$ // $valid \in$ {true, false}

Speichern der globalen öffentlichen Schlüssel

StorePUKS() dient dem *Admin* zum Speichern oder Aktualisieren der globalen Liste öffentlicher Schlüssel ($PUKS$) aller *User*.

Algorithm 28: StorePUKS($uid_a, VK_a, \langle PUKS \rangle$)

Input: *User Identifier* of the admin $uid_a \in \mathcal{U}$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$
Signed public key file
$\langle PUKS \rangle = ((uid, PUK, VK)^n, sig) \in (\mathcal{U} \times \mathcal{K}_{PUK} \times \mathcal{K}_{VK})^n \times \mathcal{S}$

$(PUKS, sig) \leftarrow \langle PUKS \rangle$
$VK \leftarrow$ GetVK(uid_a, VK_a) // see Alg. 50
if ¬verify(VK, $PUKS$, sig) **then**
  return $\perp$

$valid \leftarrow true$

if ¬ store_file("$Public$", "$puks$", $PUKS$) **then**
  $valid \leftarrow false$

if ¬ store_file("$Public$", "$puks.sig$", sig) **then**
  $valid \leftarrow false$

return $valid$ // $valid \in \{\text{true}, \text{false}\}$

Laden der globalen öffentlichen Schlüssel

LoadPUKS() lädt die Datei mit den globalen öffentlichen Schlüsseln ($PUKS$) und deren Signatur aus dem öffentlichen Verzeichnis.

Algorithm 29: LoadPUKS()

$PUKS \leftarrow$ load_file("$Public$", "$puks$")
$sig \leftarrow$ load_file("$Public$", "$puks.sig$")

return $(PUKS, sig)$
// $(PUKS, sig) = ((uid, PUK, VK)^n, sig) \in (\mathcal{U} \times \mathcal{K}_{PUK} \times \mathcal{K}_{VK})^n \times \mathcal{S}$

Abrufen der verifizierten globalen öffentlichen Schlüssel

GetVerifiedPUKS() lädt die $PUKS$-Datei und deren Signatur und prüft die Gültigkeit der Signatur mit dem *Admin*-VK.

Algorithm 30: GetVerifiedPUKS(VK_a)

Input: *Admin*'s verification key $VK_a \in \mathcal{K}_{VK}$

$PUKS \leftarrow$ load_file("$Public$", "$puks$")
$sig \leftarrow$ load_file("$Public$", "$puks.sig$")

if ¬verify($VK_a, PUKS, sig$) **then**
  return $\perp$

return $PUKS$ // $PUKS = (uid, PUK, VK)^n \in (\mathcal{U} \times \mathcal{K}_{PUK} \times \mathcal{K}_{VK})^n$

Speichern der Berechtigungsdatei

StorePermissions() verifiziert, dass die signierte Berechtigungsdatei ($PERF$) vom *Admin* stammt, und speichert bei Gültigkeit diese als auch deren Signatur.

Algorithm 31: StorePermissions($uid_a, folderPath, \langle PERF \rangle, VK_a$)

Input: *User Identifier* of the admin $uid_a \in \mathcal{U}$
Path of the folder where the file is lying $folderPath \in \Sigma^*$
Signed permissions file $\langle PERF \rangle == ((fileName,\ (uid,\ cfsk)^m,\ (uid)^k)^n,\ sig) \in (\Sigma^\star \times (\mathcal{U} \times \mathcal{C})^m \times (\mathcal{U})^k)^n \times \mathcal{S}$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$

$(PERF, sig) \leftarrow \langle PERF \rangle$
$VK \leftarrow$ GetVK(uid_a, VK_a) // see Alg. 50
if ¬verify($VK, PERF, sig$) **then**
  return $\perp$

$valid \leftarrow true$
if ¬ store_file($folderPath$, "$permissions$", $PERF$) **then**
  $valid \leftarrow false$

if ¬ store_file($folderPath$, "$permissions.sig$", sig) **then**
  $valid \leftarrow false$

return $valid$ // $valid \in$ {true, false}

Abrufen und Verifizieren der Berechtigungsdatei

GetVerifiedPermissions() lädt die $PERF$-Datei für einen gegebenen Ordner und verifiziert deren Signatur mit dem *Admin*-VK.

Algorithm 32: GetVerifiedPermissions($folderPath, VK_a$)

Input: Folder path used to locate the permissions file $folderPath \in \Sigma^*$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$

$PERF \leftarrow$ load_file($folderPath$, "$permissions$")
$sig \leftarrow$ load_file($folderPath$, "$permissions.sig$")

if ¬verify($VK_a, PERF, sig$) **then**
  return $\perp$

return $PERF$
// $PERF = (fileName, (uid, cfsk)^m, (uid)^k)^n \in (\Sigma^\star \times (\mathcal{U} \times \mathcal{C})^m \times (\mathcal{U})^k)^n$

Speichern einer verschlüsselten Datei

StoreEncFile() speichert eine verschlüsselte und signierte Datei im Verzeichnis des angegebenen *Users*.

Algorithm 33: StoreEncFile($uid, folderPath, fileName, \langle[d]\rangle, VK_a$)

Input: *User Identifier* of the user attempting to store the file $uid \in \mathcal{U}$
Path to the folder where the file will be stored $folderPath \in \Sigma^*$
Filename $fileName \in \Sigma^*$
Encrypted and signed data $\langle[d]\rangle = ((n, a, c, t), sig) \in (\mathcal{N} \times \mathcal{A} \times \mathcal{C} \times \mathcal{T}) \times \mathcal{S}$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$

if ¬ VerifyWriteAccess($uid, folderPath, fileName, \langle[d]\rangle, VK_a$) **then**
  return $\perp$
// see Alg. 11

$((n, a, c, t), sig) \leftarrow \langle[d]\rangle$
$valid \leftarrow true$
if ¬ store_file($folderPath$, $fileName +$ "$.enc$", (n, a, c, t)) **then**
  $valid \leftarrow false$
if ¬ store_file($folderPath$, $fileName +$ "$.sig$", sig) **then**
  $valid \leftarrow false$

return $valid$ // $valid \in \{\text{true}, \text{false}\}$

Laden einer verschlüsselten Datei

LoadEncFile() lädt eine verschlüsselte Datei und ihre zugehörige Signatur, nachdem der Lesezugriff überprüft wurde.

Algorithm 34: LoadEncFile($uid, folderPath, fileName, VK_a$)

Input: *User Identifier* of the user attempting to read the file $uid \in \mathcal{U}$
Path of the folder where the file is lying $folderPath \in \Sigma^*$
Filename to load $fileName \in \Sigma^*$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$

// see Alg. 12

if ¬VerifyReadAccess($uid, folderPath, fileName, VK_a$) **then**
  return $\perp$

$c \leftarrow$ load_file($folderPath,\ fileName +$ ".enc")
$sig \leftarrow$ load_file($folderPath,\ fileName +$ ".enc.sig")

return (c, sig) // $(c, sig) \in (\mathcal{N} \times \mathcal{A} \times \mathcal{C} \times \mathcal{T}) \times \mathcal{S}$

Speichern einer signierten Datei

StoreSignFile() speichert eine signierte, unverschlüsselte Datei und deren Signatur, nachdem Schreibrechte überprüft wurden.

Algorithm 35: StoreSignFile($uid, folderPath, fileName, \langle d \rangle, VK_a$)

Input: *User Identifier* of the user attempting to store the file $uid \in \mathcal{U}$
Path to the folder where the file will be stored $folderPath \in \Sigma^*$
Filename $fileName \in \Sigma^\star$
Signed data $\langle d \rangle = (d, sig) \in \mathcal{D} \times \mathcal{S}$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$

if ¬ VerifyWriteAccess($uid, folderPath, fileName, \langle d \rangle, VK_a$) **then**
  return $\perp$

// see Alg. 11

$(d, sig) \leftarrow \langle d \rangle$
$valid \leftarrow true$

if ¬ store_file($folderPath,\ fileName,\ d$) **then**
  $valid \leftarrow false$

if ¬ store_file($folderPath,\ fileName +$ ".sig", sig) **then**
  $valid \leftarrow false$

return $valid$ // $valid \in \{$true, false$\}$

Laden einer signierten Datei

LoadSignFile() lädt eine signierte, unverschlüsselte Datei und ihre Signatur, nachdem Leserechte überprüft wurden.

Algorithm 36: LoadSignFile($uid, folderPath, fileName, VK_a$)

Input: *User Identifier* of the user attempting to read the file $uid \in \mathcal{U}$
Path of the folder where the file is lying $folderPath \in \Sigma^*$
Filename to load $fileName \in \Sigma^*$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$

if ¬VerifyReadAccess($uid, folderPath, fileName, VK_a$) **then**
  return $\perp$

// see Alg. 12

$d \leftarrow$ load_file($folderPath,\ fileName$)
$sig \leftarrow$ load_file($folderPath,\ fileName +$ ".sig")

return (d, sig) // $(d, sig) \in \mathcal{D} \times \mathcal{S}$

Erstellen eines Benutzerverzeichnisses

CreateUserDir() legt ein neues Verzeichnis für einen *User* auf der *Platform* an.

Algorithm 37: CreateUserDir(uid)

Input: *User Identifier* of the user $uid \in \mathcal{U}$

$valid \leftarrow true$

if ¬ create_dir("$User/$" + uid) **then**
  $valid \leftarrow false$

return $valid$ // $valid \in \{$true, false$\}$

Speichern der Chat-Teilnehmer

StoreParties() verifiziert, dass die signierte *parties*-Datei vom *Admin* stammt, und speichert bei Gültigkeit sowohl die *parties*-Datei als auch deren Signatur im angegebenen Chat-Ordner.

Algorithm 38: StoreParties($uid_a, chatFolderPath, \langle parties \rangle, VK_a$)

Input: *User Identifier* of the admin $uid_a \in \mathcal{U}$
Folder path to chat $chatFolderPath \in \Sigma^*$
Signed data $\langle parties \rangle = (parties, sig) = (uid^n, uid^m) \in (\mathcal{U}^n \times \mathcal{U}^m) \times \mathcal{S}$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$

$(parties, sig) \leftarrow \langle parties \rangle$
$VK \leftarrow$ GetVK(uid_a, VK_a) `// see Alg. 50`
if ¬verify$(VK, parties, sig)$ **then**
  return $\perp$

$valid \leftarrow true$
if ¬ store_file$(chatFolderPath,$ "$parties$", $parties)$ **then**
  $valid \leftarrow false$

if ¬ store_file$(chatFolderPath,$ "$parties.sig$", $sig)$ **then**
  $valid \leftarrow false$

return $valid$ `//` $valid \in \{$true, false$\}$

Abrufen der verifizierten Chat-Teilnehmer

GetVerifiedParties() lädt und verifiziert die signierte *parties*-Datei aus einem spezifizierten Chat-Ordner unter Verwendung des Verifikationsschlüssels des *Admins*.

Algorithm 39: GetVerifiedParties($chatFolderPath, VK_a$)

Input: Folder path to chat $chatFolderPath \in \Sigma^*$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$

$parties \leftarrow$ load_file$(chatFolderPath,$ "$parties$")
$sig \leftarrow$ load_file$(chatFolderPath,$ "$parties.sig$")

if ¬verify$(VK_a, parties, sig)$ **then**
  return $\perp$

return $parties$ `//` $parties = ((uid)^n, (uid)^m) \in (\mathcal{U})^n \times (\mathcal{U})^m$

Hinzufügen einer UID zu einem Chat

AddUIDToChat() fügt eine uid zur *parties*-Datei eines Chat-Ordners hinzu, entweder zur Lese-, Schreib- oder beiden Listen.

Algorithm 40: AddUIDToChat($chatFolderPath, uid_a, uid, VK_a, sik_a, read, write$)

Input: Folder path to chat $chatFolderPath \in \Sigma^*$
User Identifier of the admin $uid_a \in \mathcal{U}$
User Identifier to be added to the parties file $uid \in \mathcal{U}$ Admin's verification key $VK_a \in \mathcal{K}_{VK}$
Admin's signature key $sik_a \in \mathcal{K}_{sik}$
Flag to add the UID to the read list $flag_r \in \{\text{true}, \text{false}\}$
Flag to add the UID to the write list $flag_w \in \{\text{true}, \text{false}\}$

$parties \leftarrow$ GetVerifiedParties($chatFolderPath, VK_a$) `// see Alg. 39`

$(read, write) \leftarrow parties$
if $flag_r \wedge uid \notin read$ **then**
  $read \leftarrow read \cup (uid)$
  $parties \leftarrow parties \cup (read, \cdot)$
if $flag_w \wedge uid \notin parties.$write **then**
  $write \leftarrow write \cup (uid)$
  $parties \leftarrow parties \cup (\cdot, write)$

$sig \leftarrow$ sign($sik_a, parties$)
$\langle parties \rangle \leftarrow (parties, sig)$

if $\neg$StoreParties($uid_a, chatFolderPath, \langle parties \rangle, VK_a$) **then**
  return

`// see Alg. 38`

Erteilen von Leserecht für eine Datei

GiveReadAccessToFile() erteilt einem neuen *User* Leserecht für eine spezifische Datei, indem dessen uid und der verschlüsselte Datei-Schlüssel (fsk) zur Berechtigungsdatei ($PERF$) hinzugefügt werden.

Algorithm 41: GiveReadAccessToFile($uid_a, uid, folderPath, fileName, VK_a, sik_a, prk_a$)

Input: *User Identifier* of the admin $uid_a \in \mathcal{U}$
User Identifier of the User $uid \in \mathcal{U}$
Path of the folder where the file is lying $folderPath \in \Sigma^*$
Filename $fileName \in \Sigma^*$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$
Admin's signature key $sik_a \in \mathcal{K}_{sik}$
Admin's private key $prk_a \in \mathcal{K}_{prk}$

$PERF \leftarrow$ GetVerifiedPermissions$(folderPath, VK_a)$

$cfsk_{admin} \leftarrow$ GetEncFileKey$(uid_a, folderPath, fileName, VK_a)$ // see Alg. 13
$fsk \leftarrow$ dec_a$(prk_a, cfsk_{admin})$

$PUK \leftarrow$ GetPUK(uid, VK_a) // see Alg. 49
$cfsk \leftarrow$ enc_a(PUK, fsk)
$PERF \leftarrow PERF \cup (\cdot, (uid, cfsk), \cdot)$

$sig \leftarrow$ sign$(sik_a, PERF)$
$\langle PERF \rangle \leftarrow (PERF, sig)$

if ¬StorePermissions$(uid_a, folderPath, \langle PERF \rangle, VK_a)$ **then**
  return $\perp$
// see Alg. 31

Hinzufügen eines Studenten zu einer Klasse

AddStuToClass() fügt die uid eines *Students* zur *students*-Datei einer gegebenen Klasse hinzu.

Algorithm 42: AddStuToClass($uid_a, prk_a, sik_a, VK_a, classId, uid_s$)

Input: *User Identifier* of the admin $uid_a \in \mathcal{U}$
Admin's private key $prk_a \in \mathcal{K}_{prk}$ Admin's signature key $sik_a \in \mathcal{K}_{sik}$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$
ID of the class $classId \in \Sigma^*$
User Identifier of the student $uid_s \in \mathcal{U}$

$classPath \leftarrow$ "$Classes/$" $+ classId$
$fileName \leftarrow$ "$students$"

$PERF \leftarrow$ GetVerifiedPermissions$(classPath, VK_a)$ // see Alg. 32
$cfsk \leftarrow$ GetEncFileKey$(uid_a, folderPath, fileName, VK_a)$ // see Alg. 13
$fsk \leftarrow$ dec_a$(prk_a, cfsk)$

$([ST], sig) \leftarrow$ LoadEncFile$(uid_a, classPath, fileName, VK_a)$ // see Alg. 34
$(ad_{ST}, ST) \leftarrow$ DecVerify$([ST], sig, VK_a, fsk)$ // see Alg. 10

if $uid_s \in ST$ **then**
  return
$ST \leftarrow ST \cup (uid_s)$

$\langle [ST] \rangle \leftarrow$ EncSignWithKey$(ST, ad_{ST}, fsk, sik_a)$ // see Alg. 9

if $\neg$StoreEncFile$(uid_a, classPath, fileName, \langle [ST] \rangle, VK_a)$ **then**
  return $\perp$

Hinzufügen eines Teachers zu einer Klasse

AddTeaToClass() fügt die uid eines *Teachers* zur *teachers*-Datei einer gegebenen Klasse hinzu.

Algorithm 43: AddTeaToClass($uid_a, prk_a, sik_a, VK_a, classId, uid_t$)

Input: *User Identifier* of the admin $uid_a \in \mathcal{U}$
Admin's private key $prk_a \in \mathcal{K}_{prk}$ Admin's signature key $sik_a \in \mathcal{K}_{sik}$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$
ID of the class $classId \in \Sigma^*$
User Identifier of the teacher $uid_t \in \mathcal{U}$

$classPath \leftarrow$ "$Classes/$" $+ classId$
$fileName \leftarrow$ "$teachers$"

$PERF \leftarrow$ GetVerifiedPermissions$(classPath, VK_a)$ // see Alg. 32
$cfsk \leftarrow$ GetEncFileKey$(uid_a, folderPath, fileName, VK_a)$ // see Alg. 13
$fsk \leftarrow$ dec_a$(prk_a, cfsk)$

$([TA], sig) \leftarrow$ LoadEncFile$(uid_a, classPath, fileName, VK_a)$ // see Alg. 34
$(ad_{TA}, TA) \leftarrow$ DecVerify$([TA], sig, VK_a, fsk)$ // see Alg. 10

if $uid_t \in TA$ **then**
  return
$TA \leftarrow TA \cup (uid_t)$

$\langle [TA] \rangle \leftarrow$ EncSignWithKey$(TA, ad_{TA}, fsk, sik_a)$ // see Alg. 9

if $\neg$StoreEncFile$(uid_a, classPath, fileName, \langle [TA] \rangle, VK_a)$ **then**
  return $\perp$

Abrufen der Teacher eines Studenten

GetStudentsTeachers() ruft die Liste der *Teacher* ab, die einem bestimmten *Student* zugeordnet sind.

Algorithm 44: GetStudentsTeachers(uid, prk, uid_s, VK_a)

Input: *User Identifier* of the requesting user $uid \in \mathcal{U}$
Private key of the requesting user $prk \in \mathcal{K}_{prk}$
User Identifier of the student $uid_s \in \mathcal{U}$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$

$folderPath \leftarrow$ "$User/$" $+ uid_s$
$fileName \leftarrow$ "$teachers$"

$PERF \leftarrow$ GetVerifiedPermissions($folderPath, VK_a$) `// see Alg. 32`
$cfsk \leftarrow$ GetEncFileKey($uid, folderPath, fileName, VK_a$) `// see Alg. 13`
$fsk \leftarrow$ dec_a($prk, cfsk$)

$([TA], sig) \leftarrow$ LoadEncFile($uid, folderPath, fileName, VK_a$) `// see Alg. 34`
$(ad_{TA}, TA) \leftarrow$ DecVerify($[TA], sig, VK_a, fsk$) `// see Alg. 10`

return TA // $TA = (uid_t)^n = (\mathcal{U})^n$

Abrufen der Studenten eines Teachers für dessen Klassen

GetTeachersStudents() ruft die Liste aller *Students* ab, die den Klassen zugeordnet sind, welche von einem bestimmten *Teacher* unterrichtet werden.

Algorithm 45: GetTeachersStudents($uid_t, prk_t, classes, VK_a$)

Input: *User Identifier* of the teacher $uid_t \in \mathcal{U}$
Teacher's private key $prk_t \in \mathcal{K}_{prk}$
A list of class IDs $classes$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$

$allStudents \leftarrow ()$
$fileName \leftarrow tudents$"

foreach $id \in classes$ **do**
  $classPath \leftarrow$ "$Classes/$" $+ id$
  $PERF \leftarrow$ GetVerifiedPermissions$(classPath, VK_a)$ // see Alg. 32
  $cfsk \leftarrow$ GetEncFileKey$(uid_t, folderPath, fileName, VK_a)$ // see Alg. 13
  $fsk \leftarrow$ dec_a$(prk_t, cfsk)$
  $([ST], sig) \leftarrow$ LoadEncFile$(uid_t, classPath, fileName, VK_a)$ // see Alg. 34
  $(ad_{ST}, ST) \leftarrow$ DecVerify$([ST], sig, VK_a, fsk)$ // see Alg. 10
  $allStudents \leftarrow allStudents \cup ST$

return $allStudents$ // $allStudents = (uid_s)^n = (\mathcal{U})^n$

5.3.3. Schlüssel spezifische Algorithmen

Die Sicherheit der gesamten Schulkommunikationslösung beruht massgeblich auf der korrekten Generierung, Ableitung und Handhabung kryptographischer Schlüssel. Dieser Abschnitt widmet sich den Algorithmen, die spezifisch für diese Aufgaben konzipiert wurden. Dazu gehören die Ableitung von Benutzerschlüsseln (usk) aus Benutzergeheimnissen (us), die Generierung asymmetrischer Schlüsselpaare für Verschlüsselung und Signatur sowie die sichere Verschlüsselung dieser privaten Schlüsselkomponenten. Ebenfalls werden Verfahren zum Abrufen öffentlicher Schlüssel (PUK) und Verifikationsschlüssel (VK) aus der globalen $PUKS$-Datei spezifiziert. Eine detaillierte Übersicht dieser schlüsselspezifischen Algorithmen und ihrer Verwendung in den Protokollen ist in Tabelle 5.26 dargestellt.

Nr.	Algorithmus	Beschreibung	Prot./Alg.
46	DeriveUserSecretKey()	Leitet mittels einer *KDF*, die in Abschnitt 5.1.7 beschrieben ist, einen usk aus dem us ab. Sie gibt einerseits den usk, sowie den für die *KDF* verwendeten *Salt* zurück.	Prot. 1, 4, 11
47	GenKeyPairs()	Erzeugt eine Menge asymmetrischer Schlüsselpaare zur Verschlüsselung und Signatur, bestehend aus PUK, prk, VK, sik.	Prot. 1, 4, 11
48	EncPrivateKeys()	Verschlüsselt die privaten Schlüssel (glsprk und sik) mit einem sk des *Users*, dies kann der usk aber auch ein *Device Secret Key* (dsk) sein. Zurückgegeben wird ein Tupel bestehend aus zwei *AEAD*-Tupels.	Prot. 1, 4, 11
49	GetPUK()	Ruft den öffentlichen Schlüssel (PUK) eines spezifischen uid aus der verifizierten $PUKS$-Datei ab.	Alg. 58, 59
50	GetVK()	Ruft den Verifikationsschlüssel (VK) eines spezifischen uid aus der verifizierten $PUKS$-Datei ab.	Alg. 11, 28, 31, 38, 60

Tabelle 5.26.: Spezifikation der schlüsselspezifischen Algorithmen, inklusive ihrer Beschreibung und der Protokollschritte, in denen sie eingesetzt werden.

Ableitung des Benutzerschlüssels

DeriveUserSecretKey() generiert aus einem us und einem (optionalen) *Salt* einen symmetrischen Benutzerschlüssel (usk).

Algorithm 46: DeriveUserSecretKey(us, salt)

Input : User secret $us \in \Sigma^*$
Optional *Salt* $salt \in \mathcal{R}$

Constraints: $|us| > 0$

if salt is not given **then**
  $salt \leftarrow$ GenSalt() `// see Alg. 5`

$usk \leftarrow$ kdf$(us,\ salt,\ "UserSecretKey")$

return $(usk,\ salt)$ `//` $(usk,\ salt) \in \mathcal{K} \times \mathcal{R}$

Generierung von Schlüsselpaaren

GenKeyPairs() erzeugt sowohl ein asymmetrisches Schlüsselpaar für Verschlüsselungszwecke als auch eines für digitale Signaturen.

Algorithm 47: GenKeyPairs()

$(PUK, prk) \leftarrow$ gen_key_pair_enc()
$(VK, sik) \leftarrow$ gen_key_pair_sig()

return (PUK, prk, VK, sik)
`//` $(PUK, prk, VK, sik) \in \mathcal{K}_{PUK} \times \mathcal{K}_{prk} \times \mathcal{K}_{VK} \times \mathcal{K}_{sik}$

Verschlüsselung privater Schlüssel

EncPrivateKeys() verschlüsselt den sik und prk eines *Users* mit dessen usk.

Algorithm 48: EncPrivateKeys($uid, skid, usk, prk, sik$)

Input: *User Identifier* $uid \in \mathcal{U}$
Secret key identifier $skid \in \Sigma^\star$
User secret key $usk \in \mathcal{K}$
User private key $prk \in \mathcal{K}_{prk}$
User signature key $sik \in \mathcal{K}_{sik}$

$nonce_{prk} \leftarrow$ GenNonce() `// see Alg. 4`

$(ct_{prk}, tag_{prk}) \leftarrow$ enc_s$(usk, nonce_{prk}, ad_{prk}, prk)$

$[prk] \leftarrow (nonce_{prk}, ad_{prk}, ct_{prk}, tag_{prk})$

$nonce_{sik} \leftarrow$ GenNonce()

$(ct_{sik}, tag_{sik}) \leftarrow$ enc_s$(usk, nonce_{sik}, ad_{sik}, sik)$

$[sik] \leftarrow (nonce_{sik}, ad_{sik}, ct_{sik}, tag_{sik})$

return $([prk],\ [sik])$ `//` $([prk],\ [sik]) \in (\mathcal{N} \times \mathcal{A} \times \mathcal{C} \times \mathcal{T})^2$

Abrufen eines öffentlichen Schlüssels (PUK)

GetPUK() holt den PUK eines *User* aus der globalen, verifizierten $PUKS$-Datei.

Algorithm 49: GetPUK(uid, VK_a)

Input: *User Identifier* whose public key should be retrieved $uid \in \mathcal{U}$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$

$PUKS \leftarrow$ GetVerifiedPUKS(VK_a) `// see Alg. 30`

foreach $t \in PUKS$ **do**
  $(t\_uid, t\_PUK, \cdot) \leftarrow t$
  if $t\_uid = uid$ **then**
    $PUK \leftarrow t\_PUK$

if $\neg\ PUK$ **then**
  return $\perp$

return PUK `//` $PUK \in \mathcal{K}_{PUK}$

Abrufen eines Verifikationsschlüssels (VK)

GetVK() holt den VK eines *User* aus der globalen, verifizierten $PUKS$-Datei.

Algorithm 50: GetVK(uid, VK_a)

Input: *User Identifier* whose verification key should be retrieved $uid \in \mathcal{U}$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$

$PUKS \leftarrow$ GetVerifiedPUKS(VK_a) `// see Alg. 30`

foreach $t \in PUKS$ **do**
  $(t\_uid, \cdot, t\_VK) \leftarrow t$
  if $t\_uid = uid$ **then**
    $VK \leftarrow t\_VK$

if $\neg\, VK$ **then**
  return $\perp$

return VK `//` $VK \in \mathcal{K}_{VK}$

5.3.4. Admin Algorithmen

Der *Admin* spielt eine zentrale Rolle bei der Initialisierung und Verwaltung der Schulkommunikationslösung und besitzt erweiterte Berechtigungen. Dieser Abschnitt spezifiziert die Algorithmen, die exklusiv vom oder für den *Admin* ausgeführt werden, um dessen spezifische Aufgaben zu unterstützen. Diese umfassen das Management von Einmalpasswörtern (otps) für die Registrierung anderer *Users* (wie *Caregivers* oder *Teachers*), das Abrufen von otps und die Generierung von Leseberechtigungslisten für Dateien. Tabelle 5.27 fasst diese administrativen Algorithmen zusammen und verweist auf ihre jeweilige Anwendung im System.

Nr.	Algorithmus	Beschreibung	Prot./Alg.
51	GetAdmOTP()	Wird von der *Platform* ausgeführt und und holt die $HOTPS$.	Alg. 52
52	VerifyAdmOTP()	Wird von der *Platform* ausgeführt und prüft das vom *Admin* bei seiner Registrierung angegebene otp.	Prot. 1
53	GetTeaOTP()	Holt das passende otp einer uid_t aus der $TOTPS$.	Alg. 54
54	VerifyTeaOTP()	Wird vom $client_a$ ausgeführt, um das otp des *Teacher* zu prüfen.	Prot. 5

Nr.	Algorithmus	Beschreibung	Prot./Alg.
55	GenCOTPs()	Generiert n otps für *Caregivers* eines bestimmten *Student*. Die otps werden mit dem öffentlichen Schlüssel (PUK) des *Admin* verschlüsselt und als Tupel gespeichert. Dieses Tupel, welches Teil von der Datenstruktur 5.1.8 ist, wird zurückgegeben.	Prot. 10
56	GetCarOTP()	Liest ein spezifisches otp eines *Caregiver* von einem *Student* aus der Datei $COTPS$ und entschlüsselt es mit dem privaten Schlüssel des Admins.	Alg. 57
57	VerifyCarOTP()	Vergleicht ein erhaltenes otp mit dem in $COTPS$ gespeicherten Original, um die Gültigkeit der Registrierung und die Authentifizierung des *Caregiver* sicherzustellen.	Prot. 12
58	GenReadList()	Holt die PUKs der uids der *User* welche Leserechte auf eine Datei haben sollen und verschlüsselt damit den fsk. Als Rückgabewert gibt es eine $read$-Liste für die $PERF$ Datei.	Prot. 9

Tabelle 5.27.: Zusammenstellung der administrativen Algorithmen mit ihrer jeweiligen Beschreibung und Angabe der Protokolle oder Algorithmen, in denen sie verwendet werden.

Abrufen der Admin OTP Hashes

GetAdmOTP() ist ein interner *Platform*-Algorithmus zum Laden der Datei, die die Hashwerte der otps für *Admins* enthält.

Algorithm 51: GetAdmOTP()

$HOTP \leftarrow$ load_file("$hotp$")

return $HOTP$ // $HOTP = (uid_a, hash_{otp})^n \in (\mathcal{U} \times \mathcal{H}_{out})^n$

Verifizieren des Admin OTPs

VerifyAdmOTP() wird von der *Platform* verwendet, um das vom *Admin* bei der Registrierung angegebene otp zu validieren.

Algorithm 52: VerifyAdmOTP(uid_a, otp_a)

Input: Admin's user identifier $uid_a \in \mathcal{U}$
Admin's one-time password $otp_a \in \mathcal{O}$

$HOTP \leftarrow$ GetAdmOTP() `// see Alg. 51`
$valid \leftarrow$ false

foreach $h \in HOTP$ **do**
  $(uid, hash_{otp}) \leftarrow h$
  if $uid = uid_a$ **then**
    $valid \leftarrow (\text{hash}(otp_a) = hash_{otp})$

return $valid$ `//` $valid \in \{\text{true}, \text{false}\}$

Abrufen des Teacher OTPs

GetTeaOTP() extrahiert und entschlüsselt das otp für einen bestimmten *Teacher* aus der $TOTPS$-Datei.

Algorithm 53: GetTeaOTP($prk_a, uid_t, TOTPS$)

Input: Admin's private key $prk_a \in \mathcal{K}_{prk}$
User Identifier of the teacher $uid_t \in \mathcal{U}$
TOTPS file $TOTPS = (uid_t,\ cotp_t)^n \in (\mathcal{U} \times \mathcal{C})^n$

foreach $o \in TOTPS$ **do**
  $(uid, cotp) \leftarrow o$
  if $uid = uid_t$ **then**
    $otp \leftarrow \text{dec\_a}(prk_a, cotp)$

return otp `//` $otp \in \mathcal{O}$

Verifizieren des Teacher OTPs

VerifyTeaOTP() wird vom Admin-Client ($client_a$) verwendet, um das vom *Teacher* übermittelte otp zu validieren.

Algorithm 54: VerifyTeaOTP($prk_a, uid_t, otp_t, TOTPS$)

Input: Admin's private key $prk_a \in \mathcal{K}_{prk}$
User Identifier of the teacher $uid_t \in \mathcal{U}$
Presented otp $otp_t \in \mathcal{O}$
TOTPS file $TOTPS = (uid_t,\ cotp_t)^n \in (\mathcal{U} \times \mathcal{C})^n$

$otp_{ref} \leftarrow$ GetTeaOTP$(prk_a, uid_t, TOTPS)$ `// see Alg. 53`
$valid \leftarrow otp_{ref} = otp_t$

return $valid$ `//` $valid \in \{\text{true}, \text{false}\}$

Generierung einer OTP-Liste für Caregivers

GenCOTPs() erzeugt für einen *Student* eine Liste von otps, die jeweils für einen *Caregiver* bestimmt und mit dem Admin-PUK verschlüsselt sind.

Algorithm 55: GenCOTPs($uid_s, (uid_c)^n, PUK_a$)

Input: *User Identifier* of the student $uid_s \in \mathcal{U}$
List of Caregiver UIDs $(uid_c)^n \in (\mathcal{U})^n$
Admin's public key $PUK_a \in \mathcal{K}_{PUK}$

$caregivers \leftarrow ()$
foreach $uid_c \in (uid_c)^n$ **do**
  $otp \leftarrow$ GenOTP() `// see Alg. 3`
  $cotp \leftarrow$ enc_a(PUK_a, otp)
  $caregivers \leftarrow caregivers \cup (uid_c, cotp)$

$COTPS \leftarrow (uid_s, caregivers)$

return $COTPS$ `//` $COTPS = (uid_s, (uid_c, cotp_c)^m)^n \in (\mathcal{U} \times (\mathcal{U} \times \mathcal{C})^m)^n$

Abrufen des Caregiver OTPs

GetCarOTP() extrahiert und entschlüsselt das otp für einen bestimmten *Caregiver* und *Student* aus der $COTPS$-Datei.

Algorithm 56: GetCarOTP($prk_a, uid_s, uid_c, COTPS$)

Input: Admin's private key $prk_a \in \mathcal{K}_{prk}$
User Identifier of the student $uid_s \in \mathcal{U}$
User Identifier of the caregiver $uid_c \in \mathcal{U}$
COTPS file $COTPS = (uid_s, (uid_c, cotp_c)^m)^n \in (\mathcal{U} \times (\mathcal{U} \times \mathcal{C})^m)^n$

foreach $o \in COTPS$ **do**
  $(o\_uid_s, caregivers) \leftarrow o$
  if $o\_uid_s = uid_s$ **then**
    foreach $c \in caregivers$ **do**
      $(c\_uid_c, cotp) \leftarrow c$
      if $c\_uid_c = uid_c$ **then**
        $otp \leftarrow \mathsf{dec\_a}(prk_a, cotp)$

return otp // $otp \in \mathcal{O}$

Verifizieren des Caregiver OTPs

VerifyCarOTP() wird vom Admin-Client ($client_a$) verwendet, um das vom *Caregiver* übermittelte otp zu validieren.

Algorithm 57: VerifyCarOTP($prk_a, uid_s, uid_c, otp_c, COTPS$)

Input: Admin's private key $prk_a \in \mathcal{K}_{prk}$
User Identifier of the student $uid_s \in \mathcal{U}$
User Identifier of the caregiver $uid_c \in \mathcal{U}$
Presented OTP $otp_c \in \mathcal{O}$
COTPS file $COTPS = (uid_s, (uid_c, cotp_c)^m)^n \in (\mathcal{U} \times (\mathcal{U} \times \mathcal{C})^m)^n$

$otp_{ref} \leftarrow \mathsf{GetCarOTP}(prk_a, uid_s, uid_c, COTPS)$ // see Alg. 56
$valid \leftarrow otp_{ref} = otp_c$

return $valid$ // $valid \in \{\mathsf{true}, \mathsf{false}\}$

Erstellung einer Leseberechtigungsliste

GenReadList() nimmt eine Liste von uids und einen Dateischlüssel fsk entgegen. Er verschlüsselt den Dateischlüssel fsk für jeden *User* (inklusive *Admin* selbst) mit dessen öffentlichem Schlüssel PUK.

Algorithm 58: GenReadList($uid_a,\ PUK_a,\ VK_a,\ (uid)^n,\ fsk$)

Input: Admin's *User Identifier* $uid_a \in \mathcal{U}$
Admin's public key $PUK_a \in \mathcal{K}_{PUK}$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$
List of uids of readers length n $(uid)^n \in (\mathcal{U})^n$
File secret key $fsk \in \mathcal{K}$

$cfsk_a \leftarrow$ enc_a$(PUK_a,\ fsk)$
$read \leftarrow (uid_a, cfsk_a)$

foreach $u \in (uid)^n$ **do**
  $PUK \leftarrow$ GetPUK$(u,\ VK_a)$ `// see Alg. 49`
  $cfsk \leftarrow$ enc_a$(PUK,\ fsk)$
  $read \leftarrow read \cup (u,\ cfsk)$

return $read$ // $read = (uid,\ \ cfsk)^{n+1} \in (\mathcal{U} \times \mathcal{C})^{n+1}$

5.3.5. Nachrichten Algorithmen

Der Austausch von Nachrichten stellt eine Kernfunktionalität der entwickelten Schulkommunikationslösung dar. Die sichere und korrekte Handhabung dieses Prozesses ist entscheidend für die *Confidentiality*, *Integrity* und *Authenticity* der übermittelten Informationen. Dieser Abschnitt detailliert die spezifischen Algorithmen, die für das Senden und Empfangen von Nachrichten zuständig sind. SendMessage() (59) kapselt den gesamten Prozess des Erstellens, Signierens, Verschlüsselns und persistenten Speicherns einer Nachricht, während ReceiveMessage() (60) das Laden, Entschlüsseln und Verifizieren einer empfangenen Nachricht beschreibt. Tabelle 5.28 listet diese Algorithmen auf und verweist auf die entsprechenden Protokollschritte.

Nr.	Algorithmus	Beschreibung	Protokoll
59	SendMessage()	Erstellt, signiert, verschlüsselt und speichert eine Nachricht (MF) auf der *Platform*.	Prot. 17
60	ReceiveMessage()	Lädt, entschlüsselt und verifiziert eine Nachricht (MF) von der *Platform*.	Prot. 18

Tabelle 5.28.: Definition der Algorithmen für den Nachrichtenaustausch, inklusive ihrer Beschreibung und der Protokollkontexte, in denen sie zum Einsatz kommen.

Senden einer Nachricht

SendMessage() erstellt eine neue Nachricht (MF), einschliesslich Header, Signaturinformationen und verschlüsseltem Inhalt. Die Nachricht wird mit einem temporären msk verschlüsselt, welcher wiederum für jeden Empfänger (inklusive Sender) mit dessen PUK verschlüsselt wird. Die gesamte MF wird anschliessend auf der *Platform* gespeichert.

Algorithm 59: SendMessage($uid_s, uid_{sender}, sik, (uid)^n, msg, VK_a$)

Input: *User Identifier* of the student $uid_s \in \mathcal{U}$
Sender *User Identifier* $uid_{sender} \in \mathcal{U}$
Sender signing key $sik \in \mathcal{K}_{sik}$
List of recipient *uids* $(uid)^n \in (\mathcal{U})^n$
Plaintext message $msg \in \mathcal{D}$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$

$mid \leftarrow$ GenMID() // see Alg. 6
$timeStamp \leftarrow$ gen_timestamp
$header \leftarrow (uid_{sender}, (uid)^n, mid, timeStamp)$

$data \leftarrow (header, msg)$
$sig \leftarrow$ sign$(sik, data)$
$sigInfo \leftarrow (sig, \text{"}HashAlgorithm\text{"})$

$msk \leftarrow$ gen_key()
$nonce_{msg} \leftarrow$ GenNonce() // see Alg. 4
$(ct_{msg}, tag_{msg}) \leftarrow$ enc_s$(msk, nonce_{msg}, ad_{msg}, msg)$
$[msg] \leftarrow (nonce_{msg}, ad_{msg}, ct_{msg}, tag_{msg})$

$rcpInfo \leftarrow ()$
$uids \leftarrow (uid)^n \cup uid_{sender}$
foreach $uid_p \in uids$ **do**
  $PUK \leftarrow$ GetPUK(uid_p, VK_a) // see Alg. 49
  $cmsk \leftarrow$ enc_a(PUK, msk)
  $rcpInfo \leftarrow rcpInfo \cup (uid_p, cmsk)$

$MF \leftarrow (header, sigInfo, rcpInfo, [msg])$
$fileName \leftarrow timeStamp + \text{"}-\text{"} + mid$
$target\_path \leftarrow$ get_chat_folder$(uid_s, header)$

$valid \leftarrow true$
if $\neg$ store_file$(target_path, fileName, MF)$ **then**
  $valid \leftarrow false$

return $valid$ // $valid \in \{$true, false$\}$

Empfangen einer Nachricht

ReceiveMessage() lädt die nächste ungelesene Nachricht (MF) für den *User*, entschlüsselt den msk mit dem privaten Schlüssel (prk) des Empfängers, entschlüsselt damit den Nachrichteninhalt und verifiziert dessen Signatur.

Algorithm 60: ReceiveMessage($uid, prk, VK_a, folderPath, lastMsgTime$)

Input: Recipient *User Identifier* $uid \in \mathcal{U}$
Recipient private key $prk \in \mathcal{K}_{prk}$
Admin's verification key $VK_a \in \mathcal{K}_{VK}$
Path of the folder where the messages are lying $folderPath \in \Sigma^*$
Timestamp of the last message received by the user $lastMsgTimein\mathcal{N},$

$MF \leftarrow$ load_next_message$(folderPath, lastMsgTime)$
$(header, sigInfo, rcpInfo, [msg]) \leftarrow MF$

foreach $(uid_p, cmsk_p) \in rcpInfo$ **do**
  if $uid = uid_p$ **then**
    $cmsk \leftarrow cmsk_p$

if $\neg\ cmsk$ **then**
  return $\perp$
$msk \leftarrow$ dec_a$(prk, cmsk)$

$(uid_{sender}, \cdot, \cdot, \cdot) \leftarrow header$
$(nonce_{msg}, ad_{msg}, ct_{msg}, tag_{msg}) \leftarrow [msg]$
$msg \leftarrow$ dec_s$(msk, nonce_{msg}, ad_{msg}, ct_{msg}, tag_{msg})$

$VK_{sender} \leftarrow$ GetVK(uid_{sender}, VK_a) `// see Alg. 50`
$data \leftarrow (header, msg)$
$(sig, \cdot) \leftarrow sigInfo$
if $\neg$ verify$(VK_{sender}, data, sig)$ **then**
  return $\perp$

return msg // $msg \in \mathcal{D}$

6. Proof of Concept

Ziel des *PoC* ist es, zentrale Konzepte der spezifizierten Schulkommunikationslösung (siehe Kapitel 5) prototypisch umzusetzen und damit deren praktische Realisierbarkeit zu überprüfen. Der Fokus liegt auf der korrekten Anwendung kryptographischer Mechanismen und der Abbildung wesentlicher Protokollschritte in einer modularen Codebasis.

Der *PoC* dient als praktischer Machbarkeitsnachweis und demonstriert exemplarisch, dass die entworfenen Konzepte mit etablierten Programmiersprachen und kryptographischen Bibliotheken realisierbar sind.

Die zentralen Ziele dieses *PoC* umfassen:

- **Demonstration eines funktionalen Durchstichs:** Exemplarische Implementierung ausgewählter, zentraler Protokollabläufe, von der Initialisierung bis zum Nachrichtenaustausch.
- **Validierung der Interaktion kryptographischer Primitive:** Praktische Erprobung der korrekten Anwendung und des Zusammenspiels der in Kapitel 5.1.7 spezifizierten kryptographischen Schemata.
- **Veranschaulichung der modularen Übertragbarkeit:** Aufzeigen, dass die modulare Struktur der Spezifikation, insbesondere die Definition abstrakter kryptographischer Schemata, sich gut in eine wartbare Codebasis überführen lässt.
- **Identifikation praktischer Herausforderungen:** Aufdecken potenzieller Herausforderungen und Konkretisierungsbedarfe durch die Implementierung, die in einer rein theoretischen Spezifikation weniger offensichtlich sind (z.B. Daten-Serialisierung, Encoding-Formate, Typ-Handhabung).

Der *PoC* beschränkt sich bewusst auf die zentralen Funktionen der Arbeit und verfolgt nicht den Anspruch, ein vollständiges, produktionsreifes Kommunikationssystem zu realisieren.

6.1. Struktur des Code-Repositorys

Das *PoC*-Code-Repository ist modular aufgebaut. Es folgt einer klaren Trennung zwischen Protokoll-, Kryptografie-, Speicher- und Infrastrukturkomponenten und bildet die theoretischen Konzepte der Spezifikation sauber ab. An dieser Stelle wird

die Struktur des *Repository* lediglich überblicksartig dargestellt, für Details wird die eigenständige Erkundung des *Repository* empfohlen.

- **src/**: Enthält den vollständigen Quellcode des *PoC*, gegliedert in:
  - **crypto/**: Enthält die Implementierung kryptographischer Primitive. Unterteilt in core/, welches pro Primitive (asymmetric/, symmetric/, hash/, kdf/) die zugehörigen Kernalgorithmen kapselt. Dieses Modul wird im weiteren Verlauf des *PoC*-Kapitels als crypto/core referenziert. Zusätzlich enthält util/ Hilfsfunktionen wie GenNonce und GenSalt.
  - **algorithms/**: Enthält die Umsetzung der protokollspezifischen Algorithmen, unterteilt in funktionale Teilbereiche: encryption/, keys/, message/, otp/, storage/, uid/ und utils/.
  - **config/**: Konfigurationsdateien wie paths.py (Pfaddefinitionen) und security_parameters.py (Konstanten für Sicherheitsparameter, vgl. Tabelle 6.1).
  - **protocol/**: Abbildung der Protokoll-Logik:
    - initialize_platform/: Initiale Plattformeinrichtung
    - registration_and_verification_admin/: Registrierung des *Admin*
    - bootstrapping/: Initialisierung von Klassen, Benutzerordnern sowie Registrierung von *Caregivers* und *Teachers*
    - communication/: Nachrichtenaustausch zwischen *Caregivers* und *Teachers*
    - main.py: Einstiegspunkt zur Ausführung des *PoC*
  - **utils/**: Implementierungsspezifische Python-Hilfsfunktionen zur Erstellung und (De-)Serialisierung von Datenstrukturen.
- **storage/**: Simuliert die persistente Dateispeicherung, unterteilt in:
  - **clients/**: Lokale Zustände einzelner *Clients* ($client_a$, $client_c$, $client_t$) in state.json-Dateien
  - **platform/**: Serverseitige Ordnerstruktur der *Platform*, analog zur Beschreibung in Kapitel 4
- **benchmark/**: Enthält das Skript benchmark_crypto.py zur Performance-Messung der verwendeten kryptographischen Primitive.
- **tests/**: Beinhaltet grundlegende Unit-Tests für ausgewählte kryptographische Funktionen zur Verifikation ihrer korrekten Funktionsweise.
- **Root-Verzeichnis**: Enthält zentrale Projektdateien:
  - LICENSE: Lizenzinformationen

- README.md: Anleitung zum Setup und zur Ausführung des *PoC*
- Makefile: Hilft bei der Ausführung und dem Testen des *PoC*
- requirements.txt: Listet alle benötigten Python-Bibliotheken zur automatisierten Installation beim Setup

Die Repository-Struktur unterstützt eine klare Trennung der Verantwortlichkeiten und erleichtert Navigation, Weiterentwicklung sowie die Nachvollziehbarkeit der Spezifikation im Quellcode.

6.2. Implementierungsansatz und Architektur

Dieser Abschnitt beschreibt den gewählten Implementierungsansatz und die grundlegende Architektur des *PoC*, mit Fokus auf die Überführung der theoretischen Konzepte und der modularen Struktur der Spezifikation in eine funktionierende Codebasis.

6.2.1. Technologiewahl und Wahl der kryptographischen Verfahren

Als Programmiersprache wurde Python (Version 3.12) gewählt aufgrund seiner Eignung für schnelle Prototypisierung, der klaren Syntax und der breiten Unterstützung für kryptographische Operationen durch etablierte Bibliotheken.

Bei der Implementierung wurde den Python-Namenskonventionen (PEP 8) gefolgt, sodass Algorithmusnamen aus der Spezifikation in CamelCase (z.B. EncPrivateKeys()) im *PoC* in snake_case (z.B. enc_private_keys()) und Variablennamen durchgehend kleingeschrieben werden. Zur besseren Nachvollziehbarkeit verweist der Docstring jeder relevanten Python-Funktion auf den entsprechenden Pseudocode-Algorithmus. Für eine detaillierte Einsicht in den vollständigen Quellcode wird auf das zu dieser Arbeit gehörende Code-Repository verwiesen.

Als zentrale Bibliothek kommt pyca/cryptography[1] (Version 44.0.2) zum Einsatz. Sie ist aktiv gepflegt, sicherheitsorientiert und stellt robuste Implementierungen gängiger kryptographischer Algorithmen bereit. Im Vergleich zu Alternativen wie PyCryptodome zeichnet sich pyca/cryptography durch eine klar strukturierte API, moderne Sicherheitsstandards und den bewussten Verzicht auf unsichere oder veraltete Verfahren aus [38]. Ihre breite Verwendung in sicherheitskritischen Python-Projekten[2] und Drittbibliotheken macht sie zum de-facto-Standard und unterstützt die langfristige Wartbarkeit und Kompatibilität des *PoC*.

[1] `https://cryptography.io/en/latest/` – Offizielle Dokumentation.
[2] Beispiele für sicherheitskritische Python-Projekte, die pyca/cryptography verwenden, sind unter anderem das Let's-Encrypt-Tool certbot [39] und die SSH-Bibliothek paramiko [40].

Im *PoC* wurden konkrete kryptographische Primitive ausgewählt, die sowohl die Anforderungen der Spezifikation erfüllen als auch mit der eingesetzten Bibliothek pyca/cryptography zuverlässig und sicher implementierbar sind. Die getroffene Auswahl basiert auf einer Abwägung zwischen Sicherheit, Verfügbarkeit in etablierten Bibliotheken und Umsetzbarkeit innerhalb eines prototypischen Kontexts:

- **Rivest–Shamir–Adleman (RSA):** Für asymmetrische Verschlüsselung kommt *RSA Optimal Asymmetric Encryption Padding* (RSA-OAEP) zum Einsatz, das Sicherheit im Sinne von *IND-CPA* garantiert (vgl. Abschnitt 5.1.5). Digitale Signaturen werden mittels *RSA Probabilistic Signature Scheme* (RSA-PSS) erzeugt, welches durch Randomisierung stärkere Sicherheitsgarantien als klassische *RSA*-Signaturen bietet. Das Verfahren erfüllt die in Abschnitt 5.1.5 geforderte *EUF-CMA*-Sicherheit zur Gewährleistung von *Authenticity* und *Integrity*. Die Wahl fiel auf *RSA*, da die Spezifikation dadurch direkt umsetzbar war. ElGamal als Alternative wird von pyca/cryptography nicht unterstützt. *Elliptic Curve Cryptography* (*ECC*) hingegen bietet zwar Unterstützung für Signaturen und Schlüsselaustausch, jedoch keine native asymmetrische Verschlüsselung, sondern erfordert hybride Verfahren wie *Elliptic Curve Integrated Encryption Scheme* (*ECIES*), was die Umsetzung unnötig verkompliziert hätte.
- **Advanced Encryption Standard – Galois/Counter Mode (AES-GCM):** Eingesetzt als *AEAD*-Verfahren zur symmetrischen Verschlüsselung. Es erfüllt die in Abschnitt 5.1.5 formulierten Anforderungen an *Confidentiality*, *Integrity* und *Authenticity* durch die Kombination von Verschlüsselung und Authentifizierung in einem Schema. Die Entscheidung für *AES-GCM* basiert auf dessen Standardisierung, Performance und direkter Unterstützung durch die verwendete Bibliothek.
- **Secure Hash Algorithm 256 (SHA-256):** Eingesetzt für Integritätsprüfungen, Signaturen und Schlüsselableitungen. Die in Abschnitt 5.1.5 geforderten Eigenschaften von Hashfunktionen – *Preimage-resistance*, *Second-preimage-resistance* und *Collision-resistance* – werden erfüllt. Auf *Secure Hash Algorithm 3* (*SHA-3*) wurde bewusst verzichtet, da *PBKDF2* in pyca/cryptography ausschliesslich auf *Secure Hash Algorithm 2* (*SHA-2*)-basierte Funktionen wie *SHA-256* zurückgreift. Um zusätzliche Abhängigkeiten und Sonderfälle im *PoC* zu vermeiden, wurde ein konsistenter Einsatz von *SHA-2* bevorzugt.
- **Password-Based Key Derivation Function 2 (PBKDF2) mit HMAC-SHA256:** Eingesetzt zur sicheren Ableitung kryptographischer Schlüssel aus Passwörtern. Wie in Abschnitt 5.1.5 beschrieben, erfüllt *PBKDF2* mit *Salt* und hoher Iterationszahl die Anforderungen an *Entropy preservation*, *Domain separation* und *Brute-force resistance*. Die Wahl wurde durch die native Unterstützung in pyca/cryptography begünstigt.

Diese Auswahl stellt eine robuste Grundlage dar, um die entworfenen kryptographischen Protokolle realitätsnah zu demonstrieren, ohne zusätzliche Komplexität in

Kauf zu nehmen.

6.2.2. Code-Struktur und Modularität

Ein zentrales Ziel der Umsetzung war eine modulare Architektur, die den Abstraktionsebenen der Spezifikation entspricht. Die in Abschnitt 5.1.7 beschriebenen kryptographischen Schemata wurden als unabhängige Bausteine im Python-Modul `crypto/core` implementiert und bilden die Grundlage für sämtliche übergeordneten Protokollfunktionen.

Diese Modularisierung ermöglicht es, die Primitive als Black-Box-Funktionen zu verwenden. Protokollalgorithmen greifen ausschliesslich über abstrakte Schnittstellen auf diese Primitive zu, ohne deren konkrete Implementierung zu kennen.

Auf diese Weise entsteht eine klare Trennung der Verantwortlichkeiten, die auch den Austausch einzelner Verfahren erlaubt. So kann etwa *RSA* durch ein alternatives asymmetrisches Verfahren wie *ECC* ersetzt werden, indem lediglich die Funktionen `gen_key_pair()`, `enc_a()` und `dec_a()` angepasst werden, ohne dass Änderungen an den darauf aufbauenden Protokollalgorithmen erforderlich sind. Diese Trennung wurde im *PoC* konsequent umgesetzt und erlaubt somit eine flexible Weiterentwicklung der kryptographischen Basis.

6.2.3. Konkretisierung der Spezifikation im Code

Die Umsetzung der Pseudocode-Algorithmen aus Kapitel 5 in ausführbaren Python-Code erforderte eine Reihe technischer Konkretisierungen. Während der Pseudocode bewusst abstrakt gehalten ist, um die Protokolllogik klar und unabhängig von einer konkreten Sprache zu beschreiben, muss der Python-Code zusätzliche praktische Anforderungen erfüllen. Dazu zählen insbesondere Serialisierbarkeit, Kompatibilität mit kryptographischen Funktionen sowie ein effizienter und wartbarer Aufbau.

Ein zentrales Beispiel ist die Abbildung abstrakter Datenstrukturen. In der Spezifikation werden häufig Tupel verwendet, um Inhalte kompakt darzustellen. Für die Implementierung im *PoC* wurden diese Strukturen meist durch `dict`-Objekte ersetzt. Diese erlauben nicht nur den direkten Zugriff auf benannte Felder, sondern lassen sich auch einfacher in JSON-Dateien persistieren. Da das JSON-Format jedoch keine `bytes`-Objekte unterstützt, werden binäre Inhalte wie Schlüssel oder Signaturen zusätzlich in Base64 kodiert und beim Einlesen entsprechend dekodiert. Diese Entscheidungen verbessern nicht nur die Lesbarkeit und Modularität, sondern tragen auch zur Effizienz der Umsetzung bei.

Die Unterschiede zwischen Pseudocode und Programmcode ergeben sich folglich nicht aus inhaltlichen Abweichungen sondern aus der Notwendigkeit, technische Rahmenbedingungen konkret umzusetzen. Die semantische Übereinstimmung mit der Spezifikation bleibt dabei jederzeit gewahrt.

6.3. Kryptographische Primitive im Detail

Dieser Abschnitt stellt die konkreten Python-Implementierungen der kryptographischen Primitive vor, die als Black-Box-Algorithmen in Kapitel 5 (Abschnitt 5.1.7) definiert wurden. Sie bilden die Grundlage der Sicherheitsmechanismen und wurden unter Verwendung der pyca/cryptography-Bibliothek umgesetzt.

6.3.1. Verwendete Sicherheitsparameter

Die Sicherheit der implementierten Operationen basiert auf den in Tabelle 6.1 zusammengefassten Parametern. Diese wurden zentral in src/conlst/security_parameters.py definiert und orientieren sich an aktuellen Empfehlungen [41, 32].

Parameter	Wert (Bits)	Beschreibung
SEC_PARAM_SYM_BIT	256	Schlüssellänge für *AES-GCM*
SEC_PARAM_NONCE_BIT	96	Nonce-Länge für *AES-GCM* [41]
SEC_PARAM_ASYM_BIT	2048	Schlüssellänge für *RSA*
SEC_PARAM_SALT_BIT	128	Salt-Länge für *PBKDF2*
SEC_PARAM_OTP_BIT	128	Bitlänge für *One Time Passwords* ($otps$)
SEC_PARAM_UID_BIT	128	Bitlänge für *User Identifier* (uid)

Tabelle 6.1.: Im *PoC* verwendete kryptographische Sicherheitsparameter.

6.3.2. Asymmetrische Ver- und Entschlüsselung

Die Algorithmen enc_a() (Lst. 6.1) und dec_a() (Lst. 6.2) implementieren RSA-OAEP. *SHA-256* dient als Hashalgorithmus innerhalb des *Optimal Asymmetric Encryption Padding* (*OAEP*)-Schemas. Der Algorithmus enc_a() verschlüsselt den *Plaintext* p mit dem PUK, während dec_a() den *Ciphertext* c mit dem prk entschlüsselt.

```
def enc_a(puk: bytes, p: bytes) -> bytes:
    public_key = serialization.load_pem_public_key(puk)
    c = public_key.encrypt(
        p,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None,
        ),
    )
    return c
```

Listing 6.1: Asymmetrische Verschlüsselung mit RSA-OAEP

```
def dec_a(prk: bytes, c: bytes) -> bytes:
    private_key = serialization.load_pem_private_key(prk,
        password=None)
    p = private_key.decrypt(
        c,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None,
        ),
    )
    return p
```

Listing 6.2: Asymmetrische Entschlüsselung mit RSA-OAEP

6.3.3. Symmetrische Ver- und Entschlüsselung

Die Algorithmen enc_s() (Lst. 6.3) und dec_s() (Lst. 6.4) implementieren *AES-GCM*. Der Algorithmus enc_s() verschlüsselt den *Plaintext* p mit dem sk, der *Nonce* n und den *AD* a, wobei der *Ciphertext* sowie der *Tag* getrennt zurückgegeben werden. Anschliessend entschlüsselt dec_s() den *Ciphertext* und verifiziert den *Tag*.

```
def enc_s(sk: bytes, n: bytes, a: bytes, p: bytes) -> Tuple[bytes,
    bytes]:
    if len(sk) not in {16, 24, 32}:
        raise SystemExit(
            "Protocol␣aborted:␣Key␣must␣be␣128,␣192,␣or␣256␣bits␣in␣
                length."
        )

    aesgcm = AESGCM(sk)
    ct_with_tag = aesgcm.encrypt(n, p, a)

    # AES-GCM appends a 16-byte tag at the end
    ciphertext = ct_with_tag[:-16]
    tag = ct_with_tag[-16:]
    return ciphertext, tag
```

Listing 6.3: Symmetrische Verschlüsselung mit *AES-GCM*

```
def dec_s(sk: bytes, n: bytes, a: bytes, c: bytes, t: bytes) -> bytes:
    if len(sk) not in {16, 24, 32}:
        raise SystemExit(
            "Protocol␣aborted:␣Key␣must␣be␣128,␣192,␣or␣256␣bits␣in␣
                length."
        )

    aesgcm = AESGCM(sk)

    # AESGCM expects ciphertext || tag as a single input
    ct_with_tag = c + t
    return aesgcm.decrypt(n, ct_with_tag, a)
```

Listing 6.4: Symmetrische Entschlüsselung mit *AES-GCM*

6.4. Demonstration ausgewählter Protokollschritte

Zur Demonstration der praktischen Funktionsfähigkeit und der korrekten Interaktion der Protokollalgorithmen wurden zentrale Protokollschritte implementiert. Diese exemplarische Umsetzung ermöglicht es, die spezifizierten Abläufe nachzuvollziehen.

Für eine detaillierte Gegenüberstellung von Spezifikation und Python-Implementierung wurde Prot. 4 (Erstellung des Registrierungspakets durch den *Teacher*) ausgewählt. Dieses ist besonders repräsentativ, da es die Schlüsselgenerierung, symmetrische sowie asymmetrische Verschlüsselungsverfahren, digitale Signaturen und die Datenablage umfasst. Die Analyse verdeutlicht die hohe Übereinstimmung zwischen dem Pseudocode aus Kapitel 5 und der Umsetzung, mit nur geringfügigen, implementierungsspezifischen Anpassungen.

6.4.1. Gegenüberstellung von Spezifikation und Implementierung

Abbildung 6.1 illustriert den Protokollschritt 4, dessen Implementierung als Python-Funktion register_teacher() in Listing 6.5 gezeigt wird. Die nummerierten Kommentare im Code (Listing 6.5) korrespondieren dabei mit den wesentlichen Schritten des in Abbildung 6.1 dargestellten Protokollschrittes.

Der in Abbildung 6.1 spezifizierte Ablauf wurde nahezu eins zu eins umgesetzt, wie die Korrespondenz der nummerierten Kommentare in Listing 6.5 mit den Operationen des Protokollschritts zeigt. Beispielsweise werden die Interaktionen zwischen dem *Teacher* und seinem Client ($client_t$), angedeutet durch die Operationen 1 und 4 im Protokollschritt (Abbildung 6.1) und markiert durch die Kommentare `# Nr. 1` bzw. `# Nr. 4` im Code (Listing 6.5), durch Mock-Funktionen simuliert. Der Datentransfer zur *Platform* (Operation 6) ist durch den direkten Aufruf von store_rptea() abgebildet, entsprechend StoreRPTea(). Eine praktische Konkretisierung zeigt sich in Operation 6 (Listing 6.5, vor `# Nr. 6`), wo die Signatur des Keystores (ks_sig_dict) vor der Persistenz mittels build_sig_file() in ein serialisierbares Format konvertiert wird.

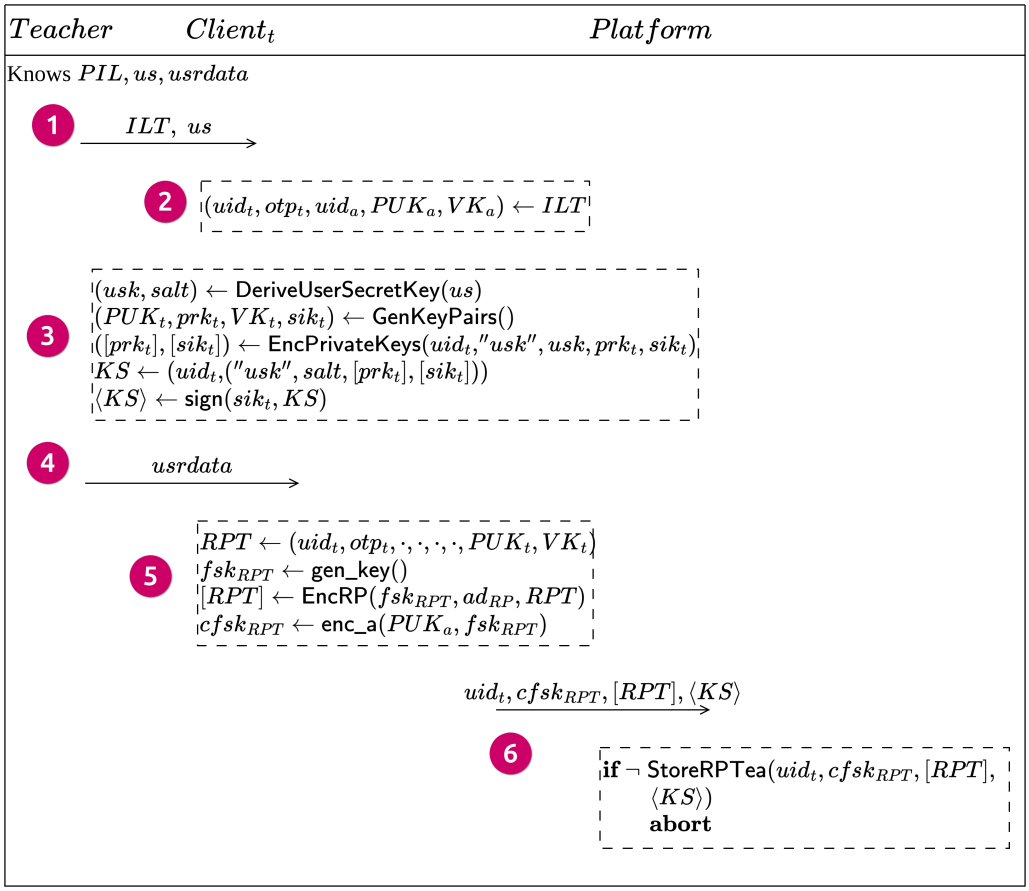


Abbildung 6.1.: Protokollschritt 4 zur Gegenüberstellung mit der *PoC*-Implementierung (register_teacher()) in Listing 6.5, welche die Operationen des Protokollschrittes durch nummerierte Kommentare referenziert.

```
def register_teacher():
    # --- Load information from the invitation letter ---
    # Simulates the teacher scanning the invitation letter and
    # entering their user secret
    (uid_t, otp_t, uid_a, puk_a, vk_a) = load_invitation_letter() #
        Nr. 2
    us = create_secret("teacher") # Nr. 1

    # --- Derive user secret key and generate key pairs ---
    # Nr. 3
    usk, salt = derive_user_secret_key(us)
    (puk_t, prk_t, vk_t, sik_t) = gen_key_pairs()
    (enc_prk, enc_sik) = enc_private_keys(uid_t, "usk", usk, prk_t,
        sik_t)
    ks = build_ks(uid_t, [build_ks_entry("usk", salt, enc_prk,
        enc_sik)])
    ks_sig = sign(sik_t, jstrb(ks))

    # Simulates the teacher entering their profile information
    usrdata = mock_teacher_userdata() # Nr. 4

    # --- Create and encrypt the registration package ---
    # Nr. 5
    rpt = build_rpt(uid_t, otp_t, puk_t, vk_t, usrdata)
    fsk_rpt = gen_key()
    ad_rp = f"creator:{uid_t}|target:public|
␣␣␣␣label:registrationpackage|algorithm:AES-GCM|version:v1"
    rpt_enc = enc_rp(fsk_rpt, ad_rp, jstr(rpt))
    cfsk_rpt = enc_a(puk_a, fsk_rpt)

    # --- Persist the encrypted RPT and keystore ---
    ks_sig_dict = build_sig_file(uid_t, ks_sig, "RSA-PSS-SHA256")
    # Nr. 6
    if not store_rptea(uid_t, cfsk_rpt, rpt_enc, (ks, ks_sig_dict)):
        raise SystemExit(
            "Protocol␣aborted:␣Unable␣to␣persist␣encrypted␣RPT␣and␣
                keystore␣from␣teacher."
        )
```

Listing 6.5: *PoC*-Implementierung der Funktion register_teacher() für Protokollschritt 4 (vgl. Abbildung 6.1). Nummerierte Kommentare im Code (`# Nr. X`) verweisen auf korrespondierende Schritte im Protokollschritt.

6.4.2. Umsetzung der Pseudocode-Algorithmen

Auch die im Rahmen von Protokollschritt 4 eingesetzten Pseudocode-Algorithmen wurden spezifikationsnah implementiert. Die Abbildungen 6.2, 6.3 und Listings 6.6, 6.7 zeigen exemplarisch die Gegenüberstellung des jeweiligen Pseudocodes mit seiner Python-Implementierung im *PoC*.

Algorithm 28: GenKeyPairs()

$(PUK, prk) \leftarrow$ gen_key_pair_enc()
$(VK, sik) \leftarrow$ gen_key_pair_sig()

return $(PUK, prk, VK, sik) \in \mathcal{K}_{PUK} \times \mathcal{K}_{prk} \times \mathcal{K}_{VK} \times \mathcal{K}_{sik}$

Abbildung 6.2.: Pseudocode des Algorithmus GenKeyPairs() (vgl. Implementierung in Listing 6.6).

```
def gen_key_pairs() -> Tuple[bytes, bytes, bytes, bytes]:
    puk, prk = gen_key_pair_enc()
    vk, sik = gen_key_pair_sig()
    return puk, prk, vk, sik
```

Listing 6.6: *PoC*-Implementierung gen_key_pairs() des Pseudocode-Algorithmus aus Abbildung 6.2.

Algorithm 29: EncPrivateKeys($uid, skid, usk, prk, sik$)

Input: *User Identifier* $uid \in \mathcal{U}$
Secret key identifier $skid \in \Sigma^\star$
User secret key $usk \in \mathcal{K}$
User private key $prk \in \mathcal{K}_{prk}$
User signature key $sik \in \mathcal{K}_{sik}$

$nonce_{prk} \leftarrow$ GenNonce() // see Alg. 4
$(ct_{prk}, tag_{prk}) \leftarrow$ enc_s$(usk, nonce_{prk}, ad_{prk}, prk)$
$[prk] \leftarrow (nonce_{prk}, ad_{prk}, ct_{prk}, tag_{prk})$

$nonce_{sik} \leftarrow$ GenNonce()
$(ct_{sik}, tag_{sik}) \leftarrow$ enc_s$(usk, nonce_{sik}, ad_{sik}, sik)$
$[sik] \leftarrow (nonce_{sik}, ad_{sik}, ct_{sik}, tag_{sik})$

return $([prk],\ [sik])$ // $([prk],\ [sik]) \in (\mathcal{N} \times \mathcal{A} \times \mathcal{C} \times \mathcal{T})^2$

Abbildung 6.3.: Pseudocode des Algorithmus EncPrivateKeys() (vgl. Implementierung in Listing 6.7).

Die Implementierungen folgen der Logik des Pseudocodes sehr genau. Geringfügige Abweichungen ergeben sich primär aus Python-spezifischen Erfordernissen, wie der expliziten Konvertierung von Strings zu Bytes oder der Nutzung von Typ-Annotationen.

```
def enc_private_keys(uid: str, skid: str, usk: bytes, prk: bytes,
    sik: bytes) -> Tuple[Tuple[bytes, str, bytes, bytes],
    Tuple[bytes, str, bytes, bytes]]:
    # Construct associated data as uid|type|skid and encode to bytes
    ad_prk = f"creator:{uid}|target:{uid}|label:prk|
␣␣␣␣algorithm:AES-GCM|version:v1"
    ad_sik = f"creator:{uid}|target:{uid}|label:sik|
␣␣␣␣algorithm:AES-GCM|version:v1"

    # Encrypt prk using usk and associated data
    nonce_prk = gen_nonce()
    ct_prk, tag_prk = enc_s(usk, nonce_prk, tbytes(ad_prk), prk)
    prk_enc = (nonce_prk, ad_prk, ct_prk, tag_prk)

    # Encrypt sik using usk and associated data
    nonce_sik = gen_nonce()
    ct_sik, tag_sik = enc_s(usk, nonce_sik, tbytes(ad_sik), sik)
    sik_enc = (nonce_sik, ad_sik, ct_sik, tag_sik)

    return prk_enc, sik_enc
```

Listing 6.7: *PoC*-Implementierung enc_private_keys() des Pseudocode-Algorithmus aus Abbildung 6.3.

6.5. Entwicklungs- und Qualitätssicherungspraktiken

Obwohl der primäre Fokus des *PoC* auf der Demonstration kryptographischer Machbarkeit lag, wurden grundlegende Praktiken zur Sicherstellung von Codequalität und -sicherheit angewendet, angelehnt an einen *SecDevOps*-Ansatz. Ein Makefile (Listing B.2) automatisiert diese Prozesse.

Die definierten Makefile-Ziele umfassen:

- **test, coverage:** Ausführung der Testfälle mit pytest und Generierung eines Coverage-Reports zur Analyse der Testabdeckung.
- **lint, format, check-format:** Überprüfung des Codes mit ruff auf stilistische Konsistenz und potenzielle Fehler. Zudem wird automatische Formatierung unterstützt.
- **bandit:** Statische Sicherheitsanalyse zur Erkennung typischer Schwachstellen in Python-Code.
- **audit:** Überprüfung der verwendeten Abhängigkeiten mittels pip-audit auf bekannte Sicherheitslücken.
- **setup:** Initialisierung einer Python-Virtualenv und Installation aller Projektabhängigkeiten.
- **run:** Start des PoC-Hauptprogramms (src/protocol/main.py) zur Ausführung eines vollständigen Ablaufes.
- **clean:** Bereinigung temporärer Artefakte wie _pycache_, Coverage-Dateien und Caches.

Diese automatisierten Prüfungen tragen zur Robustheit des entwickelten Codes bei.

Es sei angemerkt, dass keine Unit-Tests im klassischen Sinne implementiert wurden. Der *PoC* selbst fungiert als integraler Test der Protokollabläufe, dessen erfolgreiche Durchführung die korrekte Funktionsweise der Primitive im Zusammenspiel validiert. Die Erstellung dedizierter Unit-Tests gegen extern erzeugte kryptographische Daten lag ausserhalb des Projektumfangs.

6.6. Ergebnisse und Beobachtungen

Der *PoC* lieferte wesentliche Erkenntnisse zur praktischen Umsetzbarkeit und Robustheit der Spezifikation. Zentrales Ergebnis war die erfolgreiche Implementierung und Validierung aller spezifizierten Protokollabläufe, was einen Durchstich von der Benutzerregistrierung bis zum Nachrichtenaustausch demonstrierte und die Machbarkeit des kryptographischen Gesamtkonzepts bestätigte.

Der modulare Aufbau der Spezifikation, insbesondere die Kapselung der kryptographischen Schemata in Black-Box-Algorithmen und deren Umsetzung im Python-Modul crypto/core, konnte erfolgreich nachgebildet werden und erwies sich als vorteilhaft für Code-Struktur sowie potenzielle Wartbarkeit. Als primäre Herausforderung zeigte sich die Abbildung abstrakter Datenstrukturen und Typen des Pseudocodes auf Python-Datentypen, was iterative Anpassungen bei Konvertierungen (z.B. für kryptographische Bibliotheken) erforderte. Die Implementierungsarbeit führte zudem zu wertvollen Rückkopplungen, wodurch die Spezifikation in Kapitel 5 iterativ verfeinert und deren Konsistenz sowie Vollständigkeit verbessert werden konnte.

6.6.1. Performance der eingesetzten Kryptographie

Die Optimierung der Performance war kein Ziel dieses *PoC*. Dennoch wird im Folgenden eine pragmatische Analyse der wichtigsten kryptographischen Operationen durchgeführt, basierend auf den real implementierten Algorithmen im Python-Modul crypto/core.

Die Erstellung digitaler Signaturen ist die aufwendigste kryptographische Operation im *PoC*, da sie wie auch das Entschlüsseln rechenintensive Berechnungen mit dem privaten Schlüssel erfordert. Bei RSA ist der private Exponent deutlich grösser als der öffentliche, was längere Rechenzeiten zur Folge hat. Entsprechend ist vor allem die Performance beim Signieren und Entschlüsseln von Interesse.

Signaturen im Schulalltag

Signaturen sind im regulären Plattformbetrieb häufig, aber in der Regel einzeln. Ein *Caregiver* signiert zum Beispiel bei der Erstellung seines *Registrationpackage Caregiver* nur den zugehörigen *Keystore*, beim Versand von Nachrichten jeweils eine Nachricht. Gleiches gilt für *Teachers*. Der *Admin* hingegen signiert beim Schuleintritt neuer *Students* deutlich mehr Dateien, was den rechenintensivsten Anwendungsfall darstellt.

Bei drei ersten Klassen mit je 20 *Students* ergibt sich im Maximalfall:

- eine Signatur für das Profil des *Students* (profile.json),
- eine Signatur für die Absenzen-Datei (absences.json),
- eine Signatur für die *Teacher*-Datei (teachers.json),
- eine Signatur für die *Caregiver*-Datei (caregivers.json),
- eine Signatur für den Eintrag in die Klassendatei (students.json).

Dies ergibt bis zu $3 \cdot 20 \cdot 5 = 300$ Signaturen. Da einige Dateien nur einmal pro Klasse signiert werden, liegt die tatsächliche Zahl darunter.

6.6.2. Benchmark und Systemprofil

Ein dediziertes Benchmark-Skript (siehe Listing B.1) mass die Anzahl Operationen pro Sekunde auf realer Hardware. Auf einem Laptop mit CPU von Ende 2018 erreichten RSA-OAEP und RSA-PSS (2048 Bit) ausreichende Werte, um 300 Operationen in akzeptabler Zeit zu verarbeiten. Die symmetrische *AES-GCM*-Verschlüsselung war erwartungsgemäss deutlich schneller. Siehe dazu die Abbildung 6.4.

```
nicolin@ubuntu ~> python3 benchmark/benchmark_crypto.py
System information for benchmark:
  System         : Linux 6.11.0-26-generic
  CPU model      : Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz
  CPU cores      : 4 physical, 8 logical
  RAM (total)    : 46.71 GB
  Python version: 3.12.3
  Asym. keysize : 2048 bit
  Sym. keysize  : 256 bit

RSA-OAEP Encrypt:    0.025s for 300 ops (12013.1 ops/sec)
RSA-OAEP Decrypt:    13.905s for 300 ops (21.6 ops/sec)
RSA-PSS Sign:        12.810s for 300 ops (23.4 ops/sec)
RSA-PSS Verify:      0.023s for 300 ops (13266.0 ops/sec)
AES-GCM Encrypt:     0.002s for 300 ops (171359.3 ops/sec)
AES-GCM Decrypt:     0.002s for 300 ops (168175.8 ops/sec)
```

Abbildung 6.4.: Benchmark der kryptographischen Kernfunktionen auf einem Laptop mit Intel Core i7-8565U. Gemessen wurden asymmetrische Operationen mit RSA-OAEP und RSA-PSS (2048 Bit) sowie symmetrische Verschlüsselung mit *AES-GCM* (256 Bit).

6.6.3. Bewertung

Die Erstellung von 300 Signaturen in rund 14 Sekunden erscheint auf den ersten Blick vergleichsweise langsam. Dabei ist zu berücksichtigen, dass das Benchmark-Skript ausschliesslich auf einem CPU-Kern ausgeführt wird. Da heutige Systeme in der Regel über mehrere Kerne verfügen, lassen sich in realen Anwendungen deutlich höhere Durchsätze erzielen. Zudem ist Python als Programmiersprache für Performance-kritische Aufgaben nur bedingt geeignet. In praxisnahen Umgebungen kämen effizientere Sprachen wie Java oder C++ zum Einsatz, was eine deutlich bessere Laufzeit erwarten lässt. Darüber hinaus erlaubt die modulare Spezifikation der *Platform* den Austausch von *RSA* durch effizientere Verfahren wie *ECC*. Unter realen Bedingungen ist daher nicht mit signifikanten Wartezeiten für die Nutzenden

zu rechnen.

6.7. Einordnung und technische Grenzen des Proof of Concepts

Wie in der Einleitung zu diesem Kapitel (vgl. 6) erläutert, verfolgt der *PoC* primär das Ziel, die im Kapitel 5 beschriebenen kryptographischen Konzepte und Protokolle in einer lauffähigen Umgebung zu veranschaulichen, erhebt aber nicht den Anspruch auf Vollständigkeit.

Um die Transparenz zu wahren, sind im Folgenden die wichtigsten und bewusst gesteckten technischen Grenzen zusammengefasst:

- Die Netzwerkkommunikation wurde nicht implementiert. Datenaustausch zwischen Parteien erfolgt lokal innerhalb desselben Prozesses.
- Es existiert keine grafische Benutzeroberfläche. Die Interaktion erfolgt über direkt aufrufbare Python-Funktionen mit einfacher Konsolenausgabe.
- Die Fehlerbehandlung ist auf grundlegende Prüfungen und kritische Ausnahmen beschränkt. Die Entwicklung robuster Wiederherstellungsmechanismen oder differenzierter Fehlermeldungen war nicht Teil des Konzepts.

Der entwickelte *PoC* ist daher als gezielte Machbarkeitsstudie zu verstehen.

6.8. Fazit zum Proof of Concept

Der *PoC* hat erfolgreich die praktische Umsetzbarkeit der spezifizierten kryptographischen Mechanismen und Protokolle für eine datenschutzfreundliche Schulkommunikationslösung demonstriert. Trotz bewusst gesetzter technischen Grenzen (Abschnitt 6.7) wurde ein funktionaler Durchstich durch alle wesentlichen Protokolle realisiert.

Die Ergebnisse bestätigen, dass:

- die definierten Algorithmen mit Standardbibliotheken in Python implementierbar sind,
- der modulare Aufbau der Spezifikation sich gut in eine strukturierte Codebasis übertragen lässt, und
- die entworfenen Protokolle in ihrer Logik wie vorgesehen funktionieren.

Mit dem entwickelten *PoC* wurde ein belastbares Fundament gelegt, das nicht nur das angestrebte Ziel der Arbeit erfüllt, sondern auch das Potenzial einer durchgängig verschlüsselten, datenschutzorientierten Schulkommunikationsplattform greifbar macht. Die Ergebnisse motivieren zur Weiterentwicklung, etwa durch Integrati-

on rudimentärer Netzwerkkommunikation oder einer benutzerfreundlichen Oberfläche. Gleichzeitig unterstreicht der erreichte Stand, dass auch unter realistischen Rahmenbedingungen ein hohes Mass an Datenschutz und Datensouveränität technisch realisierbar ist.

7. Diskussion und Ausblick

Dieses Kapitel diskutiert die zentralen Ergebnisse der vorliegenden Bachelorarbeit kritisch, bewertet deren Bedeutung und Zuverlässigkeit, zieht Schlussfolgerungen für die Praxis und leitet daraus Empfehlungen für zukünftige Forschungs- und Entwicklungsarbeiten ab.

7.1. Diskussion

Das Kernziel dieser Bachelorarbeit, die Spezifikation einer Schulkommunikationslösung, die konsequent den Prinzipien von *Privacy by Design* folgt und dabei datenschutzkonforme 1:1- sowie Gruppenkommunikation ohne wesentliche Kompromisse bei Benutzerfreundlichkeit oder essenziellen Funktionen wie der Datensicherung ermöglicht, konnte erreicht werden. Als zentrales Ergebnis liegt eine detaillierte, modulare und wiederverwendbare Spezifikation vor. Diese umfasst Algorithmen in Pseudocode für Kernprozesse wie die Benutzerregistrierung, den authentifizierten Schlüsselaustausch (*Bootstrapping*) und den Nachrichtenaustausch unter *End-to-End Encryption*.

Die Entscheidung für einen *S/MIME*-inspirierten, dateibasierten Ansatz gegenüber dem *Double Ratchet*-Protokoll erwies sich im schulischen Kontext als zielführend. Die Anforderungen an eine zentrale Persistierung der Daten, insbesondere der Nachrichten, sowie die Integration eines niederschwelligen *Bootstrapping*-Verfahrens konnten mit diesem Ansatz besser erfüllt werden, was dessen Praxistauglichkeit unterstreicht. Die technische Machbarkeit der Kernkonzepte und der eingesetzten kryptographischen Primitive wurde durch den *Proof of Concept* exemplarisch demonstriert. Die vorgelegte Spezifikation und der *PoC* zeigen somit, dass die Entwicklung einer datenschutzkonformen Schulkommunikationslösung technisch und praktisch realisierbar ist und auch komplexe Anforderungen wie *E2EE* für Gruppenkommunikation sowie zentrale Persistierung in einem benutzerfreundlichen System vereint werden können.

Die Verlässlichkeit der vorgestellten Lösung basiert auf der sorgfältigen Ausarbeitung der Spezifikation, die in etablierten kryptographischen Prinzipien verankert ist, und der Validierung der Kernideen durch den *PoC*. Kritisch zu bewerten ist hierbei jedoch, dass der *PoC* eine prototypische Umsetzung darstellt und somit keine umfassenden Tests unter realen Bedingungen oder Lastszenarien beinhaltet. Die praktische Skalierbarkeit und Performance des Systems unter realer Netzwerklast

oder bei einer grossen Anzahl gleichzeitiger Nutzer bleiben somit Gegenstand zukünftiger Untersuchungen. Des Weiteren konnte die Benutzerfreundlichkeit der konzipierten Abläufe, insbesondere des Bootstrapping-Prozesses, im Rahmen des *PoC* nicht evaluiert werden. Obwohl diese theoretisch als niederschwellig konzipiert wurden, bedarf es hier mindestens einer Pilotierung oder einfacherer Akzeptanztests, um die Annahmen bezüglich der Praxistauglichkeit für eine heterogene Nutzerschaft (Lehrpersonen, Erziehungsberechtigte) zu überprüfen.

Trotz dieser Limitationen ergänzt die erarbeitete Spezifikation bestehendes Wissen, indem sie einen konkreten, technisch detailliert ausgearbeiteten Entwurf für eine *PbD*-orientierte Schulkommunikationslösung liefert. Sie schlägt somit eine Brücke zwischen abstrakten Datenschutzanforderungen und deren praktischer Implementierung. Im Vergleich zu vielen kommerziellen Lösungen, die Datenschutz oft erst nachträglich oder unzureichend (z.B. durch Verzicht auf *E2EE*) berücksichtigen, demonstriert diese Arbeit einen alternativen, konsequent *PbD*-fokussierten Weg. Dessen Notwendigkeit wird durch wiederkehrende Datenschutzvorfälle im Bildungssektor, wie dem in der Einleitung erwähnten Fall der App *Stay Informed*, eindrücklich unterstrichen. Die hier präsentierte Arbeit stützt somit die Erkenntnis, dass etablierte kryptographische Konzepte eine robuste Grundlage für sichere Kommunikation im sensiblen schulischen Umfeld bieten können und stellt die gängige Annahme, dass hohe Sicherheit zwangsläufig zu Lasten der Benutzerfreundlichkeit oder essenzieller Funktionen gehen muss, zumindest fundiert in Frage.

Der Fokus dieser Arbeit lag explizit auf der Spezifikation der Sicherheitsarchitektur und der kryptographischen Kernprozesse. Bestimmte funktionale Aspekte, wie die spezifische Funktion zur Meldung von Absenzen oder die detaillierte Ausgestaltung eines Klassenchats, wurden aus zeitlichen Gründen weder in der Spezifikation vollständig ausgearbeitet noch im *PoC* implementiert. Obwohl diese Features für eine umfassende Schulkommunikationslösung relevant sind, würde die konzipierte, flexible Nachrichtenstruktur deren nachträgliche Ergänzung mit voraussichtlich überschaubarem Aufwand ermöglichen. Die Absenzmeldung könnte beispielsweise über speziell formatierte Nachrichten realisiert werden, und da der grundlegende Nachrichtenaustausch bereits spezifiziert und im PoC demonstriert wurde, wäre auch die Erweiterung zu einem voll funktionsfähigen Klassenchat ein logischer nächster Schritt. Ebenso zeigten sich bei der Entwicklung des *PoC* geringfügige Abweichungen von der abstrakten Spezifikation, bedingt durch Python-spezifische Anforderungen oder pragmatische Entscheidungen zur Sicherstellung eines funktionsfähigen Prototyps. Diese Anpassungen tangierten jedoch nicht die Validität der Kernkonzepte oder die grundlegende Sicherheitsarchitektur, was die Robustheit des modularen Entwurfs unterstreicht.

Zusammenfassend bietet die vorgelegte Spezifikation eine solide Grundlage und kann als Blaupause für zukünftige Entwicklungen im Bereich datenschutzkonformer Schulkommunikationslösung dienen. Softwareentwicklerinnen und -entwickler, IT-Sicherheitsverantwortliche sowie Anbieter von Schulkommunikationslösung erhalten ein

detailliertes technisches Konzept, das als Basis für die Neuentwicklung oder die Weiterentwicklung bestehender Systeme unter Berücksichtigung strenger Datenschutzanforderungen dienen kann. Die Resultate dieser Arbeit regen somit dazu an, den Fokus bei der Systementwicklung verstärkt auf inhärent sichere Architekturen zu legen.

7.2. Ausblick

Aufbauend auf den Ergebnissen und den identifizierten Limitationen dieser Arbeit eröffnen sich diverse Ansatzpunkte für weiterführende Forschungs- und Entwicklungsprojekte. Diese ergeben sich direkt aus der Notwendigkeit, die theoretischen Konzepte und den prototypischen Nachweis in eine praxisreife und umfassend validierte Lösung zu überführen:

- **Vollständige Implementierung, Pilotprojekte und Evaluation:** Ein zentraler nächster Schritt wäre die vollständige Implementierung der konzipierten *Platform*, einschliesslich der noch fehlenden funktionalen Aspekte wie der Absenzmeldung, dem Klassenchat und einer rudimentären Benutzeroberfläche. Daran anschliessend ist die Erprobung in Pilotprojekten an Schulen von besonderem Interesse. Solche Praxistests würden nicht nur die technische Machbarkeit und Praxistauglichkeit unter realen Bedingungen validieren, sondern auch wertvolle Erkenntnisse zur Benutzerakzeptanz und Usability (insbesondere des Bootstrapping-Prozesses sowie der Backup-Funktionalität bei der Zielgruppe Lehrpersonen, Erziehungsberechtigte und Schuladministration), zur Skalierbarkeit der Lösung in grösseren Umgebungen und zu potenziellen Optimierungen liefern. Die Sicherheit eines Systems hängt schliesslich massgeblich von seiner korrekten und konsistenten Nutzung ab.
- **Skalierbarkeits- und Performance-Analysen:** Die Durchführung von Performance-Analysen unter Last ist notwendig, um die Skalierbarkeit der Lösung für grössere Schulen oder Schulverbünde zu untersuchen und potenzielle Engpässe frühzeitig zu identifizieren. Dies steht im direkten Verhältnis zur Limitation des aktuellen *PoC*, der solche Aspekte nur bedingt abdecken konnte.
- **Integration mit bestehenden Schulverwaltungssystemen:** Eine Untersuchung zur technischen und organisatorischen Integration der konzipierten Schulkommunikationslösung mit bestehenden Schulverwaltungssystemen (z.B. für den Import von Nutzerdaten oder die Synchronisation von Absenzen) würde die praktische Relevanz und den Implementierungsaufwand für Schulen weiter konkretisieren.
- **Optimierung der kryptographischen Verfahren:** Eine vielversprechende Weiterentwicklung wäre die Evaluation und der potenzielle Ersatz des aktuell verwendeten *RSA*-Verfahrens durch modernere und effizientere asymmetrische

Algorithmen, wie beispielsweise *ECC*. Ein solcher Wechsel könnte insbesondere bei der Signaturerstellung und -verifikation sowie bei der Entschlüsselung zu einer signifikanten Performance-Steigerung führen, was die Benutzerfreundlichkeit und Skalierbarkeit der *Platform* weiter verbessern würde, ohne die etablierten Sicherheitsgarantien zu kompromittieren.

Die Realisierung dieser Ausblicke könnte dazu beitragen, das Vertrauen in digitale Kommunikationslösungen im Bildungsbereich nachhaltig zu stärken und den Weg für eine breitere Anwendung datenschutzfreundlicher Technologien zu ebnen.

Eigenständigkeitserklärung

Ich bestätige mit meiner Unterschrift, dass ich meine vorliegende Bachelor-Thesis selbständig durchgeführt habe. Alle Informationsquellen (Fachliteratur, Besprechungen mit Fachleuten, usw.) und anderen Hilfsmittel, die wesentlich zu meiner Arbeit beigetragen haben, sind in meinem Arbeitsbericht im Anhang vollständig aufgeführt. Sämtliche Inhalte, die nicht von mir stammen, sind mit dem genauen Hinweis auf ihre Quelle gekennzeichnet. Ich bestätige weiterhin, dass ich bei der Erstellung dieser Studienarbeit durchgehend steuernd gearbeitet habe und von einer KI erzeugte Inhalte nicht unreflektiert übernommen habe.

12. Juni 2025

A. Vejseli | N. Dora

Literatur

[1] Holger Bleich und Ronald Eikenberg. *Datenleck bei beliebter KiTa-App Stay Informed*. März 2024. URL: https://www.heise.de/news/Datenleck-bei-beliebter-KiTa-App-Stay-Informed-9662578.html (besucht am 23. 02. 2025).

[2] Adrienne Fichter, Patrick Seemann und Lisa Rock. *Wollen Sie wissen, womit Viola Amherd geimpft ist?* März 2021. URL: https://www.republik.ch/2021/03/23/wollen-sie-wissen-womit-viola-amherd-geimpft-ist (besucht am 23. 02. 2025).

[3] Abidin Vejseli und Nicolin Dora. „Privacy by Design in Schulkommunikations-Apps“. Projektbericht im Studiengang Bachelor of Science in Computer Science, Berner Fachhochschule, betreut von Prof. Dr. Philipp Locher. Jan. 2025.

[4] Lukas Wüthrich. *Schweizer Bildung im digitalen Wandel*. Nov. 2023. URL: https://www.itmagazine.ch/artikel/80927/Schweizer_Bildung_im_digitalen_Wandel.html (besucht am 23. 02. 2025).

[5] Marius Beerli. *Das Digitale ist zu einem Taktgeber in der Bildung geworden – Teil 1*. Feb. 2022. URL: https://www.edk.ch/de/die-edk/blog/220224 (besucht am 23. 02. 2025).

[6] Matthias Stürmer. *Chancen und Herausforderungen der Digitalisierung*. Nov. 2022. URL: https://www.bfh.ch/dam/jcr:40e305dc-135f-40f4-bae9-3f862fc1eed8/Seiten%20aus%20Schulblatt_17-2022_low_M.St%C3%BCrmer.pdf (besucht am 23. 02. 2025).

[7] Untis GmbH. *Untis Messenger – Die DSGVO-konforme Schulkommunikation*. Zugriff am 18. April 2025. 2025. URL: https://www.untis.at/ (besucht am 18. 04. 2025).

[8] Fox Education Services GmbH. *SchoolFox – Die All-in-One App für die Schule*. Zugriff am 18. April 2025. 2025. URL: https://foxeducation.com/ (besucht am 18. 04. 2025).

[9] Sdui GmbH. *Sdui – Die Plattform für datenschutzkonforme Schulkommunikation*. Zugriff am 18. April 2025. 2025. URL: https://sdui.de/ (besucht am 18. 04. 2025).

[10] Threema GmbH. *Threema Education – Sicher kommunizieren in Bildungseinrichtungen*. Zugriff am 18. April 2025. 2025. URL: https://threema.ch/de/work/bildung (besucht am 18. 04. 2025).

[11] Heinekingmedia GmbH. *schul.cloud – Die DSGVO-konforme Messenger-Lösung mit Dateiablage*. Zugriff am 18. April 2025. 2025. URL: https://schul.cloud/ (besucht am 18. 04. 2025).

[12] IServ GmbH. *IServ Schulplattform – Datenschutz und Sicherheit*. Zugriff am 18. April 2025. 2025. URL: https://iserv.de/ (besucht am 18. 04. 2025).

[13] Henk van Rossum u. a. *Privacy-Enhancing Technologies: The Path to Anonymity*. Den Haag: Information und Privacy Commissioner / Ontario, Canada & Registratiekamer, The Netherlands, Aug. 1995.

[14] Peter Hustinx. „Privacy by design: delivering the promises". In: *Identity in the Information Society* 3.2 (2010), S. 253–255. ISSN: 1876-0678. DOI: 10.1007/s12394-010-0061-z. URL: https://doi.org/10.1007/s12394-010-0061-z.

[15] Ph.D. Ann Cavoukian. *Privacy by Design - The 7 Foundational Principles*. Jan. 2011. URL: https://www.datatilsynet.no/globalassets/global/english/7foundationalprinciples_anncavoukian2.pdf (besucht am 16. 11. 2024).

[16] Eidgenössischer Datenschutz- und Öffentlichkeitsbeauftragter (EDÖB). *Das neue Datenschutzgesetz aus Sicht des EDÖB*. Feb. 2021. URL: https://www.edoeb.admin.ch/dam/edoeb/de/Dokumente/datenschutz/Leitfaden%20Das%20neue%20Datenschutzgesetz%20aus%20Sicht%20des%20ED%C3%96B_20221009.pdf.download.pdf/Leitfaden%20Das%20neue%20Datenschutzgesetz%20aus%20Sicht%20des%20ED%C3%96B_20221009.pdf (besucht am 16. 11. 2024).

[17] Schweizerische Eidgenossenschaft. *Bundesgesetz über den Datenschutz (DSG)*. 2022. URL: https://www.fedlex.admin.ch/eli/cc/2022/491/de (besucht am 16. 11. 2024).

[18] Eidgenössischer Datenschutz- und Öffentlichkeitsbeauftragter (EDÖB). *Leitfaden zu den technischen und organisatorischen Massnahmen des Datenschutzes (TOM)*. Jan. 2024. URL: https://www.edoeb.admin.ch/dam/edoeb/de/Dokumente/datenschutz/leitfaden_tom.pdf.download.pdf/TOM_DE.pdf (besucht am 16. 11. 2024).

[19] IBM. *Was ist End-to-End-Verschlüsselung (E2EE)?* 2024. URL: https://www.ibm.com/de-de/topics/end-to-end-encryption (besucht am 15. 12. 2024).

[20] TeamDrive. *Ende-zu-Ende-Verschlüsselung: Definition, Funktion und Vorteile der sicheren Kodierung*. 2024. URL: https://teamdrive.com/blog-de/ende-zu-ende-verschluesselung-im-ueberblick/ (besucht am 15. 12. 2024).

[21] TeamWire. *Ende-zu-Ende-Verschlüsselung (E2EE)*. 2024. URL: https://teamwire.eu/lexikon/ende-zu-ende-verschluesselung/ (besucht am 15. 12. 2024).

[22] Jonathan Katz und Yehuda Lindell. *Introduction to Modern Cryptography (3rd edition)*. Chapman & Hall/CRC, 2020. ISBN: 978-0-8153-5436-9.

[23] Moxie Marlinspike Trevor Perrin. *The Double Ratchet Algorithm*. Nov. 2016. URL: https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf (besucht am 24. 03. 2025).

[24] Trevor Perrin und Moxie Marlinspike. *The X3DH Key Agreement Protocol*. Nov. 2016. URL: https://signal.org/docs/specifications/x3dh/ (besucht am 27. 03. 2025).

[25] Dominik Grolimund u. a. „Cryptree: A Folder Tree Structure for Cryptographic File Systems". In: *2006 25th IEEE Symposium on Reliable Distributed Systems (SRDS'06)*. 2006, S. 189–198. DOI: 10.1109/SRDS.2006.15.

[26] Proton. *Proton Drive's Security Model: How We Protect Your Files*. Aug. 2024. URL: https://proton.me/blog/protondrive-security (besucht am 22. 02. 2025).

[27] WhatsApp. *Chatverlauf sichern*. 2024. URL: https://faq.whatsapp.com/481135090640375/?locale=de_DE&cms_platform=android&category=5245251 (besucht am 27. 03. 2025).

[28] Threema GmbH. *Threema Cryptography Whitepaper*. 2024. URL: https://threema.ch/press-files/2_documentation/cryptography_whitepaper.pdf (besucht am 27. 03. 2025).

[29] Signal Support. *Nachrichten sichern und wiederherstellen*. 2024. URL: https://support.signal.org/hc/de/articles/360007059752-Nachrichten-sichern-und-wiederherstellen (besucht am 27. 03. 2025).

[30] Patrick Beuth u. a. *Datenleck beim Volkswagen-Konzern: Wir wissen, wo dein Auto steht*. Aus DER SPIEGEL 1/2025. 2024. URL: https://www.spiegel.de/netzwelt/web/volkswagen-konzern-datenleck-wir-wissen-wo-dein-auto-steht-a-e12d33d0-97bc-493c-96d1-aa5892861027 (besucht am 19. 04. 2025).

[31] Rolf Haenni u. a. *CHVote Protocol Specification*. Cryptology ePrint Archive, Paper 2017/325. 2017. URL: https://eprint.iacr.org/2017/325.

[32] Federal Office for Information Security (BSI). *BSI TR-02102-1: Cryptographic Mechanisms: Recommendations and Key Lengths*. Techn. Ber. Bundesamt für Sicherheit in der Informationstechnik, 2025. URL: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf.

[33] Reto König. „BTI4201 Applied Cryptography 1 – Part 05: Authenticated Encryption with Associated Data (Sozi-Präsentation)". Vorlesungsfolien, Berner Fachhochschule – Technik und Informatik, FS 2024. 2024.

[34] Rolf Hänni und Philipp Locher. „BTI4202 Applied Cryptography 2 – Topic 1: Introduction to Public-Key Cryptography". Vorlesungsfolien, Berner Fachhochschule – Technik und Informatik, FS 2024. 2024.

[35] Rolf Hänni und Philipp Locher. „BTI4202 Applied Cryptography 2 – Topic 5: RSA and DSA Digital Signatures“. Vorlesungsfolien, Berner Fachhochschule – Technik und Informatik, FS 2024. 2024.

[36] Burt Kaliski. *PKCS #5: Password-Based Cryptography Specification Version 2.0*. RFC 2898. Sep. 2000. DOI: 10.17487/RFC2898. URL: https://www.rfc-editor.org/info/rfc2898.

[37] Hugo Krawczyk. *Cryptographic Extraction and Key Derivation: The HKDF Scheme*. Cryptology ePrint Archive, Paper 2010/264. 2010. URL: https://eprint.iacr.org/2010/264.

[38] Python Cryptographic Authority. *Why use cryptography?* Siehe: https://cryptography.io/en/latest/faq/. Zugegriffen am 08.06.2025. 2024.

[39] Electronic Frontier Foundation. *Certbot Documentation*. https://eff-certbot.readthedocs.io/en/stable/index.html. Zugegriffen am 08.06.2025. 2024.

[40] The Paramiko Developers. *Paramiko – Python implementation of SSHv2*. https://github.com/paramiko/paramiko/tree/main?tab=readme-ov-file. Zugegriffen am 08.06.2025. 2024.

[41] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: Galois/-Counter Mode (GCM) and GMAC*. Techn. Ber. NIST Special Publication 800-38D. National Institute of Standards und Technology (NIST), 2007. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf.

[42] Kathleen Moriarty, Burt Kaliski und Andreas Rusch. *PKCS #5: Password-Based Cryptography Specification Version 2.1*. RFC 8018. Jan. 2017. DOI: 10.17487/RFC8018. URL: https://www.rfc-editor.org/info/rfc8018.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Glossar

Accountability Prinzip der Rechenschaftspflicht, das sicherstellt, dass Akteure für ihr Handeln zur Verantwortung gezogen werden können.

associated data space Menge aller möglichen *assoziierten Daten* ($\mathcal{A}$), die bei einem *AEAD*-Verfahren authentifiziert, aber nicht verschlüsselt werden.

at-rest-Verschlüsselung Bezeichnet die Verschlüsselung von Daten im Ruhezustand, d. h. bei Speicherung auf einem Medium wie einer Festplatte oder in einer Datenbank.

Authenticity Ein kryptographisches Sicherheitsziel, das die Echtheit und Glaubwürdigkeit von Daten oder Identitäten sicherstellt, indem es deren Herkunft von der behaupteten Quelle und ihre Unverfälschtheit bestätigt.

Bootstrapping Im Kontext kryptographischer Systeme bezeichnet Bootstrapping den Prozess der initialen Einrichtung und des sicheren Austauschs von kryptographischem Material (z.B. Schlüssel, Zertifikate) zwischen Parteien, die zuvor keine gesicherte Beziehung hatten. Ziel ist es, eine vertrauenswürdige Basis für die nachfolgende sichere Kommunikation zu etablieren. In dieser Arbeit wird ein Bootstrapping-Verfahren über einen authentischen Papierkanal verwendet.

Brute-force resistance Sicherheitseigenschaft einer *Key Derivation Function*, siehe Abschnitt 5.1.7.

Brute-Force-Angriff Angriffsmethode, bei der alle möglichen Schlüssel, Passwörter oder Eingabewerte systematisch ausprobiert werden, um ein kryptographisches Verfahren zu brechen. Die Sicherheit gegenüber Brute-Force-Angriffen hängt massgeblich von der Entropie des geheimen Werts und der Effizienz der eingesetzten Schutzmechanismen wie *KDF* ab.

Ciphertext Bezeichnet die verschlüsselten Daten, die aus der Anwendung eines kryptographischen Verfahrens auf einen *Plaintext* resultieren. Der *Ciphertext* sollte ohne Kenntnis des Schlüssels nicht interpretierbar sein.

ciphertext space Menge aller möglichen *Ciphertext*s (*ciphertext space*), die durch Anwendung eines Verschlüsselungsverfahrens auf Nachrichten aus dem *plaintext space* entstehen können.

Collision-resistance Sicherheitseigenschaft kryptographischer Hashfunktionen, siehe Abschnitt 5.1.5.

Confidentiality Ein fundamental-kryptographisches Sicherheitsziel, das sicherstellt, dass Informationen nur von autorisierten Personen oder Systemen eingesehen oder offengelegt werden können. Vertraulichkeit wird typischerweise durch Verschlüsselungsverfahren erreicht, die Daten für Unbefugte unlesbar machen und somit den unberechtigten Zugriff auf sensitive Inhalte verhindern.

context space Menge aller möglichen Kontextinformationen (*context space*), die in *Key Derivation Function* genutzt werden können, um abgeleitete Schlüssel an bestimmte Anwendungsfälle zu binden.

Cryptree Ein vorgeschlagenes System zur sicheren Speicherung hierarchischer Dateisysteme auf nicht vertrauenswürdigen Servern. Es verschlüsselt Dateiinhalte sowie Metadaten mittels einer kryptographischen Schlüsselhierarchie und verwehrt so dem Serverbetreibenden den Einblick [25].

data space Menge aller möglichen Daten (*data space*), die als Eingabe eines digitalen Signaturschemas verwendet werden können.

Diffie-Hellman Kryptographisches Schlüsselaustauschverfahren, das es zwei Parteien ermöglicht, über einen unsicheren Kanal ein gemeinsames Geheimnis zu erzeugen. Das Verfahren basiert auf dem *Discrete Logarithm* (*DL*)-Problem und ist die Grundlage vieler Protokolle mit *Forward Secrecy*. Es existieren sowohl klassische als auch elliptische Varianten (ECDH).

digest Ausgabe einer kryptographischen Hashfunktion. Der Digest ist ein Bitstring fester Länge, der als komprimierte Repräsentation einer beliebig langen Eingabe dient

digest space Menge aller möglichen Ausgaben (*digest space*) einer kryptographischen Hashfunktion.

Domain separation Sicherheitseigenschaft einer *Key Derivation Function*, siehe Abschnitt 5.1.7.

ElGamal Ein asymmetrisches Kryptosystem, das auf dem *DL*-Problem basiert. Es unterstützt sowohl Verschlüsselung als auch digitale Signaturen und ist die Grundlage für viele moderne kryptographische Verfahren.

Entropy preservation Sicherheitseigenschaft einer *Key Derivation Function*, siehe Abschnitt 5.1.7.

Forward Secrecy Sicherheitsmerkmal eines kryptographischen Protokolls, das gewährleistet, dass die Kompromittierung langfristiger Schlüssel keine Rückschlüsse auf frühere Sitzungen oder deren Inhalte erlaubt. Dies wird typischer-

weise durch die Verwendung kurzlebiger Sitzungsschlüssel und flüchtiger Ephemeral-Schlüssel erreicht, wie z. B. durch *Diffie-Hellman*-basierte Verfahren.

GitLab Webbasierte Plattform zur Versionsverwaltung und Zusammenarbeit an Softwareprojekten.

hash input space Menge aller möglichen Eingaben (*hash input space*) einer kryptographischen Hashfunktion.

Integrity Ein fundamental-kryptographisches Sicherheitsziel, das die Korrektheit, Konsistenz und Vollständigkeit von Informationen und Verarbeitungsmethoden über ihren gesamten Lebenszyklus gewährleistet. Integrität stellt sicher, dass Daten während der Speicherung, Verarbeitung oder Übertragung nicht unautorisiert, unbemerkt oder versehentlich modifiziert, korrumpiert oder zerstört wurden. Mechanismen wie kryptographische Hashfunktionen, *Message Authentication Codes (MACs)* und digitale Signaturen dienen der Wahrung der Datenintegrität.

KDF input space Menge aller möglichen Eingaben (*KDF input space*) einer *Key Derivation Function*, typischerweise geheimes Schlüsselmaterial variabler Länge.

KDF output space Menge aller möglichen abgeleiteten Schlüsselmaterialien (*KDF output space*), die als Ausgabe einer *Key Derivation Function* entstehen können.

key space Schlüsselraum (engl. *key space*); Menge aller möglichen kryptographischen Schlüssel, die in einem bestimmten Verfahren verwendet werden können.

Kryptographische Hashfunktion Bildet eine Eingabe beliebiger Länge auf eine Ausgabe fixer Länge ab. In dieser Arbeit werden nur Hashfunktionen verwendet, die mindestens *Collision-resistance* und *Preimage-resistance* gewährleisten.

Man-in-the-Middle Angriffsform, bei der eine dritte Partei die Kommunikation zwischen zwei Teilnehmern unbemerkt mitliest oder manipuliert.

MongoDB Eine NoSQL-Datenbank, die Daten in Form von Dokumenten speichert, anstatt sie in Tabellen mit festen Strukturen abzulegen.

Nonce Einmalwert (engl. *number used once*), der typischerweise bei symmetrischer Verschlüsselung oder Authentifizierungsverfahren verwendet wird, um Wiederholungsangriffe zu verhindern. Er muss für jede Operation eindeutig, aber nicht zwingend geheim sein.

nonce space Menge aller zulässigen *Nonce* (*nonce space*) in einem kryptographischen Verfahren.

Passkey Asymmetrisches Schlüsselpaar zur passwortlosen Authentifizierung, bei dem der private Schlüssel gerätespezifisch gespeichert wird (z. B. in einem Secure Element). Der öffentliche Schlüssel wird bei einem Dienst registriert. Passkeys basieren auf Standards wie FIDO2/WebAuthn und ermöglichen benutzerfreundliche und sichere Logins.

Plaintext Bezeichnet die ursprünglichen, unverschlüsselten Daten vor der Anwendung eines kryptographischen Verfahrens. Der *Plaintext* ist typischerweise lesbar und schützenswert.

plaintext space Klartextraum (*plaintext space*); Menge aller möglichen Klartexte, die einem Verschlüsselungsverfahren als Eingabe übergeben werden können.

Post-Compromise Security Eigenschaft eines kryptographischen Protokolls, die es erlaubt, nach der Kompromittierung eines Sitzungsschlüssels durch fortgesetzte Kommunikation wieder einen sicheren Zustand zu erreichen. Die Sicherheit zukünftiger Nachrichten wird dabei wiederhergestellt, ohne dass ein vollständiger Neuaufbau des Schlüssels notwendig ist. *Post-Compromise Security* wird insbesondere durch Mechanismen wie dem *DR*-Protokoll ermöglicht.

Preimage-resistance Sicherheitseigenschaft kryptographischer Hashfunktionen, siehe Abschnitt 5.1.5.

Privacy Schutz der Privatsphäre und personenbezogener Daten vor unbefugtem Zugriff oder Missbrauch.

private key space Schlüsselraum (engl. *private key space*); Menge aller möglichen privaten Schlüssel, die in einem asymmetrischen Kryptosystem verwendet werden können.

Proton Drive Ein sicherer Cloud-Speicherdienst von Proton, der eine *E2EE* für Dateien und Ordner bietet.

public key space Schlüsselraum (engl. *public key space*); Menge aller möglichen öffentlichen Schlüssel, die in einem asymmetrischen Kryptosystem verwendet werden können.

pyca/cryptography Weit verbreitete, in Python geschriebene Kryptographiebibliothek, die sichere primitive Funktionen wie *AES-GCM*, *RSA*, *ECC*, Kryptographische Hashfunktion und digitale Signaturen über eine benutzerfreundliche API bereitstellt. Sie basiert auf OpenSSL und wird von der Python Cryptographic Authority (PyCA) gepflegt.

Repository Speicherort für Quellcode und Projektdateien, meist unter Versionskontrolle.

Salt Ein nicht geheimer, zufälliger Wert, der in kryptographischen Verfahren, insbesondere bei Passwort-Hashing und *Key Derivation Function*, verwendet wird.

salt space Menge aller möglichen *Salt*-Werte (*salt space*), die in *Key Derivation Function* oder Passwortableitungsverfahren verwendet werden, um Vorberechnungsangriffe zu erschweren.

SecDevOps Kurzform für *Security Development Operations*; ein Ansatz, der Sicherheitsaspekte integrativ in alle Phasen von Softwareentwicklung und IT-Betrieb einbindet. Dabei werden Prinzipien von *DevOps* um kontinuierliche Sicherheitspraktiken wie automatisierte Sicherheitsprüfungen, Schwachstellenmanagement und sichere Konfiguration erweitert.

Second-preimage-resistance Sicherheitseigenschaft kryptographischer Hashfunktionen, siehe Abschnitt 5.1.5.

Seed Ein geheimer Ausgangswert mit hoher Entropie, der als Eingabe für einen deterministischen kryptographischen Prozess dient. Aus dem *Seed* lassen sich weitere Werte wie Schlüssel oder pseudozufällige Zahlen ableiten. Die Kenntnis des *Seeds* würde die Reproduktion aller daraus abgeleiteten Werte erlauben, weshalb seine Geheimhaltung essentiell ist.

Session Key Ein temporärer, typischerweise symmetrischer Schlüssel, der für die Dauer einer einzelnen Kommunikationssitzung oder zur Verschlüsselung einer einzelnen Nachricht generiert und verwendet wird. Die Verwendung von Session Keys, die für jede Sitzung/Nachricht neu erzeugt werden, kann zur Erhöhung der Sicherheit beitragen, indem der potenzielle Schaden bei Kompromittierung eines Schlüssels begrenzt wird.

Sign-and-Encrypt (SaE) Ein kryptographisches Paradigma zur Kombination von digitaler Signatur und Verschlüsselung, bei dem der Klartext parallel und unabhängig sowohl signiert als auch verschlüsselt wird. Das finale Ergebnis besteht typischerweise aus dem Chiffretext und der separaten Signatur des ursprünglichen Klartextes. Dieses Vorgehen unterscheidet sich von Ansätzen wie *Sign-then-Encrypt*, bei dem die Signatur gemeinsam mit dem Klartext verschlüsselt wird.

signature space Signaturraum (*signature space*); Menge aller möglichen digitalen Signaturen, die durch Anwendung eines Signaturalgorithmus wie $\mathsf{sign}_{sk}(m)$ entstehen können.

signing key space Schlüsselraum (engl. *signing key space*); Menge aller möglichen privaten Signaturschlüssel, die zum Erzeugen digitaler Signaturen verwendet werden können.

Social Engineering Angriffsform, bei der menschliches Verhalten und zwischenmenschliche Manipulation ausgenutzt werden, um sicherheitsrelevante Informationen zu erhalten oder sicherheitskritische Handlungen zu bewirken.

Space Menge aller möglichen Werte für eine bestimmte Grösse in einem kryptographischen System, z. B. *key space*, *plaintext space* etc.

String Endliche Folge von Zeichen (z. B. $\in \Sigma^*$).

Tag Kurzform für Authentifizierungstag. Ein bei *AEAD*-Verfahren generierter Wert, der beim Entschlüsseln überprüft wird, um die *Integrity* und *Authenticity* der verschlüsselten Nachricht sowie der *AD* sicherzustellen.

tag space Menge aller möglichen Authentifizierungstags (*tag space*), die durch die Anwendung eines *AEAD*-Verfahrens entstehen können.

User Oberbegriff für sämtliche aktiven Plattformteilnehmer (*Admin*, *Caregiver*, *Student*, *Teacher*), die eine eigene Identität mit uid und Schlüsselmaterial besitzen. Der Begriff wird verwendet, wenn eine Aussage für alle Parteien gleichermassen gilt oder keine spezifische Rolle relevant ist.

User Story Kurze, meist in Alltagssprache formulierte Beschreibung einer Anforderung aus Sicht einer bestimmten Nutzerrolle. *User Stories* werden häufig in agilen Entwicklungsmethoden wie Scrum verwendet und bestehen typischerweise aus einer Rollenbeschreibung, einem Ziel und einem Nutzen.

verification key space Schlüsselraum (engl. *verification key space*); Menge aller möglichen öffentlichen Verifikationsschlüssel, die zur Überprüfung digitaler Signaturen verwendet werden können.

Weakest Link Bezeichnet das schwächste Glied in einer sicherheitskritischen Kette. In kryptographischen Systemen bestimmt die am wenigsten sichere Komponente – etwa ein schwaches Passwort, eine unsichere Implementierung oder fehlerhafte Schlüsselspeicherung – die effektive Gesamtsicherheit. Das Konzept dient als Warnung, dass Sicherheitsanalysen ganzheitlich erfolgen müssen.

Wörterbuchangriff Angriffsform, bei der ein Angreifer vorab eine Liste möglicher Passwörter (Wörterbuch) erstellt und diese systematisch ausprobiert, um ein Passwort oder daraus abgeleitete Schlüssel zu erraten.

X3DH *Extended Triple Diffie-Hellman* ist ein Schlüsselaustauschprotokoll, das es zwei Kommunikationspartnern erlaubt, auch bei asynchroner Kommunikation (z. B. wenn einer der Partner offline ist) ein gemeinsames Geheimnis sicher zu erzeugen. Es verwendet eine Kombination aus drei *Diffie-Hellman*-Berechnungen mit statischen und flüchtigen Schlüsseln und wird unter anderem im Signal-Protokoll eingesetzt. *X3DH* gewährleistet *Forward Secrecy* und *Authenticity*.

Symbolverzeichnis

AB *Absences* Datei, siehe Abschnitt 5.1.8.

$\mathcal{A}$ Menge aller möglichen assoziierten Daten (Associated Data Space).

$cfsk$ Verschlüsselter fsk.

CG *Caregivers* Datei, siehe Abschnitt 5.1.8.

$\mathcal{C}$ Menge aller möglichen Chiffrate (Ciphertext Space).

$cotp$ Asymmetrisch verschlüsseltes otp.

$COTPS$ *COTPS* Datei, siehe Abschnitt 5.1.8.

$\mathcal{D}$ Menge aller möglichen Daten, die als Eingabe für digitale Signaturschemata verwendet werden können (Data Space).

dsk *Device Secret Key*, ist ein symmetrischer Schlüssel, der spezifisch für ein Gerät generiert wird. Er dient zur sicheren Speicherung oder Kommunikation auf dem jeweiligen Gerät.

fsk *File Secret Key*, ist ein symmetrischer Schlüssel, der zur Verschlüsselung von Dateiinhalten verwendet wird. Er wird jeweils pro Datei erzeugt und ist nur für berechtigte *User* zugänglich.

$\mathcal{H}_{in}$ Menge aller möglichen Eingaben einer kryptographischen Hashfunktion (Hash Input Space).

$\mathcal{H}_{out}$ Menge aller möglichen Ausgaben (Digest) einer kryptographischen Hashfunktion (Digest Space).

$HOTPS$ *HOTPS* Datei, siehe Abschnitt 5.1.8.

ILC *Invitation Letter Caregviver*, siehe Abschnitt 5.1.8.

ILT *Invitation Letter Teacher*, siehe Abschnitt 5.1.8.

$\mathcal{I}$ Menge aller möglichen Kontextinformationen für eine KDF (Context Space).

$\mathcal{K}_{out}$ Menge aller möglichen Ausgaben einer Key Derivation Function (KDF Output Space).

$\mathcal{K}$ Menge aller möglichen kryptographischen Schlüssel (Key Space).

$\mathcal{K}_{prk}$ Menge aller möglichen privaten Schlüssel (Private Key Space).

$\mathcal{K}_{PUK}$ Menge aller möglichen öffentlichen Schlüssel (Public Key Space).

$\mathcal{K}_{sik}$ Menge aller möglichen privaten Signaturschlüssel (Signing Key Space).

$\mathcal{K}_{VK}$ Menge aller möglichen öffentlichen Verifikationsschlüssel (Verification Key Space).

$KRPC$ Datenstruktur bestehend aus $cfsk$ und verschlüsseltem RPC.

$KRPT$ Datenstruktur bestehend aus $cfsk$ und verschlüsseltem RPT.

KS *Keystore*, siehe Abschnitt 5.1.8.

$\mathcal{KS}$ Raum aller zulässigen Informationen, die im KS abgelegt werden können (KS-Space).

λ Sicherheitsparameter, typischerweise die Bitlänge der verwendeten Schlüssel, der die Sicherheitsstufe eines kryptographischen Verfahrens bestimmt.

MF *Message* Datei, siehe Abschnitt 5.1.8.

mid *Message Identifier*

$\mathcal{M}$ Menge aller zulässigen mids (mid-Space).

msk *Message Secret Key*, ist ein symmetrischer Schlüssel, der zur Verschlüsselung von Nachrichteninhalten verwendet wird. Er wird jeweils pro Nachricht erzeugt und ist nur für berechtigte *User* zugänglich.

$\mathcal{N}$ Menge aller zulässigen Nonces (Nonce Space).

otp *One Time Password*, siehe auch Abschnitt 5.1.6.

$\mathcal{O}$ Menge aller zulässigen otps (otp-Space).

$PERF$ *Permissions* Datei, siehe Abschnitt 5.1.8.

$\perp$ Spezialwert, der die Ausgabe eines fehlgeschlagenen Verifikations- oder Entschlüsselungsvorgangs kennzeichnet (z. B. bei ungültigem *Tag*).

PF *Userprofile*, siehe Abschnitt 5.1.8.

PIL *Physical / Printed Invitation Letter*, bezeichnet das physische oder gedruckte Einladungsschreiben für den *Caregiver* oder den *Teacher*, auf dem der ILC oder ILT als QR-Code abgebildet ist.

$\mathcal{P}$ Menge aller möglichen Klartexte (Plaintext Space).

prk *Private Key*, bezeichnet den geheimen, nur dem Schlüsselinhaber bekannten Teil eines asymmetrischen Schlüsselpaares. Er dient zur Entschlüsselung und muss besonders geschützt werden.

$prk_i,\ i \in \{a, t, c\}$ *Private Key* eines *Users* i zum Entschlüsseln von Informationen.

PUK *Public Key*, bezeichnet den öffentlichen Schlüssel in einem asymmetrischen Kryptosystem. Er kann frei verteilt werden und wird beispielsweise zur Verschlüsselung von Nachrichten verwendet.

PUK_a *Public Key Admin* zum Verschlüsseln von Informationen.

PUK_c *Public Key Caregiver* zum Verschlüsseln von Informationen.

$PUK_i,\ i \in \{a, t, c\}$ *Public Key* eines *Users* i zum Verschlüsseln von Informationen.

$PUKS$ *Publickeys* Datei, siehe Abschnitt 5.1.8.

PUK_t *Public Key Teacher* zum Verschlüsseln von Informationen.

RPC *Registrationpackage Caregiver*, siehe Abschnitt 5.1.8.

RPT *Registrationpackage Teacher*, siehe Abschnitt 5.1.8.

$\mathcal{R}$ Menge aller möglichen Salt-Werte (Salt Space).

sig *Signature*, Abkürzung für eine kryptographische Signatur.

$sigf$ *Signature* Datei, siehe Abschnitt 5.1.8.

$\mathcal{S}$ Menge aller möglichen digitalen Signaturen (Signature Space).

sik *Signing Key*, bezeichnet den geheimen Schlüssel, der zur Erzeugung digitaler Signaturen verwendet wird. In einem asymmetrischen Signatursystem bildet er zusammen mit dem öffentlichen VK ein Schlüsselpaar.

$sik_i,\ i \in \{a, t, c\}$ *Signing Key User* i.

sk *Secret Key*, bezeichnet einen geheimen Schlüssel, der sowohl zur Ver- als auch zur Entschlüsselung verwendet wird. *Secret Key* kommen in symmetrischen Verschlüsselungsverfahren zum Einsatz und müssen vertraulich zwischen den Kommunikationspartnern geteilt werden.

$skid$ *Secret Key Identifier*, identifiziert denjenigen symmetrischen Schlüssel (sk), mit dem die privaten Schlüssel prk und sik eines *Users* verschlüsselt wurden.

ST *Students* Datei, siehe Abschnitt 5.1.8.

str Endliche Folge von Zeichen (z. B. $\in \Sigma^\star$).

TA *Teachers* Datei, siehe Abschnitt 5.1.8.

$\mathcal{T}$ Menge aller möglichen Authentifizierungstags (Tag Space).

$TOTPS$ *TOTPS* Datei, siehe Abschnitt 5.1.8.

uid *User Identifier*

uid_a *User Identifier Admin*

uid_c *User Identifier Caregiver*

$uid_i,\ i \in \{a, t, c, s\}$ *User Identifier User* i

uid_s *User Identifier Student*

$\mathcal{U}$ Menge aller zulässigen uids (uid-Space).

uid_t *User Identifier Teacher*

us *User Secret*, ist ein nur dem *User* bekanntes Geheimnis, aus dem kryptographisches Schlüsselmaterial abgeleitet wird. Es bildet die Grundlage zur Erzeugung des usk.

usk *User Secret Key*, ist ein symmetrischer Schlüssel, der direkt aus dem us abgeleitet wird. Er dient zur Entschlüsselung sensibler Daten, wie beispielsweise des privaten Master-Schlüssels.

VK *Verification Key*, ist der öffentliche Schlüssel zur Überprüfung digitaler Signaturen. Er wird im Rahmen eines asymmetrischen Signaturverfahrens zusammen mit dem privaten sik verwendet.

VK_a *Verification Key Admin*.

VK_c *Verification Key Caregiver*.

$VK_i,\ i \in \{a, t, c\}$ *Verification Key User* i.

VK_t *Verification Key Teacher*.

$\mathcal{X}$ Menge aller möglichen Eingaben einer Key Derivation Function (KDF Input Space).

Abkürzungsverzeichnis

AD *Associated Data* (AD) bezeichnet zusätzliche Daten, die bei einem *AEAD*-Verfahren mitauthentifiziert, aber nicht verschlüsselt werden. Typische Beispiele sind Header-Informationen oder Metadaten, deren *Integrity* und *Authenticity* gewährleistet, deren Inhalt jedoch im Klartext belassen wird.

AE *Authenticated Encryption* (AE) bezeichnet ein Verschlüsselungsverfahren, das sowohl die *Confidentiality* als auch die *Authenticity* und *Integrity* der verschlüsselten Daten sicherstellt. AE ist ein übergeordneter Begriff, unter den auch spezifische Verfahren wie AEAD fallen.

AEAD *Authenticated Encryption with Associated Data* (AEAD) bezeichnet ein Verschlüsselungsverfahren, das gleichzeitig *Confidentiality*, *Integrity* und *Authenticity* gewährleistet. Neben den zu verschlüsselnden Daten können zusätzlich nichtvertrauliche, aber authentifizierte Daten (Associated Data) eingebunden werden.

AES-GCM *Advanced Encryption Standard - Galois/Counter Mode* (AES-GCM) ist ein symmetrisches Verschlüsselungsverfahren, das *Authenticity* und *Confidentiality* in einem Schritt kombiniert. Es wird unter anderem in TLS und verschlüsseltem Dateispeicher eingesetzt.

DDH *Decisional Diffie-Hellman Assumption* — Die Annahme, dass es schwierig ist, zwischen einem echten Diffie-Hellman-Triple (g^a, g^b, g^{ab}) und einem zufälligen Triple (g^a, g^b, g^c) zu unterscheiden. Diese Annahme bildet die Grundlage für die Sicherheit vieler kryptographischer Protokolle, insbesondere im Bereich der Public-Key-Verschlüsselung.

DL *Discrete Logarithm* — Bezeichnet den Exponenten x, der zu einer gegebenen Basis g und einem Gruppenelement h die Gleichung $g^x = h$ erfüllt. Die Berechnung diskreter Logarithmen gilt in geeigneten Gruppen als schwer und bildet die Grundlage vieler kryptographischer Verfahren.

DR *Double Ratchet* ist ein kryptographisches Protokoll zur Schlüsselableitung, das für Ende-zu-Ende-verschlüsselte Kommunikation entwickelt wurde. Es kombiniert eine symmetrische Ketten-Ratchet mit einer asymmetrischen Diffie-Hellman-Ratchet und gewährleistet damit Forward und Post-Compromise Security. Double Ratchet wird unter anderem im Signal-Protokoll und verwandten Systemen wie *E2EE*-Messengern eingesetzt.

DSG Schweizer Bundesgesetz über den Datenschutz

DSGVO Europäische Datenschutz-Grundverordnung

E2EE *End-to-End Encryption* (E2EE) bezeichnet eine Kommunikationsform, bei der nur die kommunizierenden Endpunkte (Sender und Empfänger) Zugriff auf die Klartextdaten haben. Die Daten sind während der gesamten Übertragung verschlüsselt, sodass kein zwischengeschalteter Server auf die Inhalte zugreifen kann. Weitere Details finden sich in Kapitel 2.1.2.

ECC *Elliptic Curve Cryptography* (ECC) ist ein asymmetrisches Kryptosystem, das auf der Mathematik elliptischer Kurven basiert. Es bietet bei gleicher Sicherheit kleinere Schlüsselgrössen als RSA und wird unter anderem in TLS, digitalen Signaturen und Blockchain-Systemen verwendet.

ECIES *Elliptic Curve Integrated Encryption Scheme*, bombiniertes asymmetrisches Verschlüsselungsschema auf Basis elliptischer Kurven. ECIES bietet keine direkte asymmetrische Verschlüsselung von Nachrichten, sondern verwendet hybrides Verschlüsseln: ein symmetrischer Schlüssel wird erzeugt und mit einer Elliptic-Curve-Schlüsselvereinbarung (z. B. ECDH) gesichert. Die eigentliche Nachricht wird anschliessend symmetrisch verschlüsselt (z. B. mit AES-GCM).

EUF-CMA *Existential Unforgeability under Chosen Message Attacks* (EUF-CMA) ist ein Sicherheitsmodell für digitale Signaturen. Es garantiert, dass ein Angreifer keine gültige Signatur für eine neue Nachricht erzeugen kann, selbst wenn er Signaturen zu beliebigen anderen Nachrichten erhalten hat.

HMAC *Hash-based Message Authentication Code* ist ein Verfahren zur Integritätsprüfung und Authentifizierung von Nachrichten mithilfe eines geheimen Schlüssels und einer Hashfunktion.

HMAC-SHA256 *HMAC-SHA256* kombiniert den *Hash-based Message Authentication Code* (*HMAC*)-Mechanismus mit der Hashfunktion *SHA-256*, um eine manipulationssichere Prüfsumme zu erzeugen. Sie wird häufig zur Integritätsprüfung und Authentifizierung in kryptographischen Protokollen eingesetzt.

IND-CCA2 *Indistinguishability under Adaptive Chosen Ciphertext Attack* (IND-CCA2) ist ein starkes Sicherheitsmodell für Verschlüsselungsverfahren. Es garantiert, dass ein Angreifer, selbst mit Zugang zu einem Entschlüsselungsorakel, keine Informationen über den Klartext eines Zielciphertexts gewinnen kann.

IND-CPA *Indistinguishability under Chosen Plaintext Attack* (IND-CPA) ist ein Sicherheitsmodell für Verschlüsselungsverfahren. Es garantiert, dass ein Angreifer, selbst wenn er die Möglichkeit hat, beliebige Klartexte verschlüsseln zu lassen, nicht zwischen zwei ausgewählten Klartexten unterscheiden kann, basie-

rend auf deren Chiffraten. IND-CPA gilt als grundlegendes Sicherheitsniveau für symmetrische und asymmetrische Verschlüsselung.

KDF *Key Derivation Function* (KDF) ist eine kryptographische Funktion zur Ableitung sicherer Schlüssel aus einem Ausgangsgeheimnis. Typische Inputs sind Passwörter, gemeinsame Geheimnisse oder Zufallswerte. Eine KDF verhindert Rückschlüsse auf das Ursprungsmaterial und ermöglicht die Verwaltung mehrerer Schlüssel aus einem einzigen Secret.

OAEP *Optimal Asymmetric Encryption Padding* (OAEP) ist ein Padding-Schema für *RSA*, das Schutz gegen Chosen-Ciphertext-Angriffe bietet und häufig für sichere asymmetrische Verschlüsselung verwendet wird.

PbD *Privacy by Design*, siehe Abschnitt 2.1.1.

PBKDF *Password-Based Key Derivation Function* (PBKDF) ist eine spezialisierte Form einer Key Derivation Function (KDF), die aus einem Passwort unter Verwendung eines Salts und Key-Stretching-Techniken einen kryptographisch starken Schlüssel ableitet.

PBKDF2 *Password-Based Key Derivation Function 2* ist eine standardisierte Version von *PBKDF*, definiert in PKCS#5 v2.1 (RFC 8018) [42]. Sie verwendet *HMAC* mit einer Hashfunktion (z. B. *SHA-256*), ein Salt und eine Iterationsanzahl zur sicheren Ableitung von Schlüsseln aus Passwörtern.

PoC Ein *Proof of Concept* (PoC) ist eine prototypische Umsetzung zur Überprüfung, ob bestimmte Konzepte, technische Verfahren oder Systemkomponenten in der Praxis funktionieren. Im Kontext dieser Arbeit dient der PoC dazu, die Funktionsfähigkeit der spezifizierten kryptographischen Mechanismen und Protokollschritte mit realen Algorithmen und Parametern zu demonstrieren.

PRNG *Pseudorandom Number Generator* (PRNG) ist ein deterministischer Algorithmus zur Erzeugung pseudozufälliger Werte aus einem geheimen Seed. Ein kryptographisch sicherer PRNG wird für Operationen wie die Erzeugung von Nonces, Zufallszahlen und temporären Schlüsseln verwendet.

PSS *Probabilistic Signature Scheme* (PSS) ist ein Padding-Verfahren für *RSA*-Signaturen, das probabilistische Elemente verwendet und als sicherere Alternative zu älteren Standards wie PKCS#1 v1.5 gilt.

RSA *Rivest-Shamir-Adleman* (RSA) ist ein asymmetrisches Kryptosystem, das auf der Faktorisierung grosser Zahlen basiert. Es wird häufig für digitale Signaturen und den sicheren Austausch symmetrischer Schlüssel eingesetzt.

RSA-OAEP *RSA Optimal Asymmetric Encryption Padding*, Kombination aus *RSA* und *OAEP*.

RSA-PSS *RSA Probabilistic Signature Scheme*, Kombination aus *RSA* und *Probabilistic Signature Scheme* (*PSS*).

S/MIME *Secure/Multipurpose Internet Mail Extensions* (S/MIME) ist ein Standard für die Ende-zu-Ende-Verschlüsselung und Signierung von E-Mails. Er basiert auf X.509-Zertifikaten und ermöglicht die Authentifizierung des Absenders sowie die *Confidentiality* und *Integrity* des Inhalts. S/MIME wird häufig in Unternehmensumgebungen eingesetzt und ist in vielen E-Mail-Clients nativ integriert.

SHA-2 *Secure Hash Algorithm 2*, kryptographische Hashfamilie, bestehend aus mehreren Varianten wie SHA-224, SHA-256, SHA-384 und SHA-512. SHA-2 gilt als weit verbreitet, sicher und ist die Basis vieler moderner kryptographischer Verfahren.

SHA-256 *Secure Hash Algorithm 256* (SHA-256) ist ein kryptographischer Hash-Algorithmus aus der SHA-2-Familie. Er erzeugt aus beliebigen Eingabedaten einen 256-Bit-Hashwert. SHA-256 wird häufig für Integritätsprüfungen, digitale Signaturen, Blockchain-Anwendungen und Passwort-Hashing eingesetzt.

SHA-3 *Secure Hash Algorithm 3* ist eine Familie kryptographischer Hashfunktionen, basierend auf dem Keccak-Algorithmus.

TLS *Transport Layer Security* (TLS) ist ein kryptographisches Protokoll, das die Sicherheit der Kommunikation über Netzwerke gewährleistet. Es bietet Authentication, Data *Integrity* und *Confidentiality* und wird unter anderem in HTTPS, E-Mail-Verschlüsselung und VPNs eingesetzt.

A. Methodische Ergänzungen

A.0.1. Zeitplan

Der Zeitplan (vgl. Abb.A.2, A.3, A.4, A.5, A.6, A.7, A.8, A.9, A.10) strukturierte das Projekt entlang definierter Meilensteine (vgl. Abb.A.1). Er diente der Sicherstellung eines zeitgerechten Fortschritts und bildete die Grundlage für die Terminierung der agilen Iterationen. Wie ersichtlich ist, kam es in gewissen Phasen, insbesondere in Phase 4 Detaillierte Spezifikation der gewählten Variante und Phase 5 Proof of Concept, zu erheblichen Abweichungen zwischen dem SOLL- und dem IST-Zustand.

Diese Abweichungen ergaben sich hauptsächlich daraus, dass insbesondere die Entwicklung des Angreifermodells deutlich mehr Zeit in Anspruch nahm als ursprünglich geplant. Zudem stellte sich das Schreiben der Spezifikation mit sämtlichen Protokollschritten, Pseudo-Codes etc. als zeitintensiver heraus als angenommen. Dies führte letztlich dazu, dass sich die Umsetzung des *PoC* verzögerte.

Wir betrachten diese Erfahrungen als eine lehrreiche Erkenntnis für zukünftige Projekte ähnlicher Art.

A.0.2. Risikomanagement

Im Rahmen des Risikomanagements wurden potenzielle Projektrisiken, siehe Abb. A.11, identifiziert, bewertet und mit geeigneten Massnahmen hinterlegt. Die Risikoanalyse unterstützt die Qualitätssicherung und Projektstabilität.

Meilensteine

Nr.	Titel	Beschreibung
K	Kick-Off mit Betreuer	
1	Abschluss Phase 1	Abgestimmter Projektplan, Dokumentierte Ausgangslage, Erste Risikoliste
2	Abschluss Phase 2	Zwei grobe Varianten (A/B) mit Vor- und Nachteilen, Katalog von Bewertungskriterien
3	Abschluss Phase 3	Entscheidung für eine Variante (z. B. Variante A), Dokumentation der Entscheidungsgründe
4	Abschluss Phase 4	Vollständige Spezifikation (inkl. technischer Doku, UML oder Diagramme), Freigabe für PoC
5	Abschluss Phase 5	Lauffähiger PoC-Teil mit dokumentierter Implementierung, Erste Tests (Unit-/Integrationstests)
6	Abschluss Phase 6	Abnahme PoC, Zusammenfassung Testergebnisse, Überarbeitungen bei Bedarf

Abbildung A.1.: AMeilensteine des Projekts.

A.0.3. User Stories

Die User Stories dienten während des Projekts der strukturierten Aufgabenverteilung. Dies förderte eine effiziente, asynchrone Zusammenarbeit und unterstützte das iterative Vorgehensmodell. Ein Ausschnitt während des Projekts vom Story Board ist in Abb. A.12 ersichtlich.

Bachelorarbeit Zeitplan

Semester Woche				0							1		
Meilensteine				K									
Spezielles													
Datum	10.2	11.2	12.2	13.2	14.2	15.2	16.2	17.2	18.2	19.2	20.2	21.2	22.2	23.2
Tag	Mo	Di	Mi	Do	Fr	Sa	So	Mo	Di	Mi	Do	Fr	Sa	So
1 *Projekt-Kickoff und Vorbereitung*								SOLL						
								IST						
2 *Grobe Konzeption & Vergleich zweier Varianten*														
3 Variantenentscheidung														
4 *Detaillierte Spezifikation der gewählten Variante*														
5 *Proof of Concept*														
6 *Testing & Validierung*														
7 *Abschluss & Dokumentation*														

Abbildung A.2.: Zeitplan: Woche 0 und 1

Bachelorarbeit Zeitplan

Semester Woche				2							3		
Meilensteine						1							
Spezielles													
Datum	24.2	25.2	26.2	27.2	28.2	1.3	2.3	3.3	4.3	5.3	6.3	7.3	8.3	9.3
Tag	Mo	Di	Mi	Do	Fr	Sa	So	Mo	Di	Mi	Do	Fr	Sa	So
1 *Projekt-Kickoff und Vorbereitung*														
2 *Grobe Konzeption & Vergleich zweier Varianten*	SOLL													
	IST													
3 Variantenentscheidung														
4 *Detaillierte Spezifikation der gewählten Variante*														
5 *Proof of Concept*														
6 *Testing & Validierung*														
7 *Abschluss & Dokumentation*														

Abbildung A.3.: Zeitplan: Woche 2 und 3

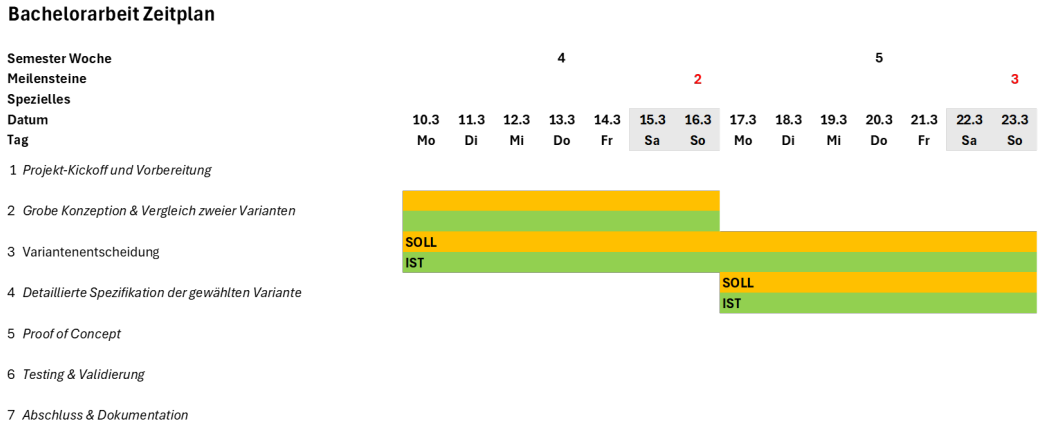


Abbildung A.4.: Zeitplan: Woche 4 und 5

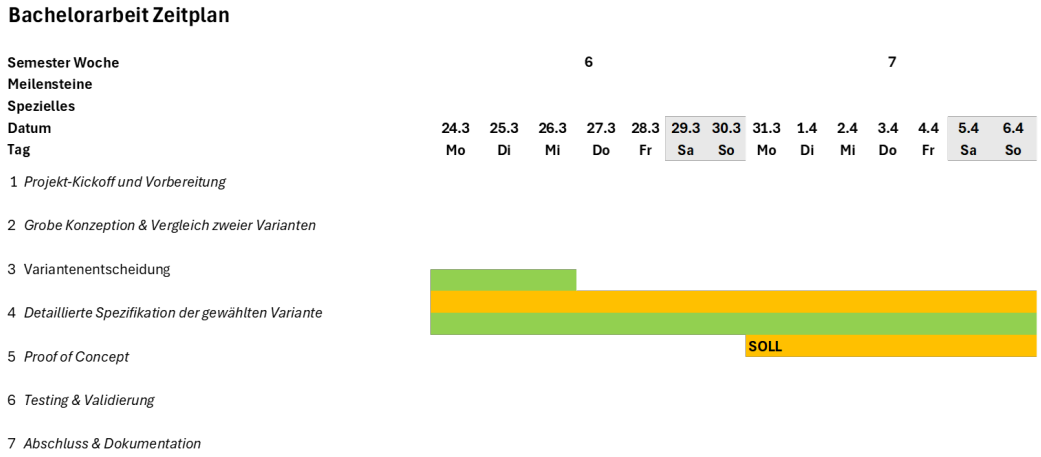


Abbildung A.5.: Zeitplan: Woche 6 und 7

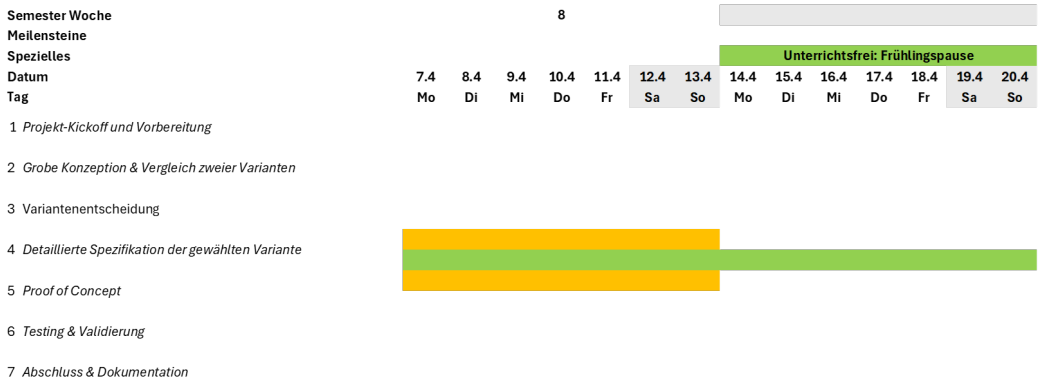


Abbildung A.6.: Zeitplan: Woche 8 und Semesterferien Woche

Bachelorarbeit Zeitplan

Semester Woche				9							10		
Meilensteine							4						
Spezielles													
Datum	21.4	22.4	23.4	24.4	25.4	26.4	27.4	28.4	29.4	30.4	1.5	2.5	3.5	4.5
Tag	Mo	Di	Mi	Do	Fr	Sa	So	Mo	Di	Mi	Do	Fr	Sa	So
1 *Projekt-Kickoff und Vorbereitung*														
2 *Grobe Konzeption & Vergleich zweier Varianten*														
3 Variantenentscheidung														
4 *Detaillierte Spezifikation der gewählten Variante*	SOLL													
5 *Proof of Concept*	SOLL													
6 *Testing & Validierung*	IST													
7 *Abschluss & Dokumentation*														

Abbildung A.7.: Zeitplan: Woche 9 und 10

Bachelorarbeit Zeitplan

Semester Woche				11							12		
Meilensteine													5
Spezielles													
Datum	5.5	6.5	7.5	8.5	9.5	10.5	11.5	12.5	13.5	14.5	15.5	16.5	17.5	18.5
Tag	Mo	Di	Mi	Do	Fr	Sa	So	Mo	Di	Mi	Do	Fr	Sa	So
1 *Projekt-Kickoff und Vorbereitung*														
2 *Grobe Konzeption & Vergleich zweier Varianten*														
3 Variantenentscheidung														
4 *Detaillierte Spezifikation der gewählten Variante*														
5 *Proof of Concept*														
6 *Testing & Validierung*	SOLL													
7 *Abschluss & Dokumentation*														

Abbildung A.8.: Zeitplan: Woche 11 und 12

Bachelorarbeit Zeitplan

Semester Woche				13							14		
Meilensteine								AP					6
Spezielles													
Datum	19.5	20.5	21.5	22.5	23.5	24.5	25.5	26.5	27.5	28.5	29.5	30.5	31.5	1.6
Tag	Mo	Di	Mi	Do	Fr	Sa	So	Mo	Di	Mi	Do	Fr	Sa	So
1 *Projekt-Kickoff und Vorbereitung*														
2 *Grobe Konzeption & Vergleich zweier Varianten*														
3 Variantenentscheidung														
4 *Detaillierte Spezifikation der gewählten Variante*														
5 *Proof of Concept*														
6 *Testing & Validierung*	IST													
7 *Abschluss & Dokumentation*						SOLL								
						IST								

Abbildung A.9.: Zeitplan: Woche 13 und 14

Bachelorarbeit Zeitplan

Semester Woche					15							16		
Meilensteine											A	P		
Spezielles								Letzte Semester Woche						
Datum	2.6	3.6	4.6	5.6	6.6	7.6	8.6	9.6	10.6	11.6	12.6	13.6	14.6	15.6
Tag	Mo	Di	Mi	Do	Fr	Sa	So	Mo	Di	Mi	Do	Fr	Sa	So
1 *Projekt-Kickoff und Vorbereitung*														
2 *Grobe Konzeption & Vergleich zweier Varianten*														
3 Variantenentscheidung														
4 *Detaillierte Spezifikation der gewählten Variante*														
5 *Proof of Concept*														
6 *Testing & Validierung*														
7 *Abschluss & Dokumentation*														

Abbildung A.10.: Zeitplan: Woche 15 und 16

Risikomanagement

* Gering, Mittel, Hoch

Risiko	Ursache	Auswirkung	Wahrscheinlich	Auswirkur	Bewertung	Vermeidung	Reaktive Massnahmen	Kommentar
Projektkoordination und Zeitmanagement	- Unterschätzung Dokumentations- und Implementierungsaufwand - Umfangreiches Thema für eine Bachelorarbeit - Parallel laufende andere Verpflichtungen	- Nicht erreichen von Meilensteinen - Qualitätseinbussen - Unvollständige Ergebnisse - Nicht erreichen der Ziele für die BA.	Mittel	Hoch	Mittel	- Realistische Planung mit Meilensteinen - Einplanung von Pufferzeiten - Regelmässige Status-Checks intern & mit Betreuer	- Fokus auf Kernfunktionalität - Falls nötig: Überstunden leisten, um kritische Funktionen abzuschliessen	Zeitmanagement ist kritisch für den Projekterfolg. Regelmässige Neubewertung der Zeitplanung nötig.
Unzureichende Abstimmung	- Unklare Aufgabenverteilung - Fehlende interne und externe Status-Updates	- Missverständnisse - Doppelte Arbeit - Spezifikationslücken	Gering	Hoch	Mittel	- Wöchentliche interne Meetings - Zweiwöchentliche Meetings mit Betreuer - Kanban Board - Klare Verantwortlichkeiten	Schnell Eskalationsgespräch mit Betreuer vereinbaren, wenn Blockaden auftretten.	Dokumentation von Meetings und Abstimmungen mit klaren Zielen und Deadlines.
Krankheit	Krankheit von Teammitgliedern während kritischer Projektphasen	- Verzögerung von Meilensteinen - Verzögerung von Projektfortschritt	Mittel	Mittel	Mittel	- Wissenstransfer zwischen Teammitgliedern - Vorarbeiten leisten, wenn möglich	- Aufgabenverteilung flexibel halten - Falls nötig, temporäre Umverteilung von Verantwortlichkeiten - Kürzung Implementierungsgrad PoC	
Datenverluste	- Hardware Probleme - Verlust Zugriff auf Cloud Dinste - Fehlbedienung oder versehentliches Löschen durch Teammitglieder	- Verlust von geleisteter Arbeit - Zeitverlust durch Versuch über Backup Arbeit wiederherzustellen	Gering	Hoch	Mittel	- Einsatz von GIT (Versionierungssoftware) - Backup der Arbeit auf externen Datenträger - Backup der Arbeit auf verschiedenen Cloud Diensten	- Einspielen von Backups - Zurückgehen auf frühere Commits	

Abbildung A.11.: Ein Ausschnitt von Projektrisiken des Risikomanagements.

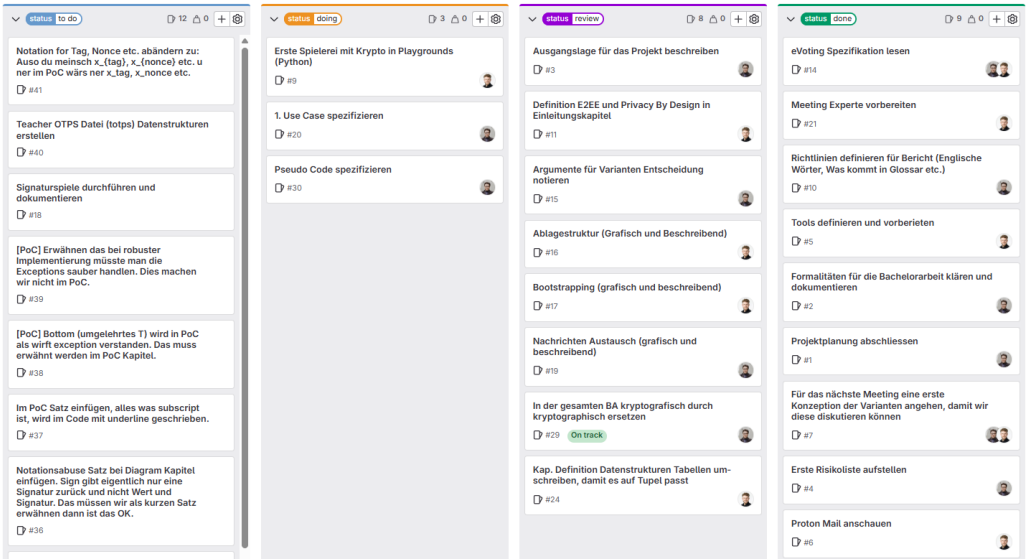


Abbildung A.12.: Ausschnitt des Storyboards während des Projektes

B. Proof of Concept Skripte

```python
import os
import time
import platform
import psutil
import cpuinfo

from src.config.security_parameters import SEC_PARAM_ASYM_BIT, SEC_PARAM_SYM_BIT
from src.crypto.core.asymmetric.enc_a import enc_a
from src.crypto.core.asymmetric.dec_a import dec_a
from src.crypto.core.asymmetric.sign import sign
from src.crypto.core.asymmetric.verify import verify
from src.crypto.core.symmetric.enc_s import enc_s
from src.crypto.core.symmetric.dec_s import dec_s
from src.crypto.core.asymmetric.gen_key_pair import gen_key_pair
from src.crypto.core.symmetric.gen_key import gen_key


def print_system_info():
    try:
        cpu_name = cpuinfo.get_cpu_info().get('brand_raw', platform.processor())
    except Exception:
        cpu_name = platform.processor()

    print("System␣information␣for␣benchmark:")
    print(f"␣␣System␣␣␣␣␣␣␣␣:␣{platform.system()}␣{platform.release()}")
    print(f"␣␣CPU␣model␣␣␣␣␣:␣{cpu_name}")
    print(f"␣␣CPU␣cores␣␣␣␣␣:␣{psutil.cpu_count(logical=False)}␣physical,␣
         {psutil.cpu_count(logical=True)}␣logical")
    print(f"␣␣RAM␣(total)␣␣␣␣:␣{round(psutil.virtual_memory().total␣/␣1024**3,␣2)}␣GB")
    print(f"␣␣Python␣version:␣{platform.python_version()}")
    print(f"␣␣Asym.␣keysize␣:␣{SEC_PARAM_ASYM_BIT}␣bit")
    print(f"␣␣Sym.␣keysize␣␣:␣{SEC_PARAM_SYM_BIT}␣bit\n")


def benchmark(name: str, fn, rounds: int = 1000) -> float:
    start = time.time()
    for _ in range(rounds):
        fn()
    duration = time.time() - start
    print(f"{name}:␣{duration:.3f}s␣for␣{rounds:,}␣ops␣({rounds␣/␣duration:.1f}␣ops/sec)")
    return duration


def main():
    print_system_info()

    # Generate RSA key pair
    public_key, private_key = gen_key_pair()

    # ==== RSA-OAEP Encrypt/Decrypt ====
    plaintext = os.urandom(32)
```

```
    ciphertext = enc_a(public_key, plaintext)

    benchmark("RSA-OAEP Encrypt", lambda: enc_a(public_key, plaintext), rounds=300)
    benchmark("RSA-OAEP Decrypt", lambda: dec_a(private_key, ciphertext), rounds=300)

    # ==== RSA-PSS Sign/Verify ====
    message = os.urandom(128)
    signature = sign(private_key, message)

    benchmark("RSA-PSS Sign", lambda: sign(private_key, message), rounds=300)
    benchmark("RSA-PSS Verify", lambda: verify(public_key, message, signature), rounds=300)

    # ==== AES-GCM Encrypt/Decrypt ====
    sym_key = gen_key()
    nonce = os.urandom(12)
    aad = b"optional aad"
    data = os.urandom(1024)
    ciphertext, tag = enc_s(sym_key, nonce, aad, data)

    benchmark("AES-GCM Encrypt", lambda: enc_s(sym_key, nonce, aad, data), rounds=300)
    benchmark("AES-GCM Decrypt", lambda: dec_s(sym_key, nonce, aad, ciphertext, tag),
         rounds=300)


if __name__ == "__main__":
    main()
```

Listing B.1: Benchmark-Skript zur Performance-Messung der Kryptographie

```
# Project Makefile for PoC Crypto Framework
# Run: `make` or `make all` to execute tests

.PHONY: all test coverage lint check-format format bandit audit clean

# === Configuration ===
SRC_DIRS := src/ tests/
PYTHONPATH := .

# === Default target ===
all: test coverage lint format check-format bandit audit

# === Tests ===
test:
        PYTHONPATH=$(PYTHONPATH) pytest -v tests/

coverage:
        PYTHONPATH=$(PYTHONPATH) pytest --cov=src --cov-report=term-missing --cov-report=html
             tests/

# === Linting and Formatting ===
lint:
        ruff check $(SRC_DIRS)

format:
        ruff format $(SRC_DIRS)

check-format:
        ruff format --check $(SRC_DIRS)

# === Security ===
bandit:
```

```
32          bandit -r src/ -x tests/
33
34  audit:
35          pip-audit
36
37  # === Setup Environment ===
38  setup:
39          python3 -m venv .venv
40          . .venv/bin/activate && pip install -U pip && pip install -r requirements.txt
41
42  # === Run application ===
43  run:
44          PYTHONPATH=. python3 src/protocol/main.py
45
46  # === Maintenance ===
47  clean:
48          find . -type d -name "__pycache__" -exec rm -r {} +
49          rm -rf htmlcov .pytest_cache .coverage
```

Listing B.2: Makefile zur Automatisierung von Qualitäts- und Sicherheitsprüfungen im *PoC*.

C. Aufgabenstellung Bachelorarbeit

Siehe nächste Seite.

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Bachelorthesis

für	Abidin Vejseli Nicolin Claudio Dora
Studiengang	Informatik
Betreuer:in	Philipp Locher

Digital und sicher: Neue Wege für die Schulkommunikation

Durch die fortschreitende Digitalisierung eröffnen sich für Schulen vielfältige Möglichkeiten, Abläufe zu optimieren. Davon profitieren Lehrpersonen, Eltern sowie Schülerinnen und Schüler gleichermassen. Digitale Lösungen verringern nicht nur den Verwaltungsaufwand, sondern vereinfachen zugleich die schulische Kommunikation. Mit diesen Chancen steigen jedoch auch die Anforderungen an Datenschutz und Datensicherheit, insbesondere bei der Verarbeitung sensibler Daten wie Schülerinformationen.

Klapp ist eine schweizerische Kommunikationslösung für den Bildungsbereich und verspricht nebst einfacher Bedienung, Reduzierung des administrativen Aufwands auch Privacy by Design. Allerdings kann Klapp sein Versprechen bezüglich des Schutzes der Privatsphäre nicht vollständig einhalten, da es keine konsequente Ende-zu-Ende-Verschlüsselung für sensible Daten bietet.

Aufbauend auf den Erkenntnissen der Projektarbeit soll in dieser Bachelorarbeit eine Schulkommunikations-Anwendung entworfen werden, welches den heutigen Ansprüchen an Privacy by Design gerecht wird. Im Zentrum steht die Ausarbeitung und Spezifikation der kryptographischen Protokolle zur Umsetzung der Kernfunktionen einer Schulkommunikations-Anwendung. Darüber hinaus sollen ausgewählte Aspekte im Rahmen eines Proof of Concepts implementiert werden.

Betreuer:innen:

P. Locher
Angabe des Signierenden
philipp.locher@bfh.ch
06.02.2025
Einfache Signatur
EES
Unterzeichnet über Switch Sign - Powered by Certifaction

Der Studiengangsleiter:

michael.roethlin@bfh.ch
06.02.2025
Einfache Signatur
EES
Unterzeichnet über Switch Sign - Powered by Certifaction