Mifare DESFire

Eine Einführung

Mario Haustein

14. März 20201

Chemnitzer Linux-Tage 20201

Chemnitzer Linux-Tage
13. und 14. März 2021

Mach es einfach!
anders.

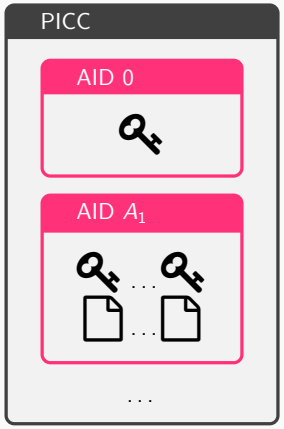
Mifare DESFire EV1

- Kontaktloser Nahfeld-Chipkartenstandard (NFC), 13.56 MHz
- Kartenleser (PCD) versorgt Chipkarte (PICC) durch HF-Feld mit Energie
- Die Karte ist ein Datenspeicher, mehr nicht!
  - flexible Organisation der Datenstrukturen
  - Speichergrößen: 2 KiB, 4 KiB, 8 KiB
  - kein Hardware Security Module!
- Nachfolger von Mifare Classic
  - Einsatz von Standard-Krypto (DES, 3DES, AES)

Software

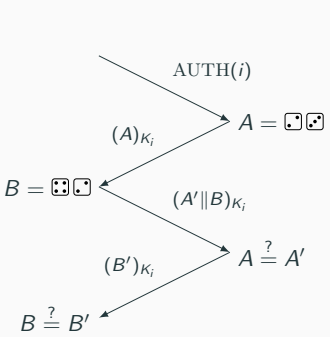
- libnfc
  - Low-Level-Kommunikation zwischen USB-Kartenlesern und der Karte
  - unabhängig von der Kartentechnologie
  - `https://github.com/nfc-tools/libnfc`
- libfreefare
  - `https://github.com/nfc-tools/libfreefare`
  - Zugriffslogik für Mifare Classic, Mifare DESFire, Mifare Plus, ...
  - baut auf libnfc auf
- Anwendungssoftware
  - Für DESFire konnte ich keine Software ausfindig machen.
  - Eigenentwicklung als interaktive Shell aufbauend auf libfreefare
  - Genehmigung meines Arbeitgebers zur Veröffentlichung unter eine Open-Source-Lizenz notwendig.

Apps



- Bis zu 28 Apps pro Karte
  - AID: 3 Byte
  - max. 14 Schlüssel
  - max. 32 Files
- Schlüssel dienen der Authentifizierung ggü. der Karte.
- Daraus leiten sich Zugriffsrechte auf Files ab.
- Sonderfall App 0
  - Immer vorhanden.
  - Keine Files, nur ein Schlüssel ⇒ PICC-Master-Key
  - Kann Apps anlegen und löschen.
  - Kann Karte zurücksetzen.

Schlüssel



- Gegenseitige Authentifizierung von Leser und Karte

1. PCD startet Auth. und überträgt Schlüsselnummer i.
2. PICC würfelt Zufallszahl A und verschlüsselt sie mit K_i.
3. PCD entschlüsselt A, ergänzt Zufallszahl B und verschlüsselt mit K_i.
4. PICC entschlüsselt A und B, prüft A auf Gleichheit und verschlüsselt B.
5. PCD entschlüsselt B und prüft auf Gleichheit.
6. $A \oplus B$ bildet Sitzungschlüssel für weitere Kommunikation.

Files

- Standard Data File eine Datei beliebiger Größe
- Backup Data File eine Datei mit transaktionsfähigem Zugriff
  - Änderungen müssen durch eine Commit-Operation abgeschlossen werden.
- Value File 32 Bit Ganzzahl
  - Wert kann je nach Berechtigung auf- oder abgewertet werden.
  - Es kann eine Ober- und Untergrenze festgelegt werden.
  - Änderungen müssen durch eine Commit-Operation abgeschlossen werden.
- Linear Record File satzorientierte Datei
  - Es wird immer ein kompletter Satz angehängt.
  - Änderungen müssen durch eine Commit-Operation abgeschlossen werden.
  - Ist die Maximalzahl an Sätzen erreicht, können keine weiteren Sätze angehängt werden.
- Cyclic Record File
  - wie LRF, nur dass bei Bedarf der älteste Satz überschrieben wird

ACLs

- Für jedes File werden vier Berechtigungen vergeben.

Data File, Record File	Value File
lesen	lesen + abwerten
schreiben	zzgl. eingeschränkt aufwerten
lesen + schreiben	zzgl. unbeschränkt aufwerten
Berechtigungen ändern	

- Jede Berechtigung wird als 4-Bit-Zahl dargestellt und kann ...
  - einer Schlüsselnummer (0 – 13) zugewiesen sein
  - für unauthentifizierte Zugriffe gelten (14)
  - nicht vergeben sein (15)

Dateizugriff: Data Files

- `WriteData(`$\langle f \rangle$`,`$\langle offset \rangle$`,`$\langle content \rangle$`)`
  - $\langle content \rangle$ ab $\langle offset \rangle$ in Datei $\langle f \rangle$ schreiben
- `ReadData(`$\langle offset \rangle$`,`$\langle len \rangle$`)`
  - $\langle len \rangle$ Bytes ab $\langle offset \rangle$ aus File $\langle f \rangle$ lesen
- `Commit`
  - Änderungen sichern
  - nicht notwendig bei Standard Data Files
  - Bis zum Commit liest `ReadData` die ursprünglichen Daten
- `Abort`
  - Änderungen verwerfen

Anwendungsbeispiel: Standard Data File

Operation		Inhalt	Hinweis
`WriteData(0, "Hallo")`		`Hallo`	schreiben
`ReadData(0, 5)`	→ `"Hallo"`	`Hallo`	lesen
`ReadData(1, 3)`	→ `"all"`	`Hallo`	Teilbereich lesen
`WriteData(5, "␣Welt")`		`Hallo␣Welt`	Teilbereich schreiben
`ReadData(0, 10)`	→ `"Hallo␣Welt"`	`Hallo␣Welt`	
`WriteData(1, "e")`		`Hello␣Welt`	überschreiben
`ReadData(0, 10)`	→ `"Hello␣Welt"`	`Hello␣Welt`	

Anwendungsbeispiel: Backup Data File

Operation		Inhalt	Hinweis
`WriteData(0, "Hallo")`		`Hallo`	
`ReadData(0, 5)`	→ `\x00\x00\x00\x00\x00`	`Hallo`	
`Commit`		`Hallo`	Änderung sichern
`ReadData(0, 5)`	→ `"Hallo"`	`Hallo`	
`WriteData(1, "e")`		`Hello`	
`ReadData(0, 5)`	→ `"Hallo"`	`Hello`	
`Abort`		`Hallo`	Änderung verwerfen
`ReadData(0, 5)`	→ `"Hallo"`	`Hallo`	

Dateizugriff: Value Files

- `GetValue(`$\langle f \rangle$`)` Zählerstand abfragen
- `Debit(`$\langle f \rangle$`,`$\langle amount \rangle$`)` Zähler um den Betrag $\langle amount \rangle$ abwerten.
- `Credit(`$\langle f \rangle$`,`$\langle amount \rangle$`)` Zähler um den Betrag $\langle amount \rangle$ aufwerten.
- `LimitedCredit(`$\langle f \rangle$`,`$\langle amount \rangle$`)`
  - Zähler innerhalb des Kreditrahmen um den Betrag $\langle amount \rangle$ aufwerten.
  - Der Betrag ist durch die letzte Transaktion mit `Debit`-Aufrufen begrenzt.
  - Es darf höchsten um die Summe der `Debit`-Aufrufe dieser Transaktion aufgewertet werden.
  - Anschließend wird das Limit für `LimitedCredit` auf 0 gesetzt.
- Die Zugriffe müssen mit `Commit` bzw. `Abort` abgeschlossen werden.
- Mehrere Operationen innerhalb einer Transaktion werden kumuliert.

Anwendungsbeispiel: Value File

- Value File mit Ausgangswert 0

Operation	Wert	Kreditrahmen
`Credit(50)`	50	0
`Debit(10)`	40	0
`Debit(5)`	35	0
`Commit`	35	15
`LimitedCredit(7)`	42	0
`Commit`	42	0
`Debit(10)`	32	0
`Abort`	42	0
`Credit(12)`	30	10
`Commit`	30	10
`Debit(0)`	32	0
`Commit`	32	0

Dateizugriff: Record Files

- `WriteRecord(`⟨*f*⟩`,`⟨*offset*⟩`,`⟨*content*⟩`)`
  - Schreibt ⟨*content*⟩ ab ⟨*offset*⟩ in einen leeren Satz der ⟨*f*⟩.
  - Bei Cyclic Record Files wird der älteste Satz überschrieben, wenn es keinen leeren Satz mehr gibt.
  - Bei Linear Record Files schlägt der Aufruf fehl, wenn es keinen leeren Satz mehr gibt.
- `ReadRecord(`⟨*f*⟩`,`⟨*offset*⟩`,`⟨*count*⟩`)`
  - Liest die ⟨*count*⟩ ältesten Sätze der Datei ⟨*f*⟩ aus.
  - Die ⟨*count*⟩ ältesten Sätze werden übersprungen.
  - Die Sätze werden in chronologischer Reihenfolge ausgegeben.
- `ClearRecord(`⟨*f*⟩`)`
  - Löscht alle Sätze der Datei ⟨*f*⟩.
- Die Zugriffe müssen mit `Commit` bzw. `Abort` abgeschlossen werden.

Anwendungsbeispiel: Linear Record File

- File mit 3 Sätzen.

Operation		Inhalt	Hinweis
`WriteRecord(0, "Chemnitzer")`		`Chemnitzer`	Satz einfügen
`Commit`		`Chemnitzer`	
`WriteRecord(0, "Windows")`		`Chemnitzer, Windows`	
`Abort`		`Chemnitzer`	Änderung verwerfen
`WriteRecord(0, "Linux")`		`Chemnitzer, Linux`	
`Commit`		`Chemnitzer, Linux`	
`WriteRecord(0, "Tag")`		`Chemnitzer, Linux, Tag`	
`WriteRecord(0, "Tage")`		`Chemnitzer, Linux, Tage`	Satz korrigieren
`Commit`		`Chemnitzer, Linux, Tage`	
`WriteRecord(0, "2021")`	↯	`Chemnitzer, Linux, Tage`	kein freier Satz
`ClearRecords`		~~`Chemnitzer, Linux, Tage`~~	Alle Sätze löschen
`Commit`		∅	

Anwendungsbeispiel: Linear Record File

- File mit 4 Sätzen (von alt nach jung):
  - `Chemnitzer`
  - `Linux`
  - `Tage`
  - `2021`

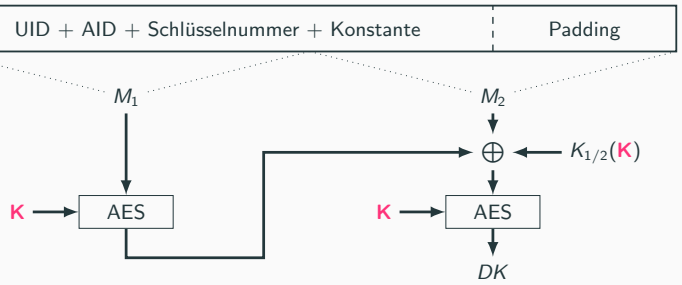
Operation	Ergebnis
`ReadRecord(0, 2)`	→ `Chemnitzer, Linux`
`ReadRecord(0, 3)`	→ `Chemnitzer, Linux, Tage`
`ReadRecord(0, 4)`	→ `Chemnitzer, Linux, Tage, 2021`
`ReadRecord(1, 2)`	→ `Linux, Tage`
`ReadRecord(2, 2)`	→ `Tage, 2021`

Key-Diversification

- Niemals nie ein und den selben Schlüssel auf mehreren Karten verwenden!
- Was tun, wenn der Kartenleser keine Schlüsseldatenbank hat?

⇒ Schlüssel *DK* im Leser berechnen aus . . .

- öffentlichen Daten der Karte: UID, AID, Schlüsselnummer, . . .
- einem Geheimnis *K*



Was wir nicht betrachtet haben

- Sicherheitseinstellungen der Apps
- Anlegen und Löschen von Files und Apps
- Key Version Number
- Schlüsseländerung
- Speicherverbrauch
- DESFire EV2 Features
- ISO 7816-4 Kompatibilität

Und nun . . .

Vielen Dank und . . .

viel Spaß am Gerät!