Karlsruher Institut für Technologie

CryptoCast: Implementierung

Institut für Kryptographie und Sicherheit

Gliederung

- Technische Umsetzung
  - Kryptosystem + Protokoll
  - Algorithmen
  - Bibliotheken + Werkzeuge
  - Probleme (+ Lösungen)
  - Zahlen
- Abweichungen vom Entwurf
- Grobe Zeitaufteilung
- Ausblick
- Produktdemo

Kryptosystem

- Naor-Pinkas mit allwissendem Server
- Generell in jeder DDH-Gruppe primaler Ordnung möglich
- Implementierungen
  - Echte, große Untergruppe von $\mathbb{Z}_p^\times$ (Schnorr-Gruppe)
  - Gruppe, die durch Punkt-Addition in einer elliptischen Kurve über $\mathbb{Z}/p\,\mathbb{Z}$ von einem geeigneten Basispunkt *Q* erzeugt wird
- Interessante Operation: Exponentieren im ersten Fall, Skalarmultiplikation im zweiten Fall
- Server und authorisierte Clients kennen durch NP den Wert *G*
- Kommunikation via Nachrichtenprotokoll:
  - Payload-Pakete, verschlüsselt mit AES-128
  - Key-Update-Pakete: NP-Nachricht + neuer Session-Key, gehasht und AES-verschlüsselt mit Schlüssel *G*

Algorithmen

- **L10**: $t = 500$ und $n = 10^5$
- Effiziente Berechnung der Lagrange-Koeffizienten?
  - Formel:

$$c_i = \prod_{j \neq i} \frac{x_j}{x_j - x_i} = \frac{\prod_k x_k}{x_i \prod_{j \neq i} (x_j - x_i)}$$

  - Nach Umformung direkte Berechnung in $O(t^2)$ für $t < 10^4$ mit Parallelisierung auf Phenom II X6 unter 5 Sekunden.
  - Optimierung: Beim Revoken und Unrevoken in $O(t)$ alle Koeffizienten updaten (jeweils nur eine Division und Multiplikation)
  - Insgesamt erscheinen t in der Größenordnung 10^5 hier durchaus machbar

Algorithmen

- Effiziente Polynomauswertung?
  - Naiver Ansatz: $O(nt)$, zu langsam!
  - Im Fall $t << n$ mithilfe von schneller Polynommultiplikation und Division (FFT) in $O(n \log^2 n)$ möglich
  - Auswertung für $t = 10^4$ und $n = 10^5$ mit Parallelisierung in ~5 Sekunden auf Phenom II X6
  - Also auch hier viel Luft nach oben, der Algorithmus skaliert vor allem in n

Algorithmen

- Berechnungen auf dem Client:
  - c_i von Server

$$g^{rP(0)} \equiv \prod_i G_i^{c_i} \quad \text{mit} \quad G_i \equiv g^{rP(I_i)}$$

bzw.

$$rP(0)Q \equiv \sum_i c_i G_i \quad \text{mit} \quad G_i \equiv rP(I_i)Q$$

  - Beide Berechnungen können mit Multiexponentierungsalgorithmen optimiert werden. Shamir's Trick (generalisiertes Square-Multiply) ohne größere Anpassungen funktioniert für beide Gruppen genau gleich
  - Implementierung daher generisch, um Duplikation zu vermeiden

Bibliotheken und Werkzeuge

- Mathematik
  - Java's BigInteger für die puren Java-Implementierungen
  - GMP (Multipräzisionsarithmetik) und FLINT 2 (Zahlentheorie, Polynome) für native Implementierungen geben Speedup von 10 und mehr
- Unit-Tests
  - JUnit 4 und Mockito für alle Tests
  - Robolectric als Laufzeitumgebung für Client-Tests (Tests im Emulator sind viel zu langsam)
  - JUnit4Android, um alle Tests auch auf Android auszuführen
- slf4j für Logging + Tracing. Backend: Logback bzw. Android
- Guava als Ergänzung zur Java-Standardbibliothek
- Java Native Interface, um einzelne kritische Funktionen durch optionale C++-Implementierungen zu ergänzen (`private native static`)
- Maven 3 als Buildsystem

Probleme und Lösungen

- Problem: Android's `MediaPlayer` kann keinen rohen Datenstrom abspielen, nur URLs und Dateien.
  - Lösung: Minimaler HTTP-Server, der den Datenstrom progressiv per Chunked Encoding dem `MediaPlayer` serviert
- Offenes Problem: EC-Arithmetik in der Theorie und auf dem Server sehr schnell, aber auf Dalvik unbrauchbar langsam
  - Einzige skalierbare Lösung: nativer Code auf dem Gerät. Aufwendig!

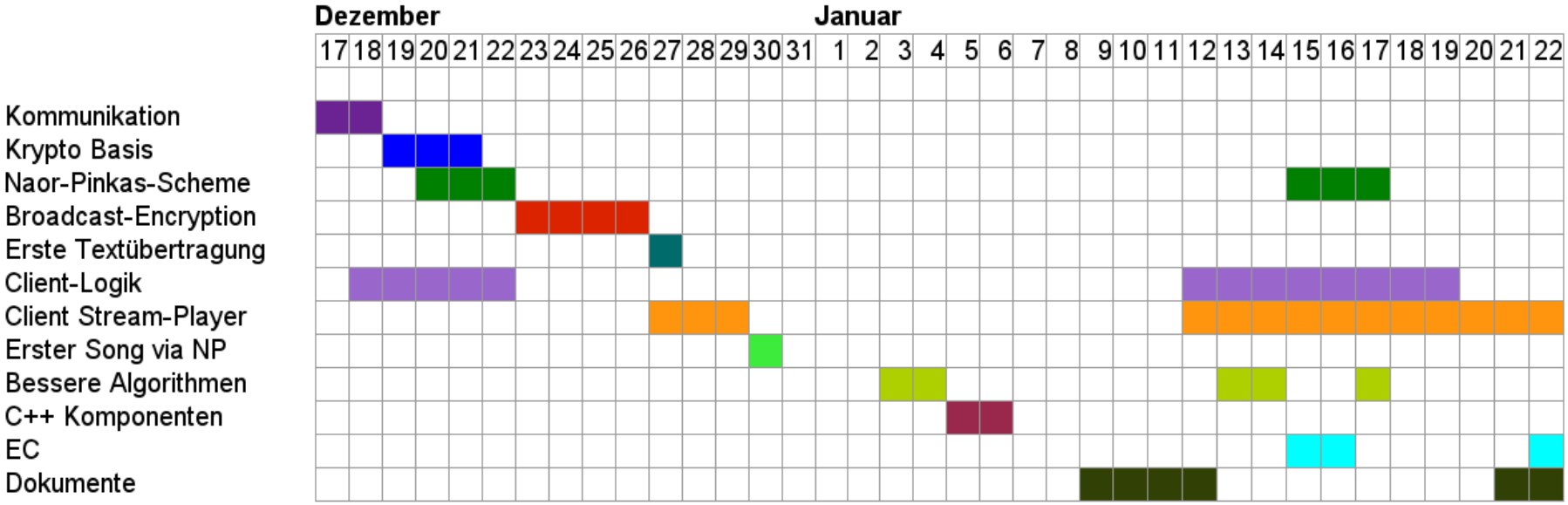
Zahlen

- 661 Commits
- ~7400 LoC (davon 1700 Unit-Tests, 400 C++)
- 140 Java-Dateien (davon 36 Tests)
- 189 Typen
- 66% Testabdeckung (>85% bei Kernfunktionalität)
- Durchschnitt
  - 6 Zeilen pro Methode
  - 2 Felder pro Typ
  - 2 Methoden pro Typ

Abweichungen vom Entwurf

- Android API Level 10 statt 8 (2.3 statt 2.2)
  - Hauptgrund: `MediaPlayer` funktioniert hier tatsächlich wie spezifiziert
- “Channel”-Abstraktion existiert mit Java's `InputStream` / `OutputStream` schon
- Ansonsten hauptsächlich Erweiterungen der bestehenden Klassen und neue Hilfskonstrukte

Grobe Zeitaufteilung



Ausblick

- Naor-Pinkas hat mehr Potential als zunächst vermutet
- Elliptische Kurven verringern Kommunikations-Overhead stark
  - Bei großer Bandbreite (HD-Video) ist auch großes *t* in der Größenordnung 10^4 denkbar
- Klarer Flaschenhals: Entschlüsselung auf mobilen Geräten. Nativer Code für größere *t* unabdingbar! PCs als Endgerät haben dieses Problem nicht
- NP-Erweiterungen:
  - Public-Key-Scheme anstatt “Gott”-Server
  - Mehrere Polynome für jeden Benutzer (mit t = 2, 4, 8, 16, …).

    Je nach Anzahl der tatsächlich ausgeschlossenen Benutzer skaliert dann auch der Kommunikations-Overhead
  - Zusätzlich Partitionierung der Benutzer in Gruppen mit verschiedenen Polynomen. Verringert die erwartete Berechnungszeit auf dem Client