▼ Kryptographie & Sicherheit

Kryptographische Skatrunde

HEIKO STAMER: Die freie Bibliothek LibTMCG und eine Referenzimplementierung für Skat

Dieser Artikel beschreibt die Grundlagen und die technische Umsetzung eines mentalen Kartenspiels. Kryptographische Protokolle sichern dabei die Fairness zwischen den Teilnehmern. Nach einer kurzen Einführung lernen wir zwei Möglichkeiten der kryptographischen Kartenkodierung kennen und vergleichen sie hinsichtlich ihrer Komplexität. Danach folgt eine ausführliche Behandlung der freien Bibliothek LibTMCG, welche uns einfache Datenstrukturen und die notwendigen Protokolle für sichere mentale Kartenspiele zur Verfügung stellt. Schließlich wird als Anwendung eine Implementierung des Skatspiels diskutiert.

Sichere Mehrpersonenspiele erfordern in Netzwerken häufig einen zentralen Server oder Schiedsrichter, welcher für die faire Verteilung des Spielmaterials sorgt und die Einhaltung der jeweils geltenden Regeln überwacht. Die Teilnehmer vertrauen hierbei praktisch auf dessen Unabhängigkeit, die jedoch nicht immer gegeben ist.[1] Zudem ist ein solcher Schiedsrichter in manchen Situationen einfach nicht verfügbar. Deshalb gibt es seit vielen Jahren kryptographische Protokolle, welche ohne zentralen Vertrauenspunkt (Trusted Third Party) die Fairness (in gewissen Grenzen) sicherstellen können. Deren Funktionsweise beruht auf sogenannten Zero-Knowledge Beweisen[2], welche – grob gesprochen – bestimmte Eigenschaften eines Objektes zeigen können, ohne daß hierbei Wissen[3] über den Gegenstand selbst offengelegt wird. Diese Beweise sind probabilistischer Natur, d. h. sie haben eine gewisse Fehler- bzw. Betrugswahrscheinlichkeit $\mathfrak{p} \leq 1/2$, die aber durch sequentielle Wiederholung schnell in den vernachlässigbaren Bereich gerückt werden kann. Ein erstaunliches Resultat auf diesem Gebiet ist die Tatsache [GMW87], daß für alle Sprachen $L \in \mathcal{NP}$ die Zugehörigkeit beliebiger Elemente zu L ohne deren Preisgabe beweisbar ist, falls man die Existenz sicherer Verschlüsselungsfunktionen voraussetzt.

Das mächtige Konzept der *Zero-Knowledge Beweise* hat vielfältige Anwendungsmöglichkeiten: Identifizierungsprotokolle, Gruppensignaturschemata (z. B. Direct Anonymous Attestation [BCC04, 2] im TPM-Standard v1.2 der Trusted Computing Group), verifizierbare Verschlüsselung und Geheimnisteilung (VSS), elektronische Wahlen, digitales Geld, faire Online-Auktionen und sichere Mehrparteien-Berechnung. Zur letzten Kategorie zählt auch die Gewährleistung der Fairness bei virtuellen Kartenspielen, mit welcher wir uns in diesem Artikel näher beschäftigen wollen. Dem Konzept sind natürlich auch Grenzen gesetzt: Verlust oder mutwillige Weitergabe vertraulicher Informationen durch den Betrachter bzw. Eigentümer kann kein Protokoll verhindern!

Einleitung

Prinzipiell können *mentale Kartenspiele* als Interaktion zwischen (mindestens zwei) konkurrierenden Parteien angesehen werden, wobei keinerlei physikalisches Spielmaterial (also Spielkarten) zum Einsatz kommt. Wer auf diese seltsam erscheinende Idee kam, ist nicht eindeutig: In einer der ersten Arbeiten [RSA79] zu diesem Thema, wird die Fragestellung „Is it possible to play a fair game of Mental Poker?“ auf Robert W. Floyd zurückgeführt. Allerdings soll Werner Heisenberg in seinen Memoiren erwähnt haben, daß sich bereits Niels Bohr während eines langweiligen Ski-Urlaubs mit mentalen Kartenspielen befaßte, aber zu keinem brauchbaren Ergebnis kam. [1]

Das einfachste Beispiel eines mentalen Spiels wurde 1981 in [Blu81] betrachtet: Zwei Parteien möchten einen Münzwurf übers Telefon durchführen. Beide wollen das Ergebnis zu ihren Gunsten beeinflussen und es gibt keine vertrauenswürdige dritte Person. Die von Manuel Blum vorgestellte Lösung verwendet erstmals eine Kodierung auf Basis quadratischer Reste. Aufgrund ihrer homomorphen Eigenschaft wurde diese Kodierung dann lange Zeit als *state-of-the-art* in vielen Protokollen eingesetzt.

Kurz darauf stellten Shafi Goldwasser und Silvio Micali eine Variante [GM82] für das Kartenspiel „Poker“ vor, welche das Offenlegen aller Karten nach dem Spielende erfordert, um stattgefundene Betrugsversuche zu entdecken. Gerade aber für Poker ist diese Lösung unbefriedigend, weil hierbei die Strategie der Spieler ersichtlich wird. Weitere Nachteile sind die enorme Komplexität und die ineffiziente Verwendung von Primzahlen.

Die erste vollständige Lösung für Poker wurde 1987 von Claude Crépeau [Cré87] veröffentlicht. Die Idee basiert ebenfalls auf der Kodierung mit quadratischen Resten, nutzt aber das ANDOS-Schema (*all-or-nothing disclosure of secrets*) [BCR87]. Insbesondere kann damit erreicht werden, daß keine verdeckte Karte bei Spielende geöffnet werden muß und so die Spielstrategie der Teilnehmer geheim bleibt. Allerdings hatte auch diese Lösung noch Nachteile: die sequentielle Abhängigkeit einzelner Teilprotokolle, die Voraussetzung eines eindeutigen Kartentyps und das Fehlen wichtiger Karten- und Stapeloperationen.

Schließlich hat Christian Schindelhauer [Sch98] diesen Ansatz 1998 aufgegriffen und an vielen Stellen erweitert bzw. verallgemeinert. Sein Werkzeugkasten bietet deshalb erstmals die Möglichkeit, fast jedes beliebige Kartenspiel in sicherer Art und Weise elektronisch umzusetzen, ohne dabei auf die Verfügbarkeit eines vertrauenswürdigen Vermittlers angewiesen zu sein. Natürlich kann auch

[1] Jeder rational handelnde Eigentümer eines Online-Spielkasinos möchte sein Geschäft natürlich profitabel und risikolos betreiben.

[2] Literaturempfehlung: Die Autoren der Arbeit *How to Explain Zero-Knowledge Protocols to Your Children* [QGB89] erklären das dahinterstehende Konzept sehr unterhaltsam und theoriearm.

[3] Man beachte den Bedeutungsunterschied der Wörter „Wissen“ und „Information“, wobei wir für letzteres die Definition aus der klassischen Informationstheorie heranziehen. Beispiel: Es sei die Zahl $2^{2^{23}} - 1$ gegeben. Ihre Primfaktoren stellen keine zusätzliche Information dar, weil die Zerlegung eindeutig ist und alle Faktoren deshalb implizit in der Zahl enthalten sind. Jedoch kann ihre Kenntnis einen Wissensvorsprung bedeuten, weil andere Instanzen aufgrund der Größe nicht in der Lage sind, die Faktorisierung sofort zu bestimmen.

diese Lösung nicht verhindern, daß unehrliche Spieler ihre (eigentlich verdeckt zu haltenden) Karten insgeheim einer Koalition preisgeben und daraus Vorteile ziehen. Es wird lediglich garantiert, daß die privaten Karten jedes Spielers nicht *gegen seinen Willen* von anderen Teilnehmern gesehen oder verändert werden können.

Zwei aktuelle Vorschläge für Kartenkodierungen stammen von Adam Barnett und Nigel Smart [BSm03]. Die Autoren verallgemeinern zuerst das Modell der Maskierung virtueller Spielkarten, indem sie die abstrakte kryptographische Primitive VTMF (*Verifiable k-out-of-k Threshold Masking Function*) einführen. Dann werden die Karten- und Stapeloperationen aus [Sch98] im Kontext dieser Notation beschrieben. Schließlich stellen Barnett und Smart noch zwei konkrete Implementierungen solcher VTMFs vor. Beide Ansätze basieren nicht mehr auf quadratischen Resten und ermöglichen deshalb eine sehr effiziente Kodierung der Spielkarten.

Kryptographische Grundlagen

Zuerst ein paar allgemeine Definitionen, stellenweise entnommen aus WIKIPEDIA [15, 16, 17]: In der Gruppentheorie ist eine *zyklische Gruppe* eine Gruppe, die von einem einzelnen Element erzeugt wird. Sie besteht aus allen Potenzen dieses Erzeugers g, d. h. $\langle g\rangle := \{g^n \mid n \in \mathbb{Z}\}$. Eine beliebige Gruppe G ist also zyklisch, wenn sie ein Element g enthält (*Erzeuger der Gruppe*), so daß jedes Element aus G eine Potenz von g ist. Zyklische Gruppen sind die einfachsten Gruppen und können vollständig klassifiziert werden: Für jede natürliche Zahl n gibt es eine zyklische Gruppe C_n mit genau n Elementen, und es gibt die unendliche zyklische Gruppe, die additive Gruppe der ganzen Zahlen $\mathbb{Z}$. Jede andere zyklische Gruppe ist zu einer dieser Gruppen isomorph. Ist n eine natürliche Zahl, dann faßt man ganze Zahlen mit gleichem Rest bei Division durch n zu sogenannten *Restklassen* zusammen. Die Restklassen bilden zusammen den *Restklassenring*, der mit $\mathbb{Z}/n\mathbb{Z}$ oder $\mathbb{Z}_n$ bezeichnet wird (sprich $\mathbb{Z}$ modulo n). Eine Restklasse $a + n\mathbb{Z}$ mit $\mathrm{ggT}(a, n) = 1$ heißt *prime Restklasse modulo* n. Die Gruppe der primen Restklassen modulo n heißt *prime Restklassengruppe modulo* n und wird mit $(\mathbb{Z}/n\mathbb{Z})^*$ bzw. $\mathbb{Z}_n^*$ bezeichnet. Weil diese Struktur eine endliche abelsche Gruppe bezüglich der Multiplikation bildet, spielt sie in der Kryptographie eine bedeutende Rolle. Die *Eulersche Phi-Funktion* ist eine zahlentheoretische Funktion. Sie ordnet jeder natürlichen Zahl n die Anzahl der Zahlen $a \in \mathbb{N}, a \leq n$ zu, die zu n teilerfremd sind (also $|\{a \in [1, n] \mid \mathrm{ggT}(a, n) = 1\}|$). Sie ist benannt nach Leonhard Euler und wird mit dem griechischen Buchstaben φ (Phi) bezeichnet. Ihr Wert entspricht der Gruppenordnung von $\mathbb{Z}_n^*$, d. h. $\varphi(n) = |\mathbb{Z}_n^*|$. Betrachten wir vorerst die Menge $\mathbb{P}$ der Primzahlen: Da jede Primzahl p nur durch 1 und sich selbst teilbar ist, bleibt sie auch zu den Zahlen 1 bis $p - 1$ teilerfremd, daher gilt $\varphi(p) = p - 1$. Die Phi-Funktion ist multiplikativ für zwei Zahlen $m, n \in \mathbb{N}$ sofern diese keinen gemeinsamen Teiler außer 1 haben, d. h. $\varphi(mn) = \varphi(m)\varphi(n)$ falls $\mathrm{ggT}(m, n) = 1$. Die Berechnung von φ für zusammengesetzte n ergibt sich aus dieser Multiplikativität: Hat unsere Zahl die kanonische Darstellung $n = p_1^{e_1} \cdot p_2^{e_2} \cdots p_k^{e_k}$ mit $e_i \geq 1, p_i \in \mathbb{P}$ für $i = 1, \ldots, k$, dann gilt $\varphi(n) = (p_1-1)p_1^{e_1-1} \cdot (p_2-1)p_2^{e_2-1} \cdots (p_k-1)p_k^{e_k-1}$. Die Menge der *quadratischen Reste modulo* n definiert man als $\mathbb{QR}_n := \{a \in \mathbb{Z}_n^* \mid \exists b \in \mathbb{Z}_n : b^2 \equiv a \pmod n\}$. Sie ist eine Untergruppe von $\mathbb{Z}_n^*$. Analog bezeichnet $\mathbb{NQR}_n := \mathbb{Z}_n^* \setminus \mathbb{QR}_n$ die Menge der *quadratischen Nichtreste modulo* n, welche jedoch nicht multiplikativ abgeschlossen ist. Das *Legendre-Symbol* für eine ungerade Primzahl p und ein $a \in \mathbb{Z}_p^*$ sei wie folgt definiert:

$$\left(\frac{a}{p}\right) := \begin{cases} +1 & a \in \mathbb{QR}_p \\ -1 & \text{sonst} \end{cases}$$

Das *Jacobi-Legendre-Symbol* ist die Verallgemeinerung für zusammengesetzte Zahlen $n \in \mathbb{N}$ und ein $a \in \mathbb{Z}_n^*$:

$$\left(\frac{a}{n}\right) := \begin{cases} \left(\frac{a}{n}\right) & \text{falls } n \text{ Primzahl} \\ \left(\frac{a}{p}\right) \cdot \left(\frac{a}{m}\right) & \text{falls } n = p \cdot m \text{ und } p \text{ Primzahl} \end{cases}$$

Weiterhin definieren wir (analog zu [Sch98]) die Mengen $\mathbb{Z}_n^\circ := \{a \in \mathbb{Z}_n^* \mid \left(\frac{a}{n}\right) = 1\}$ und $\mathbb{NQR}_n^\circ := \mathbb{Z}_n^\circ \cap \mathbb{NQR}_n$ (Pseudoquadrate modulo n). Falls nun n das Produkt von genau zwei verschiedenen Primzahlpotenzen ist,[4] dann hat die Gruppe $\mathbb{QR}_n$ die Ordnung $|\mathbb{Z}_n^*|/4$ und auch $\mathbb{NQR}_n^\circ$ enthält genau $|\mathbb{Z}_n^*|/4$ Elemente. Diese Eigenschaft ermöglicht die gleichverteilte Kodierung eines Bit als Zahl $z \in \mathbb{Z}_n^\circ$, so daß das Bit genau dann gesetzt ist, wenn z in $\mathbb{QR}_n$ liegt.

Effiziente kryptographische Protokolle verwenden oft „schwierige“ Probleme aus der Mathematik zur Umsetzung ihrer Sicherheitsziele, z. B. das Zerlegen großer natürlicher Zahlen in ihre Primfaktoren oder die Bestimmung des diskreten Logarithmus in endlichen Gruppen.[5] Für solche *kryptographischen Annahmen* existiert kein strenger Beweis im Sinne der Komplexitätstheorie. Dennoch geht man von der praktischen Schwierigkeit[6] folgender Probleme aus, falls die jeweiligen Sicherheitsparameter entsprechend groß gewählt sind:

Primfaktorisierung (FAKTOR) Darunter wollen wir das Problem verstehen, zu einer gegebenen positiven ganzen Zahl n ihre paarweise verschiedenen Primfaktoren

[4] $n = p_1^{e_1} \cdot p_2^{e_2}$ für ungerade Primzahlen $p_1 \neq p_2$ und $e_i \geq 1$

[5] Zumindest glaubt man, daß solche Fragen äußerst schwierig sind, weil bis heute kein schneller Algorithmus für ihre Lösung gefunden wurde. Natürlich kann ein falscher Glaube auch aufs „Glatteis“ führen, wie sich z. B. vor etwa zwei Jahren bei der positiven Beantwortung der lange untersuchten Fragestellung $\mathrm{PRIMES} \in_? \mathcal{P}$ gezeigt hat. Zwar waren in diesem Fall die Auswirkungen für die Kryptographie eher gering, jedoch sollte das nicht dazu verleiten, immer weitergehende Annahmen zutreffen.

[6] „Alienhardware“ (Zitat von Rüdiger Weis) ausgenommen

$p_1, p_2, \cdots, p_k$ zu finden, so daß $n = p_1^{e_1} \cdot p_2^{e_2} \cdots p_k^{e_k}$ und $e_i \geq 1$ gilt. Es existiert zwar ein Algorithmus der in Polynomialzeit feststellen kann, ob eine gegebene Zahl prim ist, allerdings scheint die Zerlegung in Faktoren weitaus schwieriger. Zur Zeit haben die schnellsten Algorithmen ein subexponentielles Laufzeitverhalten in der Größe von n, so daß die Zerlegung einer 300-stelligen Zahl (bestehend aus genau zwei, vergleichbar großen Primfaktoren) momentan noch unmöglich erscheint. Aber der technische Fortschritt bleibt nie stehen ...

Quadratisches Restproblem (QRP**)** Für eine Zahl $n \in \mathbb{N}$ und ein beliebiges $a \in \mathbb{Z}_n^*$ soll mit Wahrscheinlichkeit größer $1/2$ entschieden werden, ob a ein quadratischer Rest modulo n ist. Falls man die Primfaktorisierung von n kennt oder es sich sogar um eine Primzahl handelt, wird das Problem einfach. In diese Richtung existiert nämlich eine Reduktion $\mathrm{QRP} \leq_{\mathcal{P}} \mathrm{FAKTOR}$, die in Polynomialzeit durchgeführt werden kann.

Quadratwurzeln ($\mathrm{QWURZEL}$**)** Zu gegebenen $n \in \mathbb{N}$ und $a \in \mathbb{QR}_n$ soll ein $x \in \mathbb{Z}_n^*$ bestimmt werden, so daß $x^2 \equiv a \pmod{n}$ gilt. Falls n eine Primzahl ist, gibt es einen Polynomialzeitalgorithmus, der die gesuchte Quadratwurzel x berechnet. Für ein zusammengesetztes n kann man zeigen, daß diese Fragestellung äquivalent zur Primfaktorisierung ist, d. h. es existieren Reduktionen in beide Richtungen $\mathrm{QWURZEL} \equiv_{\mathcal{P}} \mathrm{FAKTOR}$.

Diskreter Logarithmus (DLP**)** Sei g ein Erzeuger der endlichen zyklischen Gruppe $\langle g \rangle$. Zu einem gegebenen Wert $y \in \langle g \rangle$ soll die kleinste positive ganze Zahl x (diskreter Logarithmus zur Basis g) bestimmt werden, so daß $y = g^x$ gilt. In Gruppen mit enorm vielen Elementen (große Gruppenordnung) kann zwar g^x „effizient" berechnet werden, die Umkehroperation scheint jedoch sehr schwierig zu sein.

Computational Diffie-Hellman (CDH**)** Zu gegebenen Werten $g^a \in \langle g \rangle, g^b \in \langle g \rangle$ (die Exponenten a, b sind dabei unbekannt) soll ein $z \in \langle g \rangle$ gefunden werden, so daß $z = g^{ab}$ gilt. Dieses Problem wird leicht, wenn man diskrete Logarithmen in $\langle g \rangle$ berechnen kann, d. h. $\mathrm{CDH} \leq_{\mathcal{P}} \mathrm{DLP}$. Für die Umkehrung ist eine solche Polynomialzeitreduktion nicht bekannt.

Decisional Diffie-Hellman (DDH**)** Diese stärkere Annahme besagt, daß es bei Kenntnis von $g^a \in \langle g \rangle$ und $g^b \in \langle g \rangle$ (die Exponenten a, b sind wieder unbekannt) schwierig ist, den Wert $g^{ab} \in \langle g \rangle$ von einem zufälligen Element $g^c \in \langle g \rangle$ zu unterscheiden, sofern a, b, c zufällig und uniform im Intervall $[1, |\langle g \rangle|]$ gewählt sind. Mit anderen Worten: Die Verteilungen $\{g^a, g^b, g^{ab}\}$ und $\{g^a, g^b, g^c\}$ sind effizient nicht unterscheidbar. Wie man sofort sieht, kann eine Polynomialzeitreduktion $\mathrm{DDH} \leq_{\mathcal{P}} \mathrm{CDH}$ zum vorherigen Problem konstruiert werden. Doch wie verhält sich die Fragestellung in anderer Richtung? Es gibt erstaunlicherweise einige Gruppen, in denen DDH trivial ist, aber das CDH-Problem noch schwer zu sein scheint. Das sind z. B. Gruppen, deren Ordnung durch kleine Primzahlen teilbar ist, also insbesondere $\mathbb{Z}_p^*$. Ergo muß bei der Auswahl einer DDH-schweren Gruppe mit besonderer Vorsicht vorgegangen werden. Dan Boneh [Bon98] nennt einige Familien von Gruppen, die hierfür in Frage kommen.

Random-Oracle Modell (ROM **[FS86, BR93, 3])** Ein *Zufallsorakel* ist eine Funktion $f : \{0,1\}^* \to \{0,1\}^n$ (für ein festes n), wobei der Funktionswert $f(x)$ für jede Eingabe $x \in \{0,1\}^*$ *echt zufällig* ist. Die Funktion ist öffentlich, d. h. man stellt keinerlei Anforderungen hinsichtlich der Vertraulichkeit. Insbesondere ist es also akzeptabel, wenn jedermann (auch ein potentieller Gegner) auf beliebige Werte von f zugreifen kann. Leider sind solche Orakel rein theoretische Konstruktionen und können nicht implementiert werden. Das dahinterstehende Konzept ist allerdings äußerst nützlich, weil es viele wünschenswerte Eigenschaften[7] besitzt und deshalb vereinfachte Sicherheitsbeweise ermöglicht. Approximativ kann f durch eine kryptographische Hashfunktion[8] implementiert werden. Es gibt viele Verfahren deren Sicherheit im *Random-Oracle Modell* bewiesen werden kann, also unter der Annahme, daß sich die verwendete Hashfunktion wie ein Zufallsorakel verhält.

Sichere mentale Kartenspiele

Wir wollen nun die konkrete Realisierung sicherer mentaler Kartenspiele betrachten. Allgemein werden folgende Anforderungen [Sch98, Gut00] gestellt:

1. Es sollen beliebig viele Mitspieler, Karten und Kartentypen möglich sein.
2. Die Datenstruktur für die Spielkarten erlaubt eine eindeutige Identifizierung jeder Karte (Rückseite). Der Typ (Vorderseite) einer verdeckten Karte bleibt jedoch verborgen, sofern er nicht durch eine gewollte Operation ersichtlich wird.
3. Stapel dienen als Behälter für Karten.
4. Die Karten- und Stapeloperationen garantieren ein breites Einsatzgebiet für diverse Kartenspiele. Beispielsweise sollen für Karten *das Erzeugen*, *das Verdecken (Maskieren)*, *das Offenlegen*, *das vom Stapel nehmen*, *das in den Stapel (geheim) einfügen*, *das zufällige Erzeugen* und für Stapel *das Erzeugen*, *das Vereinigen*, *das Teilen*, *das Mischen*, *das Abheben*, *das Vergleichen* zur Verfügung stehen.
5. Ein Teilnehmer kann den Typ einer von allen Mitspielern verdeckten Karte nur erfahren, falls auch alle damit einverstanden sind.
6. Keine Koalition von Teilnehmern kann den Typ privater Karten gegen den Willen eines einzelnen Spielers (Inhaber) bestimmen.[9]

[7] Beispielsweise erhält man durch Auswertung an beliebigen Stellen keine Information über den Funktionswert an noch nicht ausgewerteten Stellen. Weiterhin ist ein Zufallsorakel natürlich *kollisionsresistent*, d. h. es ist (bei großen n) nicht effizient möglich, zwei Eingaben $x, y \in \{0,1\}^*$ zu finden, für die $f(x) = f(y)$ gilt.

[8] z. B. SHA-1, RIPEMD-160 [5], ...

[9] triviale Schlußfolgerungen sind ausgenommen

7. Die Spielstrategie soll geheim bleiben, d. h. es ist nicht notwendig, verdeckt oder privat gebliebene Karten bei Spielende zu öffnen.
8. Der Berechnungs- und Kommunikationsaufwand sollte sich in vernünftigen Grenzen bewegen.

Unser *Angriffsmodell* sei wie folgt fixiert: Alle Spieler verhalten sich protokollkonform[10] (*semi-honest*), sind aber u. U. neugierig (bzgl. der Geheimnisse anderer Teilnehmer) oder arbeiten in Koalition zusammen, um sich einen Vorteil im Spiel zu verschaffen. Dieses Szenario wird oft auch als *honest-but-curious* Modell bezeichnet.

Die im nächsten Abschnitt behandelte Bibliothek LibTMCG basiert hauptsächlich auf Christian Schindelhauers *Toolbox for Mental Card Games*. Details der Zero-Knowledge Beweise können dem Technischen Report [Sch98] der Universität Lübeck entnommen werden. Wir gehen hier nur kurz auf Abweichungen bzw. Ergänzungen [BSm03] zur Kartenkodierung ein.

Modifikationen: Der Korrektheitsbeweis des öffentlichen Schlüssels ist durch die reduzierte Version[11] des Verfahrens von Gennaro, Micciancio und Rabin [GMR98] realisiert. Um deren Voraussetzungen zu erfüllen, mußte auch die Schlüsselstruktur geringfügig angepaßt werden: Wir verwenden sogenannte *sichere Primzahlen*[12] p_i und q_i, die wesentlich dünner in $\mathbb{N}$ verteilt sind, aber bessere Sicherheitseigenschaften haben. Weiterhin erfüllen p_i, q_i die Kongruenzen

$$p_i \equiv 3 \pmod 4, \qquad q_i \equiv 3 \pmod 4,$$

um eine schnelle Berechnung von Quadratwurzeln modulo $p_i \cdot q_i$ zu ermöglichen und

$$p_i \not\equiv 1 \pmod 8, \qquad p_i \not\equiv q_i \pmod 8,$$

um den Voraussetzungen von [GMR98] zu genügen. Dadurch wird natürlich die Schlüsselerzeugung langsamer, aber das Protokoll insgesamt robuster. Außerdem wurde die Verifikation des Schlüssels[13] mit Hilfe der *Fiat-Shamir Heuristik* [FS86, BR93] in einen nicht-interaktiven Zero-Knowledge Beweis [BFM88, BSG91] umgewandelt. Hier setzen wir also neben der kryptographischen Annahme, daß FAKTOR schwer ist, auch noch voraus, daß sich die Hashfunktion[14] g [BR93] wie ein Zufallsorakel (ROM) verhält.

Die erzeugten Schlüssel dienen (neben ihrer Verwendung in den Protokollen der Toolbox [Sch98]) parallel für asymmetrische Kryptographie.[15] Dabei wird das Verfahren von Rabin [Rab79, 4] sowohl für Signatur (PRab [BR96]) als auch für Verschlüsselung (SAEP [Bon01]) eingesetzt.

Kartenkodierungen: Jeder Spieler i besitzt einen Schlüssel bestehend aus geheimen (p_i, q_i) und öffentlichen (m_i, y_i) Teil. Hierbei ist $m_i = p_i \cdot q_i$ das Produkt zweier großer Primzahlen (z. B. 1024 Bit) und $y_i \in \mathbb{NQR}^{\circ}_{m_i}$ ist zufällig gewählt. Eine verdeckte Karte in einem Spiel mit k Teilnehmern und M verschiedenen Kartentypen[16] kodiert in $w = \lceil \log_2 M \rceil$ Bits wird durch die Matrix

$$Z = \begin{pmatrix} z_{1,1} & \cdots & z_{1,w} \\ \vdots & \ddots & \vdots \\ z_{k,1} & \cdots & z_{k,w} \end{pmatrix}$$

beschrieben, wobei die $z_{i,j} \in \mathbb{Z}^{\circ}_{m_i}$ sind. Die Matrixelemente enthalten jeweils ein Bit

$$b_{i,j} = \begin{cases} 0 & z_{i,j} \in \mathbb{QR}_{m_i} \\ 1 & \text{sonst} \end{cases}$$

verteilte Information über den Typ τ der Karte. Dieser Wert $\tau \in [0, M-1]$ kann durch die Formel

$$\tau = \sum_{j=1}^{w} 2^{j-1} \cdot \bigoplus_{i=1}^{k} b_{i,j}$$

berechnet werden, sofern alle Teilnehmer zustimmen und die Werte $b_{i,j}$ aus ihrer Zeile der Matrix offenlegen.[17] Unter der Annahme, daß die Bestimmung der Quadratresteigenschaft (QRP) modulo m_i schwer ist, stellt diese Kodierung eine für unsere Zwecke sichere Datenstruktur dar. Mit ihrer Hilfe können Protokolle [Sch98] konstruiert werden, die folgende Karten- und Stapeloperationen ermöglichen:

+ Erzeugung einer offenen oder verdeckten Karte mit festem Typ $\tau \in [0, M-1]$
+ Umwandlung (Maskierung) einer offenen in eine verdeckte Karte
+ Erzeugung einer verdeckten Karte mit zufälligem Typ τ (uniform aus $[0, M-1]$)
+ Aufnehmen und Offenlegen einer verdeckten Karte
+ Stapeln von offenen oder verdeckten Karten
+ Mischen[18] oder Abheben[19] eines Stapels
− Mengenvergleiche (Inklusion, Schnitt) von verdeckten mit offenen Stapeln
− Geheimes Einfügen einzelner Karten in einen verdeckten Stapel
− Zeitnahe Regelkontrolle direkt beim Ausspiel

Die Markierung gibt jeweils an, ob dieser Punkt bereits in der Bibliothek LibTMCG implementiert ist (+) oder noch nicht (−). Zur Verifikation der Korrektheit jeder Operation verwendet die Toolbox *cut-and-choose* Zero-Knowledge Beweise mit maximaler Kommunikationskomplexität der Ordnung $\mathcal{O}(k^2)$.

[10] Wir stellen also keine stärkeren Schutzziele, wie *Robustheit* oder *Verfügbarkeit*, an die Implementierung, da diese Forderungen in einer asynchronen Umgebung (Netzwerk) kaum erfüllbar sind.

[11] nur Stufe 1 (Square Free) und Stufe 2 (Prime Power Product)

[12] Eine Primzahl p heißt *sicher*, wenn auch $(p-1)/2$ prim ist.

[13] Sowohl der Beweis, daß m_i ein Produkt von genau zwei Primfaktoren ist, als auch die Überprüfung der Bedingung $y \in \mathbb{NQR}^{\circ}_{m_i}$.

[14] siehe `libTMCG/mpz_shash.cc` für deren Implementierung

[15] Diese Designentscheidung ist diskussionswürdig: Prinzipiell sollten kryptographische Schlüssel nur für den vorgesehenen Zweck benutzt werden. Ein paralleler Einsatz für andere Aufgaben kann unerwünschte Seiteneffekte haben, welche die Sicherheitsmaßnahmen des eigentlichen Hauptanwendungsgebiets unterlaufen.

[16] Beim Skatspiel haben wir z. B. $M = 32$ unterschiedliche Typen.

[17] Die Korrektheit wird mit Zero-Knowledge Beweisen gezeigt.

[18] beliebige Permutation

[19] zyklische Verschiebung

Die Betrugswahrscheinlichkeit[20] kann bei t-maliger Wiederholung auf $\mathfrak{p} \leq 2^{-t}$ beschränkt werden. Durch Variation dieses Parameters kann man einen Kompromiß zwischen Sicherheit, Laufzeit und Übertragungsvolumen erreichen.

Dennoch ist die Datenstruktur aus [Sch98] sehr exzessiv im Speicherplatzbedarf und folglich auch im Übertragungsvolumen. Das trifft insbesondere auf Spiele mit großer Teilnehmer- oder Kartentypanzahl zu, weil sich die Matrixkodierung linear in k und logarithmisch in M verhält.

Zur Verbesserung dieser Situation wurde kürzlich eine VTMF-Implementierung [BSm03] vorgestellt. Ihre Sicherheit beruht auf der kryptographischen Annahme, daß die Maskierung (ähnlich dem ElGamal-Verfahren) in einer zyklischen Gruppe $G := \langle g \rangle$ stattfindet, wo das DDH-Problem schwer ist. Für G wird vorerst die in [Bon98] erwähnte Gruppe der quadratischen Reste modulo p verwendet, wobei p hier eine sichere Primzahl sein muß. Weiterhin haben wir bei der Gruppenkonstruktion aus Effizienzgründen [vOW96, RS00] $p \equiv 7 \pmod 8$ gewählt, so daß $g = 2$ ein Erzeuger von $\mathbb{QR}_p$ ist. Dadurch kann bei der modularen Exponentiation erhebliche Rechenzeit gespart werden. Zur eindeutigen Kodierung der Kartentypen dienen uns die M kleinsten Quadratreste modulo p beginnend bei 1. Weil diese Zahlen äußerst kurz sind, ist eine Benutzung eingeschränkter Exponenten [vOW96] zwecks Laufzeitverbesserung u. U. als kritisch [BJN00] anzusehen, weshalb momentan zufällige Werte aus dem gesamten Restklassenring $\mathbb{Z}_{(p-1)/2}$ zum Einsatz kommen. Der Korrektheitsbeweis der Maskierung erfolgt über einen nicht-interaktiven *honest-verifier zero-knowledge Proof of Knowledge*, welcher in diesem Fall die Gleichheit zweier diskreter Logarithmen zu unterschiedlichen Basen zeigt. Die in der Originalarbeit [BSm03] angegebene Konstruktion nach Chaum und Pedersen [CP92] wurde durch eine effizientere Variante [CS97] ersetzt. Die Vermeidung der Interaktivität bedingt auch hier die zusätzliche Voraussetzung, daß sich die Hashfunktion h [5] wie ein Zufallsorakel (ROM) verhält.

Ein Vorteil der resultierenden Datenstruktur ist ihre Unabhängigkeit von k und M (siehe Vergleich in Tabelle 1). Trotzdem hat in beiden Kodierungsschemata auch der Sicherheitsparameter ℓ (zugrundeliegende kryptographische Annahme) einen großen Einfluß. Durch Kryptographie über elliptischen Kurven (ECC)[21] besteht auch hier noch erhebliches Potential zur Reduzierung des Übertragungsvolumens.

	Kartenkodierung nach		
	[Sch98]	[BSm03]	[BSm03]
kryptograph. Annahme	QRP	DDH	$\mathrm{EC\text{-}DDH}$
Bitkomplexität für eine Karte (allgemein)	$k \lceil \log_2 M \rceil \ell$	2ℓ	2ℓ
Bitkomplexität für eine Karte beim Skatspiel ($k = 3, M = 32, \ell \in \{1024, 160\}$)	15360	2048	322
LibTMCG Unterstützung	+	+	−

Tabelle 1: Vergleich der Kartenkodierungen

Die Bibliothek LibTMCG

Die freie[22] C++-Bibliothek LibTMCG implementiert die Konzepte der vorangegangenen Abschnitte und stellt damit eine ideale Plattform für sichere mentale Kartenspiele dar. Mittlerweile ist sie ein eigenständiges Projekt [11] bei GNU/Savannah und umfaßt ca. 10 000 Quelltextzeilen.

Unsere Bibliothek hängt von zwei freien Bibliotheken der GNU-Familie ab, d. h. sie nutzt deren Funktionalität:

`libgmp` [6] stellt uns alle notwendigen Operationen für die Arithmetik mit beliebig langen Zahlen bereit. Auch komplizierte Funktionen, wie das Jacobi-Legendre-Symbol, sind vorhanden und zudem effizient umgesetzt. Wir benötigen eine Version $\geq$ 4.1.

`libgcrypt` [7] bietet uns kryptographische Primitiven, von denen wir insbesondere die Hashfunktion RIPEMD-160 [5] und einige Routinen zur Erzeugung sicherer Pseudozufallzahlen verwenden. Diese Bibliothek benötigen wir in einer Version $\geq$ 1.2.0.

Beide Softwareprojekte wurden für viele Rechnerarchitekturen optimiert und sind unter diversen Betriebssystemen lauffähig. Sie stehen damit der Portierung auf andere Plattformen nicht entgegen.

Hauptbestandteil der Kartenspielbibliothek LibTMCG sind die Klassen `SchindelhauerTMCG` [Sch98] und `BarnettSmartVTMF_dlog` [BSm03], welche die Protokolle der erwähnten Arbeiten implementieren. Außerdem gibt es folgende vier wichtige Datenstrukturen:

`TMCG_Card` repräsentiert eine Spielkarte gemäß der Kodierung von Schindelhauer [Sch98].

`TMCG_CardSecret` stellt das zugehörige Geheimnis dar, welches zum Verdecken (Maskieren) einer Karte in diesem Schema notwendig ist.

`VTMF_Card` repräsentiert eine Spielkarte in der verbesserten Kodierung von Barnett und Smart [BSm03].

`VTMF_CardSecret` stellt das zugehörige Geheimnis dar, welches zum Maskieren in diesem Schema notwendig ist.

Die Datentypen für Stapel sind generisch (über Klassentemplates) umgesetzt, so daß sie für beide Karten- bzw. Geheimnisarten funktionieren:

`TMCG_OpenStack<CardType>` repräsentiert einen Stapel (offener) Karten der Datenstruktur `CardType`. Dieser Kontainer enthält Paare (`pair<size_t, CardType>`) bei denen die erste Komponente den Typ der korrespondierenden Karte (zweite Komponente) darstellt.

`TMCG_Stack<CardType>` repräsentiert einen Stapel (verdeckter) Karten der Datenstruktur `CardType`. Dieser Kontainer enthält die Karten und nur implizit deren Typ.

[20] in unserem Angriffsmodell (*honest-but-curious*) und unter den getroffenen kryptographischen Annahmen

[21] Beispielsweise könnte G als Gruppe der Punkte einer elliptischen Kurve über dem Körper $\mathbb{F}_p$ gewählt werden, sofern die Gruppenordnung selbst wieder prim ist. Das DDH-Problem scheint dann ebenfalls schwer zu sein. [Bon98]

[22] GNU General Public License, Version 2, June 1991

`TMCG_StackSecret<CardSecretType>` repräsentiert ein Stapelgeheimnis, welches bei einigen Stapeloperationen (z. B. Mischen, Abheben) notwendig ist. Dieser Kontainer enthält Paare (`pair<size_t, CardSecretType>`) bei denen die erste Komponente den Permutationsindex des korrespondierenden Geheimnis (zweite Komponente) angibt.

Weiterhin gibt es noch drei Schlüsselstrukturen, die allerdings nur für die ursprüngliche Kodierung [Sch98] oder asymmetrische Kryptographie (Verschlüsselung oder Signatur nach dem Rabin-Verfahren) benötigt werden:

`TMCG_SecretKey` repräsentiert einen geheimen Schlüssel. Der persistente Inhalt besteht aus einem Namen, einer Email-Adresse, dem Tupel (p_i, q_i, m_i, y_i), einem nichtinteraktiven Zero-Knowledge Beweis, daß $m_i = p_i \cdot q_i \wedge y_i \in \mathbb{NQR}^{\circ}_{m_i}$ gilt, und der Signatur für diese Daten.

`TMCG_PublicKey` steht für einen öffentlichen Schlüssel. Der Inhalt ist wie beim geheimen Schlüssel aufgebaut, mit einer Ausnahme: Die geheimen Primfaktoren p_i, q_i sind natürlich nicht im Tupel enthalten.

`TMCG_PublicKeyRing` repräsentiert ein dynamisches Feld (Kontainer) der öffentlichen Schlüssel aller Mitspieler, d. h. $((\cdots, m_1, y_1), (\cdots, m_2, y_2), \ldots, (\cdots, m_k, y_k))$.

Die Verwendung der Klassen und Datenstrukturen werden wir nun an einem kleinen Beispiel (siehe Quelltexte ab den Seiten 10 und 16) illustrieren:

> Bob und seine Ex-Freundin Alice wollen das bekannte Kartenspiel *Schwarzer Peter*[23] spielen. Der Gewinner soll den gemeinsam gekauften japanischen Kleinwagen der Marke *Reisschüssel* erhalten – es ist also davon auszugehen, daß beide das Spiel (heimlich) zu ihren Gunsten beeinflussen möchten. Alice und Bob wohnen mittlerweile hunderte Kilometer entfernt und können nur über das weltweite Datennetz kommunizieren. Außerdem ist keine Person bekannt, die beiden hinreichend vertrauenswürdig erscheint, in dieser delikaten Angelegenheit als Vermittler zu agieren. Zum Glück haben sie in einem Artikel der Zeitschrift „Offene Systeme“ von einer freien Bibliothek gelesen, die genau für solche Aufgaben entwickelt wurde.

Unser Beispiel setzt die effizientere Kartenkodierung nach [BSm03] ein, d. h. wir verwenden nur die Strukturen `VTMF_Card`, `VTMF_CardSecret` und zusätzlich die Klasse `BarnettSmartVTMF_dlog`. Alle notwendigen Datentypen stehen durch Einbinden der Datei `libTMCG.hh` im globalen Namensraum bereit. Die Kommunikation der beiden Programme erfolgt der Einfachheit halber über die Standardein- bzw. -ausgabe. Alice und Bob könnten die entsprechenden Datenströme umlenken und mit `netcat` oder `ssh` weiterleiten.

Zuerst werden die beiden Klasseninstanzen durch Konstruktoraufruf erzeugt (Zeile 10 und 13). Dabei sind verschiedene Argumente von Bedeutung:

[23] Der Beweis, daß dieses Spiel bei gleichverteilten Mischen immer terminiert, bleibt dem Leser als Übungsaufgabe überlassen.

		OpenSkat ≤ 1.9	SecureSkat [13]
Sicherheitsparameter t	Betrugswahrscheinlichkeit $\mathtt{p} \leq 2^{-t}$	Übertragungsvolumen pro Spiel und Spieler mit $\ell = 1024$ und Schema [Sch98]	Übertragungsvolumen pro Spiel und Spieler mit $\ell = 1024$ und Schema [BSm03]
2	$\leq$ 0,25	$\approx$ 3 MByte	$\approx$ 0,64 MByte
4	$\leq$ 0,0625	$\approx$ 5 MByte	$\approx$ 0,72 MByte
8	$\leq$ 0,00390625	$\approx$ 10 MByte	$\approx$ 0,8 MByte
16	$\leq$ 0,00001526	$\approx$ 20 MByte	$\approx$ 1,02 MByte
32	$\leq 2.33 \cdot 10^{-10}$	$\approx$ 40 MByte	$\approx$ 1.42 MByte
64	$\leq 5.43 \cdot 10^{-20}$	$\approx$ 80 MByte	$\approx$ 2.18 MByte

Tabelle 2: Übertragungsvolumen versus Sicherheit

`t` ist der Sicherheitsparameter, welcher die Betrugswahrscheinlichkeit `p` nach oben beschränkt, d. h. $\mathtt{p} \leq 2^{-\mathtt{t}}$.

`k` ist die Anzahl der Mitspieler.

`w` ist die Anzahl der Bits, die zur Kodierung der M verschiedenen Kartentypen notwendig ist, d. h. $\mathtt{w} = \lceil \log_2 M \rceil$.

Am Aufruf des `BarnettSmartVTMF_dlog`-Konstruktors erkennt man, daß Alice die Gruppenparameter für G (DDH-schwer) erzeugt und später (Zeile 21) an Bob sendet. Da diese Parameter in beiden Programmen geprüft werden, stellt ein solches Vorgehen kein Sicherheitsproblem dar. In der Praxis könnten wir sogar öffentlich verfügbare Parameterwerte verwenden, sofern sie unseren Optimierungs- und Kodierungsbedingungen[24] genügen.

Der weitere Ablauf ist im Quelltext ausführlich dokumentiert und sollte deshalb leicht verständlich sein.

Referenzimplementierung SecureSkat

Im Frühjahr 2002 begann ich mit der Arbeit an diesem Projekt. Anfangs fand die Entwicklung unter dem Namen *OpenSkat* statt, später erfolgte dann die Umbenennung in *SecureSkat* (Details: siehe graue Box) und der Transfer nach GNU/Savannah [13]. Der Quelltext umfaßt ca. 9 400 Zeilen C++/C. Die letzte stabile Version unter altem Namen war OpenSkat 2.4 [12], welches auch eine graphische Spieloberfläche auf Grundlage von XSkat [10] enthielt. Im neuen Projekt mußte sie entfernt werden.[25]

Prinzipiell ist *SecureSkat* eine sichere, dezentrale Implementierung des Kartenspiels Skat [9], wobei sicher in diesem Kontext bedeutet, daß die Betrugswahrscheinlichkeit bei Karten- und Stapeloperationen durch einen Sicherheitsparameter t begrenzt werden kann. Dazu bedient sich die Software der Konzepte aus [Sch98, BSm03] und benutzt mittlerweile direkt die Bibliothek `libTMCG`. Die Wahl von t hat erheblichen Einfluß auf Übertragungsvolumen und Laufzeit (siehe Tabelle 2).

[24] Beispielsweise muß $g = 2$ ein Erzeuger von $\mathbb{QR}_p$ sein, wobei p eine sichere Primzahl ist.

[25] Hinweis: Das Programm `openSkat_gui` (z. B. im Archiv `openSkat-2.4.tar.gz` enthalten) kann auch mit SecureSkat benutzt werden, denn alle Schnittstellen sind kompatibel.

OpenSkat versus SecureSkat

Vor langer, langer Zeit trug dieses Projekt mal den Namen "OpenSkat". Im Rahmen eines Umzugs in die GNU-Savanne wurde von den dortigen Parkwächtern aber eine Namensänderung gewünscht.

> *In the other hand can you look for another name to your project?*
>
> > Do you have any suggestions? 'FreeSkat' sounds a little bit strange and may be in some sense misleading.
>
> *This is because we supports projects of the Free Software movement, not projects of the Open Source movement.*
>
> > I understand your concerns. However, in this case the prefix 'open' should be considered more from a cryptographical rather than from philosophical point of view. Nevertheless, if the community has a better name suggestion ...
>
> *We are careful about ethical issues and insist on producing software that is not dependent on proprietary software.*
>
> *While Open Source as defined by its founders means something pretty close to Free Software, it's frequently misunderstood.*

Trotz aller Erklärungsversuche konnte der alte Name nicht gerettet werden und ward somit auf immer verloren. Schon drohte neues Unheil am Horizont:

> *You have to remove the 'xskat' dependence, because the license of this project is a GPL-Incompatible, if you remove this dependence we can accept the project in Savannah.*
>
> *I don't have suggestions for a new name, but you can try with FreeSkat or even with a name without the 'Skat' word on it.*
>
> *As you explain it, the 'open' word should not be considered from the philosophical point of view, but if we do not make the difference anyone will see your project thinking that has more relation with the Open Source movement rather than the Free Software movement.*

To be continued in a free world ...

SecureSkat: Anonymer CVS Zugriff

```
export CVS_RSH="ssh"
cvs -z3 -d:ext:anoncvs@savannah.gnu.org:\
              /cvsroot/secureskat co secureskat
```

Spielaushandlung und Protokollsteuerung erfolgen über den Kanal `#openSkat`, der auf einem beliebigen IRC-Server [8] (z. B. `gaos.org`) angesiedelt sein kann. Für die Beweise werden jedoch zu Beginn der Sitzung eigene Verbindungen[26] zwischen den Teilnehmern aufgebaut.[27] Eine genaue Beschreibung der Installation und Benutzung von SecureSkat liefert das obligatorische `README` [13].

TODO: Noch nicht (bzw. nur teilweise) fertiggestellt sind

- Internationalisierung mit GNU gettext [14],
- Dokumentation und Fehlerbehandlung,
- freie graphische Spieloberfläche.

Helfende Mitarbeit, sei es durch Testen der Software, Melden von Fehlern, oder eigene Verbesserungen am Quelltext[28], ist jederzeit willkommen. Auch die sichere Umsetzung anderer Kartenspiele mit Hilfe der Bibliothek LibTMCG wäre ein sehr wünschenswertes Ziel.

Zum Abschluß noch eine grundsätzliche Bemerkung: Die in diesem Artikel vorgestellte Software trägt zwar den Namen SecureSkat, ist aber keineswegs *perfekt sicher*. Jedes hinreichend komplexe (von Menschen geschaffene) Objekt enthält notwendigerweise mehr oder weniger Fehler. Im Quelltext von OpenSkat, dem Vorgänger von SecureSkat, fand ich bei bisherigen Kontrollen sogar besonders schwere Fehler. Beispielsweise erlaubte eine fehlende Festlegung in einem Zero-Knowledge Beweis die triviale Faktorisierung des Moduls m_i mittels Euklidschen Algorithmus. Daraus resultierte natürlich die totale Unsicherheit des Programms bezüglich seiner eigentlichen Aufgabe – dem sicheren mentalen Kartenspiel. Weitere, hoffentlich nicht ganz so kritische, Fehler sind wahrscheinlich enthalten und können nur durch gewissenhafte Begutachtung gefunden werden. Trotz aller Versuche, *vollständige Sicherheit* kann und wird es nie geben!

Zusammenfassung

In diesem Artikel wurden die Grundlagen sicherer mentaler Kartenspiele besprochen und eine freie Bibliothek zu ihrer Implementierung vorgestellt. Beide Projekte sind sogenannte *Freie Software* und können deshalb im Rahmen der GNU General Public License auch verändert werden.

Revision. Dieser Artikel wurde ursprünglich in der Ausgabe 4 (2004) der Zeitschrift "Offene Systeme" abgedruckt. Mittlerweile sind kleinere Korrekturen und einige inhaltliche Änderungen eingeflossen.

Referenzen

[Sch98] Christian Schindelhauer: *Toolbox for Mental Card Games*, Technical Report A-98-14, Universität Lübeck, 1998

[26] authentifiziert, verschlüsselt und komprimiert

[27] Voraussetzung: keine paranoid konfigurierten Feuerwände

[28] Große Teile von SecureSkat wurden schnell (d. h. fast immer unelegant) entwickelt und bedürfen dringend einer Verbesserung.

[BSm03] Adam Barnett, Nigel P. Smart: *Mental Poker Revisited*, In K.G. Paterson (Ed.): Cryptography and Coding 2003, LNCS 2898, pp. 370–383, 2003

[Bon98] Dan Boneh: *The Decision Diffie-Hellman Problem*, In Proceedings of the Third Algorithmic Number Theory Symposium, LNCS 1423, pp. 48–63, 1998

[GMR98] Rosario Gennaro, Daniele Micciancio, Tal Rabin: *An Efficient Non-Interactive Statistical Zero-Knowledge Proof System for Quasi-Safe Prime Products*, In Proceedings of the 5th ACM Conference on Computer and Communications Security, pp. 67–72, 1998

[BR93] Mihir Bellare, Phillip Rogaway: *Random oracles are practical: A paradigm for designing efficient protocols*, In Proceedings First Annual ACM Conference on Computer and Communications Security, pp. 62–73, 1993

[FS86] Amos Fiat, Adi Shamir: *How To Prove Yourself: Practical Solutions to Identification and Signature Problems*, In Advances in Cryptology: CRYPTO '86 Proceedings, LNCS 263, pp. 186-194, 1987

[BFM88] Manuel Blum, Paul Feldman, Silvio Micali: *Non-interactive zero-knowledge and its applications*, In Proceedings of the 20th Annual ACM Symposium on Theory of Computing, pp. 103–112, 1988

[BSG91] Manuel Blum, Alfredo De Santis, Silvio Micali, Giuseppe Persiano: *Non-Interactive Zero Knowledge*, SIAM Journal on Scientific Computing, Volume 20(6), pp. 1084–1118, 1991

[Rab79] Michael O. Rabin: *Digitalized Signatures and Public-Key Functions as Intractable as Factorization*, Technical Report MIT-LCS-TR-212, MIT Laboratory for Computer Science, 1979

[BR96] Mihir Bellare, Phillip Rogaway: *The Exact Security of Digital Signatures—How to Sign with RSA and Rabin*, In Advances in Cryptology: EUROCRYPT '96 Proceedings, LNCS 1070, pp. 399–416, 1996

[Bon01] Dan Boneh: *Simplified OAEP for the RSA and Rabin Functions*, In Advances in Cryptology: CRYPTO '2001 Proceedings, LNCS 2139, pp. 275–291, 2001

[vOW96] Paul C. van Oorschot and Michael J. Wiener: *On Diffie-Hellman Key Agreement with Short Exponents*, In Advances in Cryptology: EUROCRYPT '96 Proceedings, LNCS 1070, pp. 332–343, 1996

[RS00] Jean-François Raymond, Anton Stiglic: *Security Issues in the Diffie-Hellman Key Agreement Protocol*, ZKS Technical Report (draft), 2000

[BJN00] Dan Boneh, Antoine Joux, Phong Q. Nguyen: *Why Textbook ElGamal and RSA Encryption are Insecure*, In Proceedings of ASIACRYPT '2000, LNCS 1976, pp. 30–43, 2001

[CP92] David Chaum, Torben P. Pedersen: *Wallet Databases with Observers*, In Advances in Cryptology: CRYPTO '92 Proceedings, LNCS 740, pp. 89-105, 1992

[CS97] Jan Camenisch, Markus Stadler: *Proof Systems for General Statements about Discrete Logarithms*, Technical Report, 1997

[Gut00] Walter Guttmann: *Kartenspielen übers Telephon*, Hauptseminar Sichere Telekommunikation WS 1999/2000, Universität Ulm, 2000

[RSA79] Adi Shamir, Ronald L. Rivest, Leonard M. Adleman: *Mental Poker*, Technical Report MIT-LCS-TM-125, MIT Laboratory for Computer Science, 1979

[Blu81] Manuel Blum: *Coin Flipping by Telephone: A protocol for solving impossible problems*, In CRYPTO '81 Proceedings, pp. 11–15, 1981; and Proceedings of the 24th IEEE Computer Conference, pp. 133–137, 1982

[GM82] Shafi Goldwasser, Silvio Micali: *Probabilistic Encryption and How to Play Mental Poker Hiding All Partial Information*, In Proceedings of the 14th Annual ACM Symposium on the Theory of Computing, pp. 365–377, 1982

[Cré87] Claude Crépeau: *A zero-knowledge poker protocol that achieves confidentiality of the players' strategy or how to achieve an electronic poker face*, In Advances in Cryptology: CRYPTO '86 Proceedings, LNCS 263, pp. 239-247, 1987

[BCR87] Gilles Brassard, Claude Crépeau, Jean-M. Robert: *All-or-nothing disclosure of secrets*, In Advances in Cryptology: CRYPTO '86 Proceedings, LNCS 263, pp. 234–238, 1987

[GMW87] Oded Goldreich, Silvio Micali, Avi Wigderson: *How to Prove All NP-Statements in Zero-Knowledge and a Methodology of Cryptographic Protocol Design*, In Advances in Cryptology: CRYPTO '86 Proceedings, LNCS 263, pp. 171–185, 1987

[QGB89] Jean-J. Quisquater, Louis Guillou, Tom Berson: *How to Explain Zero-Knowledge Protocols to Your Children*, In Advances in Cryptology: CRYPTO '89 Proceedings, LNCS 435, pp. 628–631, 1989

[BCC04] Ernie Brickell, Jan Camenisch, and Liqun Chen: *Direct Anonymous attestation*, In Proceedings of 11th ACM Conference on Computer and Communications Security, 2004

[1] http://catless.ncl.ac.uk/Risks/17.21.html#subj9

[2] http://www.zurich.ibm.com/security/daa/

[3] http://www.crypto.ethz.ch/teaching/lectures/Krypto04/ROM.pdf

[4] http://www.kisa.or.kr/technology/sub1/Rabin.htm

[5] http://www.esat.kuleuven.ac.be/~bosselae/ripemd160.html

[6] GNU Multiple Precision Arithmetic Library, the fastest bignum library on the planet!, http://www.swox.com/gmp/

[7] GNU Crypto Library, http://directory.fsf.org/security/libgcrypt.html

[8] RFC 1459: Internet Relay Chat Protocol, http://www.ietf.org/rfc/rfc1459.txt?number=1459

[9] Offizielle Skatregeln: http://www.dskv.de/

[10] XSkat: http://www.xskat.de/

[11] LibTMCG, http://savannah.nongnu.org/projects/libtmcg/

[12] http://freshmeat.net/projects/openskat/

[13] SecureSkat, http://savannah.nongnu.org/projects/secureskat/

[14] http://www.gnu.org/software/gettext/

[15] http://de.wikipedia.org/wiki/Zyklische_Gruppe

[16] http://de.wikipedia.org/wiki/Restklassenring

[17] http://de.wikipedia.org/wiki/Eulersche_%CF%86-Funktion

Quelltext SchwarzerPeterAlice.cc

```
// include the libTMCG header file
#include <libTMCG.hh>

int main
   ()
{
   if (!init_libTMCG())
   {
      std::cerr << "Initalization of the libTMCG failed!" << std::endl;
      return -1;
   }

   // create an instance of the "Toolbox for Mental Card Games"
   // ---------------------------------------------------------
   // p_cheating <= 2^{-16}, k = 2 players, w = 4 bits (2^4 >= 13 card types)
   size_t t = 16, k = 2, w = 4;
   SchindelhauerTMCG *tmcg = new SchindelhauerTMCG(t, k, w);

   // create an instance of the VTMF implementation (create the group G)
   BarnettSmartVTMF_dlog *vtmf = new BarnettSmartVTMF_dlog();

   // check whether the group G was correctly generated
   if (!vtmf->CheckGroup())
   {
      std::cerr << "Group G was not correctly generated!" << std::endl;
      return -1;
   }
   // send the parameters of the group to Bob (the second party)
   vtmf->PublishGroup(std::cout);
   // create and send the (public) key
   vtmf->KeyGenerationProtocol_GenerateKey();
   vtmf->KeyGenerationProtocol_PublishKey(std::cout);
   // receive Bob's public key and update the VTMF implementation
   if (!vtmf->KeyGenerationProtocol_UpdateKey(std::cin))
   {
      std::cerr << "Bob's public key was not correctly generated!" << std::endl;
      return -1;
   }
   // finish the key generation
   vtmf->KeyGenerationProtocol_Finalize();

   // create a deck of 25 cards (12 pairs and the "Schwarzer Peter")
   // --------------------------------------------------------------
   std::cerr << "Create the deck ..." << std::endl;
   TMCG_OpenStack<VTMF_Card> deck;
   for (size_t i = 0; i < 13; i++)
   {
      for (size_t j = 0; j < 2; (i != 0) ? j++ : j = 2)
      {
         VTMF_Card c;
         tmcg->TMCG_CreateOpenCard(c, vtmf, i); // create a card of type i
         deck.push(i, c);      // push this card to the open stack deck
```

```
      }
   }

   // shuffle the deck: Alice first, after it Bob.
   // ---------------------------------------
   std::cerr << "Shuffle the deck ..." << std::endl;
   TMCG_Stack<VTMF_Card> stack, stack_Alice, stack_Bob;
   TMCG_StackSecret<VTMF_CardSecret> secret;
   stack.push(deck); // push the whole deck to the working stack
   // create the secret for a full shuffle (permutation) of the stack
   tmcg->TMCG_CreateStackSecret(secret, false, stack.size(), vtmf);

   // ... Alice
   tmcg->TMCG_MixStack(stack, stack_Alice, secret, vtmf); // shuffle operation
   std::cout << stack_Alice << std::endl; // send the result to Bob
   tmcg->TMCG_ProveStackEquality(stack, stack_Alice, secret, false, vtmf,
      std::cin, std::cout); // prove the correctness of the operation

   // ... Bob
   std::cin >> stack_Bob;
   if (!std::cin.good())
   {
      std::cerr << "Stack corrupted!" << std::endl;
      return -1;
   }
   if (!tmcg->TMCG_VerifyStackEquality(stack_Alice, stack_Bob, false, vtmf,
      std::cin, std::cout)) // verify the proof of correctness
   {
      std::cerr << "StackEquality: proof of correctness failed!" << std::endl;
      return -1;
   }

   // dealing: Alice gets the 1st to 13th and Bob the remaining cards.
   // ---------------------------------------------------------------
   std::cerr << "Deal the cards ..." << std::endl;
   TMCG_Stack<VTMF_Card> hand_Alice, hand_Bob;
   for (size_t i = 0; i < 13 ; i++)
      hand_Alice.push(stack_Bob[i]);
   for (size_t i = 13; i < 25 ; i++)
      hand_Bob.push(stack_Bob[i]);

   // The real game proceeds like an endless loop.
   // ---------------------------------------
   while (1)
   {
      // reveal the card type to the owner: Alice first, after it Bob.
      // ------------------------------------------------------------
      // The three operations SelfCardSecret(), VerifyCardSecret() and
      // TypeOfCard() MUST be called exactly in this order!
      TMCG_OpenStack<VTMF_Card> hand;
      size_t type = 0;

      // ... Alice
      std::cerr << "My cards are: ";
      for (size_t i = 0; i < hand_Alice.size(); i++)
      {
         tmcg->TMCG_SelfCardSecret(hand_Alice[i], vtmf);
```

```
110            if (!tmcg->TMCG_VerifyCardSecret(hand_Alice[i], vtmf,
                  std::cin, std::cout))
               {
                  std::cerr << "CardSecret: proof of correctness failed!" << std::endl;
                  return -1;
115            }
               type = tmcg->TMCG_TypeOfCard(hand_Alice[i], vtmf);
               hand.push(type, hand_Alice[i]);
               std::cerr << type << " ";
            }
120         std::cerr << std::endl;

            // ... Bob
            for (size_t i = 0; i < hand_Bob.size(); i++)
               tmcg->TMCG_ProveCardSecret(hand_Bob[i], vtmf, std::cin, std::cout);
125
            // publish and draw a pair, if possible: Bob first, after it Alice.
            // ---------------------------------------------------------------
            size_t pairs = 0, lasttype = 0;

130         // ... Bob
            std::cin >> pairs;
            std::cin.ignore(1, '\n'); // reject the newline
            if (pairs > 1)
            {
135            std::cerr << "Bob wants to reveal more than one pair!" << std::endl;
               return -1;
            }
            if (pairs)
            {
140            std::cerr << "Bob reveals: ";
               for (size_t i = 0; i < 2; i++)
               {
                  VTMF_Card c;

145               std::cin >> c;
                  if (!std::cin.good())
                  {
                     std::cerr << "Card corrupted!" << std::endl;
                     return -1;
150               }
                  // Check whether the card is in the stack.
                  if (!hand_Bob.find(c))
                  {
                     std::cerr << "Bob does not own this card!" << std::endl;
155                  return -1;
                  }
                  // Reveal the card and verify the proof of correctness.
                  tmcg->TMCG_SelfCardSecret(c, vtmf);
                  if (!tmcg->TMCG_VerifyCardSecret(c, vtmf, std::cin, std::cout))
160               {
                     std::cerr << "CardSecret: proof of correctness failed!" << std::endl;
                     return -1;
                  }
                  type = tmcg->TMCG_TypeOfCard(c, vtmf);
165               // Check whether it is really a pair.
                  if (i % 2)
```

```
            {
               if (lasttype != type)
               {
170               std::cerr << "Bob reveals no pair!" << std::endl;
                  return -1;
               }
               else
                  std::cerr << type << " ";
175         }
            else
               lasttype = type;
            hand_Bob.remove(c); // remove the card from Bob's hand
         }
180      std::cerr << std::endl;
      }

      // .. Alice
      TMCG_OpenStack<VTMF_Card> pairstack;
185   // search for pairs
      for (size_t i = 0; i < hand.size(); i++)
      {
         for (size_t j = 0; j < hand.size(); j++)
         {
190         // pair found?
            if ((i < j) && (hand[i].first == hand[j].first))
            {
               pairstack.push(hand[i]), pairstack.push(hand[j]);
               i = hand.size(); // break the search
195            break;
            }
         }
      }
      // Send the number of pairs to Bob.
200   std::cout << (pairstack.size() / 2) << std::endl;
      // Reveal the pairs, prove the correctness and remove them from our hand.
      if (pairstack.size() > 0)
      {
         std::cerr << "My pairs to reveal: ";
205      for (size_t i = 0; i < pairstack.size(); i++)
         {
            std::cout << pairstack[i].second << std::endl;
            tmcg->TMCG_ProveCardSecret(pairstack[i].second, vtmf,
               std::cin, std::cout);
210         hand_Alice.remove(pairstack[i].second);
            hand.remove(pairstack[i].first);
            if (i % 2)
               std::cerr << pairstack[i].first << " ";
         }
215      std::cerr << std::endl;
      }

      // Cleanup and check the game outcome.
      // -----------------------------------
220   pairstack.clear(), stack_Alice.clear(), stack_Bob.clear();
      if (hand_Alice.size() == 0)
      {
         std::cerr << "You win the game!" << std::endl;
```

```
            break;
225         }
            if (hand_Bob.size() == 0)
            {
               std::cerr << "><> You loose. Zonk! ('Schwarzer Peter')" << std::endl;
               break;
230         }

            // Draw a private card from the opponent: Only the player who have
            // fewer cards will draw. After the draw the hand is shuffled again.
            // ---------------------------------------------------------------
235         size_t position;
            std::vector<bool> who;
            VTMF_Card c;

            if (hand_Alice.size() > hand_Bob.size())
240            who.push_back(false);
            else
               who.push_back(true);

            for (size_t i = 0; i < who.size(); i++)
245         {
               if (who[i])
               {
                  // ... Alice
                  position = mpz_srandom_ui() % hand_Bob.size();
250               std::cout << position << std::endl;
                  c = hand_Bob[position]; // draw a card
                  hand_Bob.remove(c); // remove it
                  tmcg->TMCG_SelfCardSecret(c, vtmf); // reveal it privatly
                  if (!tmcg->TMCG_VerifyCardSecret(c, vtmf, std::cin, std::cout))
255               {
                     std::cerr << "CardSecret: proof of correctness failed!" << std::endl;
                     return -1;
                  }
                  type = tmcg->TMCG_TypeOfCard(c, vtmf);
260               std::cerr << "Alice draws the card (from position " << position <<
                     "): " << type;
                  if (type)
                     std::cerr << std::endl;
                  else
265                  std::cerr << " (Zonk!)" << std::endl;
                  hand_Alice.push(c); // push it to Alice's hand
                  // shuffle, because Bob must not know the position of the drawed card
                  tmcg->TMCG_CreateStackSecret(secret, false, hand_Alice.size(), vtmf);
                  tmcg->TMCG_MixStack(hand_Alice, stack_Alice, secret, vtmf);
270               std::cout << stack_Alice << std::endl; // send the result to Bob
                  tmcg->TMCG_ProveStackEquality(hand_Alice, stack_Alice, secret, false,
                     vtmf, std::cin, std::cout);
                  hand_Alice = stack_Alice;
               }
275            else
               {
                  // ... Bob
                  std::cin >> position;
                  std::cin.ignore(1, '\n'); // reject the newline
280               if (position >= hand_Alice.size())
```

```
                {
                   std::cerr << "Bob wants to draw from a wrong position!" << std::endl;
                   return -1;
                }
285             c = hand_Alice[position]; // draw a card
                hand_Alice.remove(c); // remove it
                tmcg->TMCG_ProveCardSecret(c, vtmf, std::cin, std::cout); // revealing
                hand_Bob.push(c); // push it to Bob's hand
                // shuffle, because Alice must not know the position of the drawed card
290             std::cin >> stack_Bob;
                if (!std::cin.good())
                {
                   std::cerr << "Stack corrupted!" << std::endl;
                   return -1;
295             }
                if (!tmcg->TMCG_VerifyStackEquality(hand_Bob, stack_Bob, false, vtmf,
                   std::cin, std::cout)) // verify the proof of correctness
                {
                   std::cerr << "StackEquality: proof of correctness failed!" << std::endl;
300                return -1;
                }
                hand_Bob = stack_Bob;
             }
          }
305    }

       // final cleanup
       delete vtmf, delete tmcg;
    }
```

Quelltext SchwarzerPeterBob.cc

```
// Einbinden der LibTMCG Kopfdatei
#include <libTMCG.hh>

int main
   ()
{
   if (!init_libTMCG())
   {
      std::cerr << "Initalization of libTMCG failed!" << std::endl;
      return -1;
   }

   // create an instance of the "Toolbox for Mental Card Games"
   // ---------------------------------------------------------
   // p_cheating <= 2^{-16}, k = 2 players, w = 4 bits (2^4 >= 13 card types)
   size_t t = 16, k = 2, w = 4;
   SchindelhauerTMCG *tmcg = new SchindelhauerTMCG(t, k, w);

   // create an instance of the VTMF implementation (receive the group G)
   BarnettSmartVTMF_dlog *vtmf = new BarnettSmartVTMF_dlog(std::cin);
   // check whether the group G was correctly generated
   if (!vtmf->CheckGroup())
   {
      std::cerr << "Group G was not correctly generated!" << std::endl;
      return -1;
   }
   // create and send the (public) key
   vtmf->KeyGenerationProtocol_GenerateKey();
   vtmf->KeyGenerationProtocol_PublishKey(std::cout);
   // receive Alice's public key and update the VTMF implementation
   if (!vtmf->KeyGenerationProtocol_UpdateKey(std::cin))
   {
      std::cerr << "Alice's public key was not correctly generated!" << std::endl;
      return -1;
   }
   // finish the key generation
   vtmf->KeyGenerationProtocol_Finalize();

   // create a deck of 25 cards (12 pairs and the "Schwarzer Peter")
   // --------------------------------------------------------------
   std::cerr << "Create the deck ..." << std::endl;
   TMCG_OpenStack<VTMF_Card> deck;
   for (size_t i = 0; i < 13; i++)
   {
      for (size_t j = 0; j < 2; (i != 0) ? j++ : j = 2)
      {
         VTMF_Card c;
         tmcg->TMCG_CreateOpenCard(c, vtmf, i); // create a card of type i
         deck.push(i, c);      // push this card to the open stack deck
      }
   }
```

```
        // Anfangsstapel mischen: Alice zuerst, dann Bob.
        // ----------------------------------------------
 55     std::cerr << "Anfangsstapel mischen ..." << std::endl;
        TMCG_Stack<VTMF_Card> Mischstapel, Mischstapel_Alice, Mischstapel_Bob;
        TMCG_StackSecret<VTMF_CardSecret> Mischgeheimnis;
        Mischstapel.push(deck); // offenen in allgemeinen Stapel umwandeln
        tmcg->TMCG_CreateStackSecret(Mischgeheimnis, false, // volle Permutation
 60        Mischstapel.size(), vtmf);

        // ... Alice
        std::cin >> Mischstapel_Alice;
        if (!std::cin.good())
 65     {
           std::cerr << "><  Stapelformat falsch (Trollversuch?)" << std::endl;
           return -1;
        }
        if (!tmcg->TMCG_VerifyStackEquality(Mischstapel, Mischstapel_Alice,
 70        false, vtmf, std::cin, std::cout))
        {
           std::cerr << "><  Stapelbeweis falsch (Betrugsversuch?)" << std::endl;
           return -1;
        }
 75     // ... Bob
        tmcg->TMCG_MixStack(Mischstapel_Alice, Mischstapel_Bob,
           Mischgeheimnis, vtmf); // Maskieren
        std::cout << Mischstapel_Bob << std::endl; // Stapel an Alice senden
        tmcg->TMCG_ProveStackEquality(Mischstapel_Alice, Mischstapel_Bob,
 80        Mischgeheimnis, false, vtmf, std::cin, std::cout); // Korrektheit beweisen

        // Stapel teilen: Alice erhält die Karten 1 bis 13 und Bob den Rest.
        // -----------------------------------------------------------------
        std::cerr << "Stapel teilen ..." << std::endl;
 85     TMCG_Stack<VTMF_Card> Kartenstapel_Alice, Kartenstapel_Bob;
        for (size_t i = 0; i < 13 ; i++)
           Kartenstapel_Alice.push(Mischstapel_Bob[i]);
        for (size_t i = 13; i < 25 ; i++)
           Kartenstapel_Bob.push(Mischstapel_Bob[i]);
 90
        // Das eigentliche Spiel läuft in einer (Endlos-)Schleife.
        // -------------------------------------------------------
        while (1)
        {
 95        // Handkarten privat aufdecken: Alice zuerst, dann Bob.
           // ----------------------------------------------------
           // Die drei Operationen SelfCardSecret, VerifyCardSecret und
           // TypeOfCard müssen in *genau* dieser Reihenfolge aufgerufen werden!
           TMCG_OpenStack<VTMF_Card> Handkarten;
100        size_t Typ = 0;

           // ... Alice
           for (size_t i = 0; i < Kartenstapel_Alice.size(); i++)
           {
105           tmcg->TMCG_ProveCardSecret(Kartenstapel_Alice[i], vtmf,
                 std::cin, std::cout);
           }

           // ... Bob
```

```
110         std::cerr << "Meine Karten: ";
            for (size_t i = 0; i < Kartenstapel_Bob.size(); i++)
            {
               tmcg->TMCG_SelfCardSecret(Kartenstapel_Bob[i], vtmf);
               if (!tmcg->TMCG_VerifyCardSecret(Kartenstapel_Bob[i], vtmf,
115               std::cin, std::cout))
               {
                  std::cerr << "><  Öffnungsbeweis falsch (Betrugsversuch?)" <<
                     std::endl;
                  return -1;
120            }
               Typ = tmcg->TMCG_TypeOfCard(Kartenstapel_Bob[i], vtmf);
               Handkarten.push(Typ, Kartenstapel_Bob[i]);
               std::cerr << Typ << " ";
            }
125         std::cerr << std::endl;

            // Ein Paar offen ablegen (sofern möglich): Bob zuerst, dann Alice.
            // ----------------------------------------------------------------
            TMCG_OpenStack<VTMF_Card> Paarstapel;
130         size_t Paare = 0, LetzterTyp = 0;

            // ... Bob
            // Paare suchen
            for (size_t i = 0; i < Handkarten.size(); i++)
135         {
               for (size_t j = 0; j < Handkarten.size(); j++)
               {
                  // Paar gefunden?
                  if ((i < j) &&
140                  (Handkarten[i].first == Handkarten[j].first))
                  {
                     Paarstapel.push(Handkarten[i]);
                     Paarstapel.push(Handkarten[j]);
                     i = Handkarten.size();          // keine weiteren Paare suchen
145                  break;
                  }
               }
            }
            // Anzahl senden, Paare aufdecken und entfernen
150         std::cout << (Paarstapel.size() / 2) << std::endl;
            if (Paarstapel.size())
            {
               std::cerr << "Ich lege ab: ";
               for (size_t i = 0; i < Paarstapel.size(); i++)
155            {
                  std::cout << Paarstapel[i].second << std::endl;
                  tmcg->TMCG_ProveCardSecret(Paarstapel[i].second, vtmf,
                     std::cin, std::cout);
                  Kartenstapel_Bob.remove(Paarstapel[i].second);
160               Handkarten.remove(Paarstapel[i].first);
                  if (i % 2)
                     std::cerr << Paarstapel[i].first << " ";
               }
               std::cerr << std::endl;
165         }
```

```
            // ... Alice
            std::cin >> Paare;
            std::cin.ignore(1, '\n'); // Newline verwerfen
170         if (Paare > 1)
            {
               std::cerr << "><  Unerlaubte Paaranzahl (Trollversuch?)" << std::endl;
               return -1;
            }
175         if (Paare)
            {
               std::cerr << "Gegner legt ab: ";
               for (size_t i = 0; i < 2; i++)
               {
180               VTMF_Card c;

                  std::cin >> c;
                  if (!std::cin.good())
                  {
185                  std::cerr << "><  Kartenformat falsch (Trollversuch?)" <<
                        std::endl;
                     return -1;
                  }
                  // Prüfen, ob Karte im Stapel ist
190               if (!Kartenstapel_Alice.find(c))
                  {
                     std::cerr << "><  Karte nicht vorhanden (Betrugsversuch?)" <<
                        std::endl;
                     return -1;
195               }
                  // Karte aufdecken und Korrektheit prüfen
                  tmcg->TMCG_SelfCardSecret(c, vtmf);
                  if (!tmcg->TMCG_VerifyCardSecret(c, vtmf, std::cin, std::cout))
                  {
200                  std::cerr << "><  Öffnungsbeweis falsch (Betrugsversuch?)" <<
                        std::endl;
                     return -1;
                  }
                  Typ = tmcg->TMCG_TypeOfCard(c, vtmf);
205               // Prüfen, ob wirklich ein Paar vorliegt
                  if (i % 2)
                  {
                     if (LetzterTyp != Typ)
                     {
210                     std::cerr << "><  Kein Paar (Trollversuch?)" << std::endl;
                        return -1;
                     }
                     else
                        std::cerr << Typ << " ";
215               }
                  else
                     LetzterTyp = Typ;
                  // Karte entfernen
                  Kartenstapel_Alice.remove(c);
220            }
               std::cerr << std::endl;
            }
```

```
        // Aufräumen und Abbruchkriterium prüfen
225     // -------------------------------------
        Paarstapel.clear();
        Mischstapel_Alice.clear(), Mischstapel_Bob.clear();
        if (Kartenstapel_Alice.size() == 0)
        {
230        std::cerr << "><> Zonk! ('Schwarzer Peter')" << std::endl;
           break;
        }
        if (Kartenstapel_Bob.size() == 0)
        {
235        std::cerr << "><> Glück gehabt!" << std::endl;
           break;
        }

        // Eine Karte beim Gegner ziehen: Der mit weniger Karten zieht.
240     // Nach dem Ziehen wird neu gemischt und die Korrektheit gezeigt.
        // -------------------------------------------------------------
        size_t WelchePosition;
        std::vector<bool> Ablauf;
        VTMF_Card c;
245
        if (Kartenstapel_Alice.size() > Kartenstapel_Bob.size())
           Ablauf.push_back(true);
        else
           Ablauf.push_back(false);
250
        for (size_t i = 0; i < Ablauf.size(); i++)
        {
           if (Ablauf[i])
           {
255           // Bob zieht ...
              WelchePosition = mpz_srandom_ui() % Kartenstapel_Alice.size();
              std::cout << WelchePosition << std::endl;
              c = Kartenstapel_Alice[WelchePosition]; // Karte holen, ...
              Kartenstapel_Alice.remove(c);           // entfernen, ...
260           tmcg->TMCG_SelfCardSecret(c, vtmf);     // aufdecken, ...
              if (!tmcg->TMCG_VerifyCardSecret(c, vtmf, std::cin, std::cout))
              {
                 std::cerr << "><> Öffnungsbeweis falsch (Betrugsversuch?)" <<
                    std::endl;
265              return -1;
              }
              Typ = tmcg->TMCG_TypeOfCard(c, vtmf);
              std::cerr << "Ich ziehe die Karte (von Position " <<
                 WelchePosition << "): " << Typ;
270           if (Typ)
                 std::cerr << std::endl;
              else
                 std::cerr << " (Zonk!)" << std::endl;
              Kartenstapel_Bob.push(c); // ... und auf Bobs Stapel legen.
275           // ... und mischt neu.
              tmcg->TMCG_CreateStackSecret(Mischgeheimnis, false,
                 Kartenstapel_Bob.size(), vtmf);
              tmcg->TMCG_MixStack(Kartenstapel_Bob, Mischstapel_Bob,
                 Mischgeheimnis, vtmf);                          // Maskieren, ...
280           std::cout << Mischstapel_Bob << std::endl;         // senden, ...
```

```
                tmcg->TMCG_ProveStackEquality(Kartenstapel_Bob, Mischstapel_Bob,
                   Mischgeheimnis, false, vtmf, std::cin, std::cout); // beweisen.
                Kartenstapel_Bob = Mischstapel_Bob;
             }
             else
             {
                // Alice zieht ...
                std::cin >> WelchePosition;
                std::cin.ignore(1, '\n'); // Newline verwerfen
                if (WelchePosition >= Kartenstapel_Bob.size())
                {
                   std::cerr << "><  Falscher Index (Trollversuch?)" << std::endl;
                   return -1;
                }
                c = Kartenstapel_Bob[WelchePosition];        // Karte holen, ...
                Kartenstapel_Bob.remove(c);                  // entfernen, ...
                tmcg->TMCG_ProveCardSecret(c, vtmf, std::cin, std::cout); // aufdecken,
                Kartenstapel_Alice.push(c); // ... und auf Alices Stapel legen.
                // ... und mischt neu.
                std::cin >> Mischstapel_Alice;
                if (!std::cin.good())
                {
                   std::cerr << "><  Stapelformat falsch (Trollversuch?)" << std::endl;
                   return -1;
                }
                if (!tmcg->TMCG_VerifyStackEquality(Kartenstapel_Alice,
                   Mischstapel_Alice, false, vtmf, std::cin, std::cout)) // verifizieren
                {
                   std::cerr << "><  Stapelbeweis falsch (Betrugsversuch?)" << std::endl;
                   return -1;
                }
                Kartenstapel_Alice = Mischstapel_Alice;
             }
          }
       }

       // Aufräumen
       delete vtmf, delete tmcg;
    }
```