FORMALE ANALYSE VON MASTERMIND

RAJIV LUND

Cryptography and Security
Computer Science
Saarland Univercity

21. Dezember 2009

BETREUER:
Prof. Michael Backes
Dr. Boris Köpf

ORT:
Saarbrücken

Für meine Familie und Freunde.

ZUSAMMENFASSUNG

Immer öfter werden Systeme über Side-Channel attackiert. Dabei werden Informationen, wie z.B. Antwortzeit, Energieverbrauch und Cacheinhalt, benutzt um den geheimen Schlüssel zu bekommen. Wir wollen uns nun anschauen wie sicher bestimmte Systeme gegen solche Angriffe sind. Dabei beschränken wir uns auf die Prüfungsverfahren von Kredit-Karten. Vorher müssen wir aber ein Modell (Mastermind) betrachten, mit dem wir solche Angriffe simulieren können. Danach werden wir versuchen Strategien zu finden, die einen Angriff effizient und nicht nur theoretisch möglich machen. Und am Ende werden wir dann ein Teil der Ergebnisse in unserem Beispiel-Fall (Visa PIN Verifikationsverfahren) anwenden.

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth —

DANKSAGUNG

Ich danke allen, die mir auf diesem langen Weg geholfen, mich unterstüzt und mitgearbeitet haben.
Ohne Euch hätte ich diese Arbeit nicht vollenden können.

Danke!

INHALTSVERZEICHNIS

1 EINFÜHRUNG

Mastermind ist bereits ein weltbekanntes Spiel. Aber es ist auch mehr als nur ein Spiel. Man kann Mastermind formalisieren und dadurch bekommt man ein Modell, das sowohl als Seitenkanalattacken (Kapitel 2.2) als auch auf NP-Vollständigkeit [10] analysiert oder für eine Konto-PIN Attacke [9] bzw. für Attacke auf Genomdaten [4] benutzt werden kann. Man kann erkennen, dass das Modell sehr allgemein ist. Es gibt bereits einige Ergebnisse auf dem Gebiet (Kapitel 8).

Es gibt auch den Forschungsgebiet der Seitenkanalattacken. Auch dort gibt es viele Resultate. Die Art der Attacken ist relativ jung im Vergleich zu anderen Attacken, aber trotzdem darf man sie nicht unterschätzen. Man kann zum Beispiel die Reaktionszeit eines Systems bei der Passworteingabe messen und dadurch auf die Ähnlichkeit zwischen dem echtem Passwort und gerade eingegebenem zurückschliessen. Beim Design des Systems hat man nicht bedacht, dass solche Angriffe möglich sind, weil sie nicht offensichtlich sind. Die entstehende Gefahr ist deswegen groß. Weitere Beispiele werden im Kapitel 2.2 aufgeführt.

Diese Arbeit beschäftigt sich mit der Fragestellung: “Kann man die Erkenntnisse aus einem der beiden Gebiete auf das andere übertragen?”. Wir wollen zuerst im Kapitel 2 das Spiel Mastermind, die Seitenkanalattacken sowie andere notwendige Konstrukte vorstellen und formalisieren. Im Kapitel 3 wollen wir uns mit der Implementierung von Mastermind in der Programmiersprache Haskell beschäftigen, um das Spiel dann in den Kapiteln 4 und 5 als eine Seitenkanalattacke mit bereits bestehendem Tool (welches auch in Haskell implementiert ist) [6] analysieren zu können. Daraufhin wollen wir die Nützlichkeit der Analyse untersuchen und nach den hinreichenden Bedingungen für Optimalität der Analyse im Kapitel 6 suchen. Anschliessend wollen wir eine bestimmte Seitenkanalattacke näher betrachten. Und zwar handelt es sich um eine Attacke auf das VISA Geheimzahl Verifizierungsverfahren [8]. Wir wollen zuerst das Verfahren verstehen (Kapitel 7.1), dann die Attacke (Kapitel 7.2) und daraufhin einen Schutz (Kapitel 7.3) und eine Analyse des Schutzes (Kapitel 7.4) aufstellen. Und am Ende wollen wir weitere Arbeiten auf den Gebieten vorstellen (Kapitel 8).

2 GRUNDLAGEN

Bevor wir unsere Arbeit anfangen, benötigen wir ein gewisses Grundgerüst an Wissen über das Spiel Mastermind, die Seitenkanalattacken sowie weitere Zusatzkonstrukte. Sie alle werden in diesem Kapitel erklärt.

2.1 MASTERMIND

Mastermind (Figure 1) wurde 1970 erfunden. Der Erfinder war Mordechai Meirovitz. Es handelt sich um ein Zwei-Personen (der Codierer und der Rater) Spiel. Das Spiel an sich besteht aus einem Spielbrett und verschiedenfarbigen Stiften (weisse und schwarze Stifte spielen eine besondere Rolle). Das Spielbrett besitzt eine feste Anzahl an Reihen (dabei stellt jede Reihe einen Spielzug dar). Und jede Reihe besitzt eine feste Anzahl an Löchern, in die man die Stifte einstecken kann.

Figure 1: Mastermind. Die vier Stifte ganz vorne bilden den Code. Die kleinen Stifte auf der rechten Seite sind die Antworten vom Codierer. (Das Bild ist entnommen von http://en.wikipedia.org/wiki/File:Mastermind.jpg, vom Benutzer: ZeroOne, unter der Creative Commons-Lizenz Attribution ShareAlike 2.0.)

Zum Anfang des Spiels bestimmt der Codierer geheim den Code aus Stiften. Dann fängt das Spiel an. Der Ratende stellt eine Anfrage, indem er eine potentiellen Code erzeugt und es dem Codierer zeigt. Der Codierer antwortet nun auf folgende Weise:

- Jeder Treffer in Farbe und Position des Stifts wird mit einem schwarzen Stift beantwortet.

- Jeder Treffer in Farbe aber nicht in der Position wird mit einem weissen Stift beantwortet.
- Falsche Stifte bekommen keine Antwort.
- Die Reihenfolge der weissen und schwarzen Stifte ist beliebig.

Man kann Mastermind als eine Funktion formalisieren:

$$\mathit{mastermind} : C \times C \to O$$

Dabei ist C die Menge der Codes und O die Menge der Antworten (Beobachtungen). Das erste Argument der mastemind-Funktion ist der echter Code und das zweite ist die Anfrage des Ratenden. Das Ergebnis der Funktion ist die entsprechende Antwort (Zahl der schwarzen und weissen Stiften). Das Ziel des Ratenden ist den Code noch innerhalb der erlaubten Versuche zu erraten, also eine Antwort nur aus schwarzen Stiften (wobei die Anzahl der Länge des Codes entspricht) zu bekommen.

2.2 SEITENKANALATTACKEN

Die Seitenkanalattacken wurden von dem Kryptologen Paul C. Kocher im Jahre 1996 vorgestellt. Im Gegensatz zur theoretischen Kryptoanalyse, bei der man das Verschlüsselungsverfahren analysiert und Schwächen im Verfahren an sich sucht, versucht man anhand der physischen Beobachtungen oder Fehlermeldungen, wie z.B. Laufzeit, Temperatur, Energieverbrauch, Cacheinhalt und Reflexionen auf Oberflächen, Rückschlüsse über das Geheimnis zu ziehen und somit den Schutz zu umgehen. Man kann z.B. am Energieverbrauch der Smartkarten die Verschlüsselung des Inhaltes rückwirkend machen oder an Reflexionen in Glässer oder Teekannen das Bild vom Bildschirm großteils rekonstruieren.

Man kann solche Attacken als folgende Funktionen formalisieren:

$$f : K \times M \to O$$

Dabei ist K die Menge aller Schlüssel, M die Menge aller Anfragen und O die Menge aller Beobachtungen. Wenn man die Formalisierung von Mastermind (Kapitel 2.1) mit dieser vergleicht, erkennt man, dass Mastermind als eine Instanz der Seitenkanalattacken gesehen werden kann. Dafür muss man nur die Mengen K und M gleichsetzen zu C.

2.3 PARTITION

Um die Funktionsweise unserer Analyse später zu verstehen, müssen wir das Konzept der Partitionen betrachten. Partitionen sind Datenstrukturen, die uns helfen bestimmte Objekte (hier die Schlüsseln) zu verwalten.
Für jede mögliche Anfrage $m \in M$ erzeugen wir eine Partition P_m. Dann für jeden möglichen Schlüssel $k \in K$ berechnen wir $o := f(k, m)$. Alle Schlüsseln k, die dieselbe Beobachtung $o \in O$ erzeugten, speichern wir in einem Block. Es entstehen Blöcke $B_o^m = \{k | f(k, m) = o\} \in P_m$. (In späteren Kapiteln werden wir m und o nicht genauer spezifizieren. Falls es um den i-ten Block der j-ten Partition geht, dann werden wir es auf folgende Art bezeichen $B_i^{(j)}$.) Die disjunkte Vereinigung aller Blöcke einer Partition ist die Menge aller Schlüsseln. Das folgende Bild (Figure 2) ist eine visuelle Darstellung der Datenstruktur.

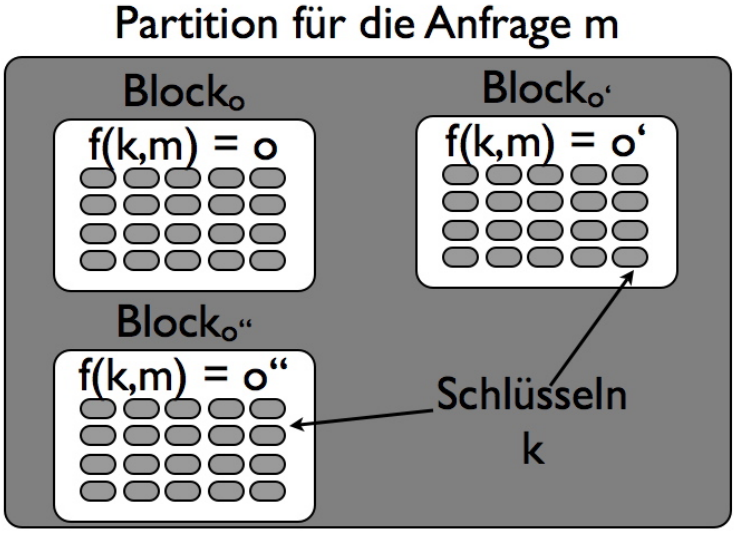

Figure 2: Beispiel einer Partition. Da alle Blöcke aus der Partition P_m stammen, wurde die Bezeichnung B_o^m auf B_o reduziert.

2.4 GREEDY-STRATEGIE

Um das Spiel Mastermind zu analysieren, werden wir die Greedy-Strategie benutzen. Zuerst werden wir einen einzelnen Schritt betrachten und dann die gesamte Strategie.

2.4.1 *Ein Schritt der Greedy-Strategie*

Im ersten Schritt sucht Greedy-Strategie (in Zukunft Greedy) unter allen Partitionen $\{P_m | m \in M\}$ die mit höchstem Informationsgehalt aus. Dabei ist für unsere Zwecke vollkommen ausreichend den Informationsgehalt einer Partition als die Feinheit der Partition zu interpretieren. Im i-ten Schritt nehmen wir die Partition P_m aus dem (i-1)-ten Schritt und versuchen die Blöcke B_o^m (für $o \in O$) so stark wie möglich zu verfeinern. Dafür schneiden wir jeden Block B_o^m mit allen Blöcken $B_{o'}^{m'}$ ($m' \neq m$ und $o' \in O$) und suchen für jeden B_o^m die Partition $P_{m'}$ aus, die mit den Blöcken $\{B_{o'}^{m'}\}$ den Block B_o^m am stärksten verfeinert hat. Nach dem Schneiden der Blöcke entsteht die feinere Partition P'_m, die als Partition für den i-ten Schritt gespeichert wird.

2.4.2 *Die gesamte Greedy-Strategie*

Nun wollen wir die einzelnen Schritte vereinen. Im ersten Schritt hatten wir P_m ausgesucht. Dann wird unsere erste Anfrage m sein. Darauf werden wir die Beobachtungen $o \in O$ bekommen. Somit entsteht ein Baum (wir werden diese Bäume Angriffsbäume nennen, um die besser von Spielbäumen (Kapitel 4.2) bei Spielen zu unterscheiden). Die Wurzel des Angriffsbaums ist die Menge aller Schlüsseln. Jede Kante symbolisiert eine Anfrage m' und eine Beobachtung o. Jeder Knoten ist dann der Block B_o^m, der durch das Schneiden mit den Blöcken der Partition $P_{m'}$ entsteht, also $B_o^m \in P'_m$. Die Menge der Knoten (Blöcke) der ersten Ebene des Angriffsbaums entspricht der Partition P_m. Um

die Situation besser zu verstehen, betrachten wir den Angriffsbaum (Figure 3). Dabei handelt es sich um Mastermind mit drei Farben und zwei Position. Das Spiel wird als eine Seitenkanalattacke aufgefasst.

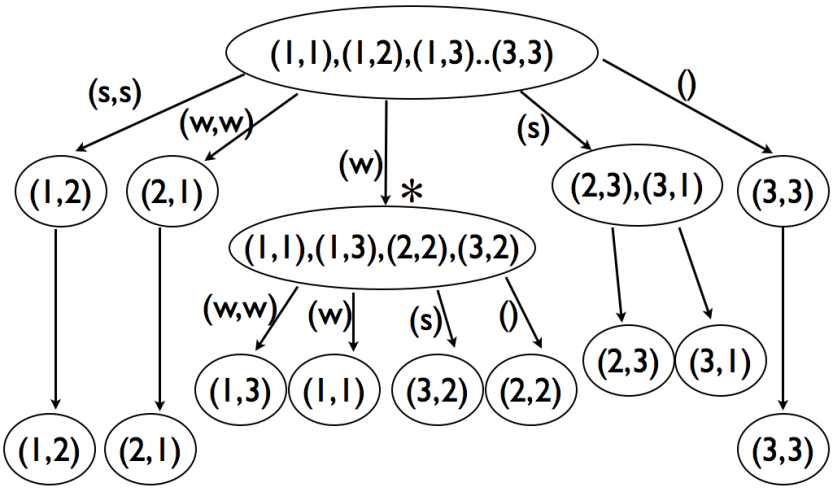

Figure 3: Angriffsbaum. Um die Lesbarkeit zu erhöhen sind nicht alle Kanten beschriftet und die Anfragen an den Kanten wurden weggelassen. Diese werden im Text näher erläutert.

Es gibt neuen verschiedene Schlüssel (in der Wurzel aufgelistet). Im ersten Schritt suchen wir m = (1,2) als Anfrage aus. Darauf können wir fünf verschiedene Beobachtungen ("s" steht für schwarzen Stift, "w" steht für weissen Stift und die leere Klammer steht für leere Beobachtung) bekommen. Die Menge der Knoten der ersten Ebene des Baums stellt die Partition $P_{(1,2)}$ dar. Der mit Stern markierte Knoten ist der Block $B_{(w)}^{(1,2)}$, der enthält noch vier potentielle Schlüssel. Im zweiten Schritt wollen wir die $P_{(1,2)}$ verfeinern, also u.a. $B_{(w)}^{(1,2)}$ in mehrere Teile spalten. Dafür wird Greedy $P_{(3,1)}$ aussuchen (weil durch das Schneiden dr Blöcke der Block $B_{(w)}^{(1,2)}$ maximal gespalten wird) und die Blöcke von $P_{(3,1)}$ mit $B_{(w)}^{(1,2)}$ schneiden. Also ist unsere zweite Anfrage (falls wir auf die erste Anfrage (1,2) die Beobachtung (w) bekommen) (3,1). Mit analogem Vorgehen verfeinern wie die anderen Blöcke so weit wie möglich und kennen den Schlüssel bereits nach zwei Anfragen.
Greedy verfeinert bis zu einem Fixpunkt (wenn in einem Schritt keine Verfeinerung mehr erreicht wird). Danach kann keine weitere Verfeinerung erreicht werden [6]. Der entstehende Angriffsbaum enthält die komplette Information über den Angriff. Die zugehörige Strategie zum Angriffsbaum bezeichenen wir als Angriff-Strategie. Man kann Greedy als eine lokal-optimale Angriff-Strategie sehen, da Greedy in jedem Schritt den maximal Informationsgewinn für einen Schritt gesehen gewährleiset. Es kann aber passieren, dass auf mehrere Schritte gesehen ein anderer Angriff besser gewessen wäre (Kapitel 6).

3 MASTERMIND

In diesem Kapitel implementieren wir die Funktion Mastermind (Kapitel 2.1) um sie mit der Greedy-Strategie (Kapitel 2.4) zu untersuchen. Dabei habe ich die Funktion in mehrere Unterfunktionen geteilt um eine bessere Übersichtlichkeit zu erreichen.

Wir bekommen die aktuelle *Anfrage* ($m \in M$) und besitzen den *Code* ($k \in K$) und wollen $o := f(k, m)$ berechnen. Beide Informationen werden als Listen vom Typ *Integer* übergeben. Also kodieren wir die Farben als Zahlen, was uns zwar das Lesen erschwert, aber die Handhabung im Programm vereinfacht (da bereits sehr viele Strukturen und Prozeduren für die *Integer* konzepiert sind).

Mit dem *Code* und der *Anfrage* müssen folgende Aktionen durchgeführt werden:

- Anzahl der schwarzen Stifte rausfinden (s. 3.1)
- Anzahl der weissen Stifte rausfinden (s. 3.2)

3.1 ANZAHL DER SCHWARZEN STIFTE BESTIMMEN

Diese Aktion unterteilen wir wieder in zwei:

- Anzahl der Positionen mit gleichen Farben in *Code* und *Anfrage* bestimmen (s. 3.1.1)
- Diese Positionen aus beiden Listen entfernen um weitere Aktionen nicht zu stören (danach können wir die Berechnung der weissen Stifte wesentlich einfacher durchführen) (s. 3.1.2)

3.1.1 *Anzahl bestimmen*

Dies wird durch die folgende Funktion bewerkstelligt.

```
blackNum :: (Eq t) => [t] -> [t] -> [Bool]
blackNum (x:xr) (y:yr) = if (x==y)
        then True:(blackNum xr yr)
        else (blackNum xr yr)
blackNum [] [] = []
```

Wir lesen die beiden Listen komponentenweise ein und im Falle, dass die *Integer* gleich sind, fügen wir *True* (einen schwarzen Stift) zu der Liste, die zurückgegeben wird, hinzu und sonst überspringen wir die Stelle.

3.1.2 *Anfrage und Code modifizieren*

Diese Aktion erreichen wir mit der nächsten Funktion.

```
black :: (Eq t) => [t] -> [t] -> [t]
black (x:xr) (y:yr) = if (x==y)
        then (black xr yr)
        else x:(black xr yr)
black [] [] = []
```

Hier arbeiten wir uns genau wie in der Funktion *blackNum* vor, nur jetzt wird die übereinstimmende Stelle aus der Liste *yr* gelöscht. Man beachte, dass wir diese Funktion zweimal aufrufen müssen und dabei die Parameter tauschen um die Stellen, die für die schwarzen Stifte "verbraucht" wurden in *Code* und in *Anfrage* zu löschen.

3.2 ANZAHL DER WEISSEN STIFTE BESTIMMEN

Hier darf man nicht vergessen, dass wir das Vorkommen der potenziellen Farbe an allen Positionen im modifiziertem *Code* beachten müssen. Zum unseren Glück verwalten wir unsere *Code* und unsere *Anfrage* als Listen. Deswegen können wir einfach auf die vordefinierte Funktion *elem* zugreifen. Sie testet, ob ein Element *x* in der Liste *yr* vorkommt.
Somit kommen wir zu der unterstehenden Funktion *whiteNum*. Sie funktioniert analog zu der Funktion *blackNum*.

```
whiteNum :: (Eq t) => [t] -> [t] -> [Bool]
whiteNum (x:xr) (yr) = if((elem x yr))
        then False:(whiteNum (xr) (delete x yr))
        else (whiteNum xr (yr))
whiteNum [] (xr) = []
```

Wir haben wieder das Problem, dass manche Farben evtl. doppelt gezählt werden könnten. Deswegen muss *whiteNum* nicht auf den ursprünglichen sondern auf modifizierten (nach Bereinigung von Stiften, die bereits einen schwarzen Stift als Beobachtung bekommen haben) *Code* und *Anfrage* aufgerufen werden. Zwar hätten wir die dazu nötige Funktion *black* direkt in der Funktion aufrufen können, aber aus Lesbarkeitsgründen verschieben wir das Problem in die Funktion *getWhite*.

```
getWhite :: (Eq t) => [t] -> [t] -> [Bool]
getWhite (xr) (yr) = whiteNum (black xr yr) (black yr xr)
```

3.3 MASTERMIND-FUNKTION

Und schon sind wir fast am Ende. Wir müssen nur noch die bereits vorliegenden Funktion zusammenbringen und die Ergebnisse vereinen. Das wird von der Funktion *mastermind* erledigt. Als Ergebnis entsteht eine Liste von *Boolean*, die für jeden schwarzen Stift eine *True* und für jeden weissen eine *False* enthält.

```
mastermind :: [Int] -> [Int] -> [Bool]
mastermind xr yr = (blackNum xr yr)++(getWhite xr yr)
```

Im nächsten Kapitel setzen wir uns mit dem Begriff der Optimalität auseinander. Danach können wir unsere Ergebnisse mit den bereits bekannten vergleichen.

4 OPTIMALITÄT

In diesem Kapitel wollen wir die verschiedenen Begriffe der Optimalität (Kapitel 4.2, 4.1) betrachten und dann versuchen Zusammenhänge zwischen den Bedeutungen herzustellen, damit wir unsere Analyse von Mastermind (Kapitel 2.1) mit Greedy-Strategie (Kapitel 2.4) mit den bereits bestehenden Ergebnissen (Kapitel 8) vergleichen können. Dieser Zusammenhang ist noch nicht erforscht worden.

4.1 OPTIMALITÄT BEI SEITENKANALATTACKEN

Im Kapitel 2.4 haben wir den Angriffsbaum betrachtet. Eine wichtige Bemerkung: der Angriffsbaum ist immer balanciert. Dies folgt unmittelbar aus der Tatsache, dass die Menge aller Knoten (Blöcke $B_o^m, o \in O$) einer Ebene des Angriffsbaums die Partition P_m bildet. Die optimale Angriff-Strategie für n Schritte zu haben, bedeutet P_m ist nach n Schritten feiner als mit jede anderen Angriff-Strategie. Dabei muss $m \in M$ nicht für alle Angriffe gleich sein. Konkret formuliert bedeutet dies:
Die Angriff-Strategie A ist nach $n \in \mathbf{N}$ Schritten optimal, wenn $\forall m' \in M, \forall$ andere Angriff-Strategien A' gilt : Feinheit von $P_{m'}$ mit der A' nach n Schritten $\leqslant$ Feinheit von P_m mit der A nach n Schritten. Mit anderen Wörtern ausgedrückt heisst es, es gibt keine andere Angriff-Strategie A', die eine feinere Partition $P_{m'}$ nach n Schritten erzeugen kann als A mit P_m.
Die Optimalität eine Angriff-Strategie nach n Schritten kann man berechnen, in dem man die Feinheit der n-ten Ebene des Angriffsbaums berechnet (dabei bildet man die Menge aller Knoten dieser Ebene und bekommt die entsprechende Partition P_m nach n Schritten).

4.2 OPTIMALITÄT BEI MASTERMIND ALS SPIEL

In Mastermind die optimale Strategie zu besitzen bedeutet, dass die erwartete Anzahl der Anfragen so gering wie möglich ist, also in $k \in \mathbf{R}$ vielen erwarteten Schritten auf die Lösung des Codes zu kommen. Wir haben auch hier einen Baum, den wir als Spielbaum bezeichnen. Der Spielbaum ist sehr ähnlich dem Angriffsbaum. Die Knoten und Kanten symbolisieren dasselbe (Code entspricht dem Schlüssel und Antwort entspricht der Beobachtung). Die Unterschiede liegen im Aufbau der beiden Bäume. Während der Angriffsbaum balanciert ist und jeder Pfad erst dann endet, wenn die Partition P_m nicht mehr verfeinert werden kann, kann der Spielbaum unbalanciert sein, weil hier ein Pfad endet, sobald man die Antwort aus nur schwarzen Stiften bekommt. Ein weiterer Unterschied liegt darin, dass im Angriffsbaum die Pfade enden, sobald wir alle Blöcke der Partition auf Größe eins reduziert haben. Also wissen wir, was der Schlüssel ist. Im Spielbaum müssen wir aber die Antwort aus nur schwarzen Stiften bekommen. Also reicht das Wissen nicht. Wie wir den Angriffsbaum in den Spielbaum überführen wollen, wird im Kapitel 4.3 durchgeführt.
Um zum Begriff der Optimalität zurückzukommen: optimal bedeutet hier die erwartete Pfadlänge im Spielbaum ist so gering wie möglich.

Dabei ist wichtig zu bemerken, dass man nicht nach einem Schritt $n \in \mathbf{N}$ die Optimalität untersucht, sondern immer über den kompletten Spielbaum. (Dies ist auch möglich, da man die Blöcke von Mastermind immer auf Größe eins reduzieren kann. Bei Seitenkanalattacken ist es nicht immer gewährleistet.)

4.3 ZUSAMMENHÄNGE ZWISCHEN VERSCHIEDENEN BEGRIFFEN

Wir haben jetzt die beiden Begriffe der Optimalität kennengelernt. Da wir Mastermind mithilfe der Greedy-Strategie analysieren, bekommen wir nur die Partition P_m (die Ebenen des Angriffsbaums) in einem bestimmten Schritt zurück. Dadurch können wir den Angriffsbaum rekonstruieren, aber er entspricht nicht dem Spielbaum. Dafür entwerfen wir einen Algorithmus, der als Eingabe die verschiedenen Partitionen $\{P_m^n\}$ (für verschiedene Schritte $n \in \mathbf{N}$) von Greedy bekommt und damit die erwartete Pfadlänge des Spielbaums berechnet. Somit können wir die Ergebnisse der Greedy-Strategie in Mastermind mit den bereits bestehenden Ergebnissen vergleichen. Vor allem ist unser Algorithmus nicht auf die Mastermind-Funktion beschränkt. Er kann mit jeder Funktion arbeiten, die Greedy analysieren kann.

4.3.1 *Algorithmus*

Die Grundidee:

- Wir wollen den Zeitpunkt erkennen, an dem die Pfade im Spielbaum zu Ende gehen würden.
- Dafür betrachten wir in jedem Schritt n die aktuelle Partition P_m^n und zählen die Blöcke (Knoten im Angriffsbaum), die im letzten Schritt auf einen Code minimiert wurden.
- Die Pfadlänge dieser Knoten entspricht dem aktuellem Schritt und muss nur noch um eins erhöht werden (um den Schritt für die Antwort aus nur schwarzen Stiften mitzurechnen).
- Sobald wir alle Blöcke auf Größe eins minimiert haben, können wir die Gesamtpfadlänge berechnen und daraus die erwartete Pfadlänge.

Der Algorithmus sieht dann so (Figure 4) aus.

4.3.2 *Korrektheit*

Theorem 1. *Mit dem aufgeführtem Algorithmus (Figure 4), können die erwartete Pfadlänge des Spielbaums berechnen, wenn wir nur die Partitionen P_m^n bekommen.*

Beweis. Wir werden einen Induktionsbeweis führen über die Anzahl der Schritte n. Dabei zeigen wir, dass in n-tem Schritt die Gesamtpfadlänge der bereits terminierten Pfade korrekt bestimmt wird.

Input: Funktion greedy, die zum aktuellem Schritt n die zugehörige Partition P^n_m liefert.
Output: Die erwartete Pfadlänge im Spielbaum.
Variablen:

- n ist der aktuelle Schritt
- sb ist die Anzahl der Blöcke der Größe eins aus dem letzten Schritt
- l ist die Gesamtlänge aller vollendete Pfade bis zum aktuellen Schritt
- k ist die Anzahl der Codes

Hilfsfunktion: numberOfSBlocks zählt die Blöcke, die in der aktuellen Partition Größe eins haben.

1. *n,sb,l* := 0;
2. while *(∃ block of size* > 1*)* do
3. *newSingleBlocks* := *numberOfSBlocks(greedy n) - sb*;
4. *l* := *l* + *newSingleBlocks* * *(n*+1*)*;
5. *n* := *n*+1;
6. *sb* := *sb* + *newSingleBlocks*;
7. od;
8. return (*l* / *k*);

Figure 4: Algorithmus zur Berechnung der erwarteten Pfadlänge des Spielbaumes

IA: Für $n = 0$ ist die Aussage klar, weil wir keinen Schritt der Strategie gemacht haben. Also können wir auch keine Schlüsseln bestimmen. Deswegen gibt es keine terminierten Pfade. Es entspricht der Zeile 1 des Algorithmus.
Sei $n = 1$. Also haben wir einen Schritt der Strategie hinter uns gebracht und die initiale Menge an Schlüsseln K in die Partition P überführt. Sei k = *numberOfSBlocks(P)*. Da wir vorher keine Schritte gemacht haben, können wir auch nichts abziehen von k.
Jetzt multiplizieren wir k mit 2, da 1 der aktuelle Schritt ist und wir noch die entsprechende Beobachtung aus nur schwarzen Stiften bekommen müssen und somit einen weiteren Schritt für diese Pfade brauchen.

$$\textit{newSingleBlocks} := k = k; \qquad l := 2 * \mathrm{k} \tag{4.1}$$

Dies ist auch gleichzeitig die Summe der Pfadlängen aller bereits terminierten Pfade. Dies entspricht im Algorithmus dem erstem Durchlauf der Schleife in Zeilen $2 - 7$.

IV: Die Vorraussetzung gelte für n Schritte.

IS: $n \to n+1$
Sei k' die Anzahl der Blöcke der Größe 1 nach dem n-tem Schritt und l'

die Summe an Pfadlängen bis jetzt. Analog zum *IA* führen wir folgende Rechnung durch, die der Schleife im Algorithmus entspricht:

$$l=l'+(k-k')*((n+1)+1) \tag{4.2}$$

Wir haben nur die Blöcke gezählt, die genau in diesem Schritt auf 1 minimiert worden sind, da k' und l' die entsprechende Zahlen aus dem letzten Durchlauf sind (nach *IV*). Und die entstehende Zahl wurde mit der richtigen Länge mulitpliziert, da wir uns gerade in der n+1-ten Ebene des Baums befinden. Also haben wir die Summe der Pfadlängen immer noch korrekt berechnet. Womit die Gültigkeit für alle n folgt. □

Nun wollen wir die Hilfsfunktion *numberOfSBlocks* nachliefern. Jede Partition $P_m = \{B_o^m | o \in O\} = \{\{k | f(k, m) = o\} | o \in O\}$ ist eine Menge von Mengen. Deswegen verwalten wir eine Partition als eine Liste von Listen (wegen leichte Handhabung der Listen).

```
numberOfSBlocks :: (Num t) => [[a]] -> t
numberOfSBlocks [] = 0
numberOfSBlocks (x:xs) = if ( length x == 1)
        then 1 + numberOfSBlocks xs
        else numberOfSBlocks xs
```

Die folgende Prozedur *pahtL* ist die komplette Implementierung des Algorithmus (Figure 4). Dabei bestimmt *guesswork* anhand der Partition, ob es noch Blöcke gibt, die auf die Größe eins minimiert werden können. Die Funktion *expVal* ist der letzte Schritt. Sie nimmt die berechnete Summe der Pfade aus der vorherigen Funktion und dividiert durch die Anzahl der Blöcke am Ende. Dabei sind die Parameter c und p die Anzahl der Farben und Länge des Codes und werden für *greedy* benötigt.

```
-- l is length of all paths
-- sb is number of blocks with only one key from last level
-- \emph{n} is the recent step
-- c is number of colours
-- p is number of pegs
pathL :: Int -> Int -> Int -> Int -> Int -> Int
pathL l sb \emph{n} c p= if( notstop)
        then pathL (l+newSingleBlocks *(n+1)) (newSingleBlocks + sb
             ) (n+1) c p
        else (l+newSingleBlocks * (n+1))
 where  partion = greedy (funcPart mastermind [1..c] p) n
        block = numberOfSBlocks(partion) - sb
         where notstop = guessWork (partion) > 1

expVal:: Int->Int-> Double
expVal  \emph{n} m = fromIntegral x / fromIntegral (n^m)
 where x = pathL 0 0 0 m \emph{n}
```

Auf diese Weise können wir im nächsten Kapitel unsere Greedy-Strategie mit bereits bekannten Ergebnissen vergleichen.

Am Ende möchte ich noch ein paar Bemerkungen auflisten:

1. Unsere Funktion muss nicht terminieren. Wir wollen die erwartete Pfadlänge bis die Größe der Blöcke eins ist. Aber sollte dies

einfach nicht erreichbar sein, divigiert unser Algoritmus. Was aber gleichzeitig auch der unendlichen Pfadlänge entspricht.

2. Man muss sich nicht auf die Mastermind Funktion beschränken. Man kann beliebige Funktionen auf diese Weise untersuchen. Dazu muss man nur einen zusätlichen Parameter einführen und diesen anstatt der *funcPart* einsetzen.

5 VERGLEICH DER ERGEBNISSE

5.1 OPTIMALITÄT VON GREEDY FÜR MASTERMIND ALS SPIEL

Nun können wir die Analyse mit der Greedy-Strategie (Kapitel 2.4) mit der optimalen Strategie (Kapitel 8) vergleichen. Greedy erreicht 4.648 Schritte, während die optimale nur 4.340 Schritte benötigt. Also benötigt Greedy im Schnitt 7% mehr Schritte. Der Nachteil der optimalen Strategie liegt darin, dass sie nur für das Standard Mastermind und wenige Variationen (in Anzahl der Farben bzw. Steckplätze) bestimmt wurde. Greedy dagegen ist so allgemein, dass sie für alle Variationen verwendet werden kann.

5.2 ALTERNATIVE OPTIMALITÄTSBEGRIFFE

Ein weiterer interessanter Aspekt ist die kleinste obere Schranke an maximal benötigten Zügen, die man benötigt um den Code zu erraten. Genauer: wenn wir wissen, dass eine Strategie als obere Schranke k Zuge hat. Dann ist k die Höhe des Spielbaums (der längste Pfad). Man könnte alternativ diesen Begriff als Optimaliät definieren. Und es gibt dazu bereits Resultate. Goddard [3] hat eine Tabelle mit den oberen Schranken für Variationen von Mastermind aufgestellt. Wir erstellen eine analoge Tabelle mit der Greedy-Strategie.

Anzahl der Steckplätze

Farben	2	3	4	5	6	7	8
2	3	3	4	4	5	5	6
3	4	4	4	4	5		
4	4	4	4	5			
5	5	5	5				
6	5	5	5				
7	6	6					
8	6						
9	7						
10	7						

Goddard

Anzahl der Steckplätze

Farben	2	3	4	5	6	7	8
2	3	3	4	4			
3	3	4	4	5			
4	4	4	4	5			
5	4	5	5				
6	5	5	6				
7	5						
8	6						
9	6						
10	6						

Greedy

Figure 5: Gegenüberstellung der Ergebnisse

Wenn wir jetzt beide Tabellen (Figure 5) vergleichen bemerken wir, dass für größere k und m die Greedy-Strategie einen Schritt zu viel brauchen kann (z.B. bei Standard-Mastermind k = 6 und m = 4). Aber für die Spalte m = 2 (markierte Stellen) brauchen wir zum Teil weniger Schritte als die untere Schranke von Goddard. Man kann sogar durch explizite Beispiele von Strategien belegen, dass z.B. für k = 3 und n = 2 (s. Figure 3. Es ist zwar ein Angriffsbaum, aber man kann ihn sehr schnell in einen Spielbaum überführen.) man maximal 3 Schritte braucht. Somit folgt, dass die Werte von Goddard in diesen Fällen falsch sind. Man kann sogar die Quelle der Fehler finden. Er gibt eine explizite

Formel für die Spiele mit zwei Steckplätzen an: $f(2,k) = \lceil \frac{k}{2} \rceil + 2$. Im Beweis argumentiert er, dass man bei $\lceil \frac{k}{2} - 1 \rceil$ Schritten mind. zwei Farben nicht betrachtet hat und somit die übrigen vier Codekombinationen nicht in einem Schritt unterscheiden kann. Erstmals: bei $\lceil \frac{k}{2} - 1 \rceil$ Schritten kann nur eine Farbe oder sogar gar keine nicht betrachtet geblieben sein, wenn k ungerade ist (z.B. k = 3 $\Rightarrow$ mit zwei Schritten haben wir zwangsweise alle Farben betrachtet und wissen bereits, was die Lösung ist). Zweitens: wenn doch zwei Farben bleiben, dann kann man vlt. die übrigen Kombinationen nicht in einem Schritt unterscheiden, aber oft in zwei.

Also zusammengefasst:

- Greedy ist bei Mastermind nicht optimal, aber nah dran.
- Greedy ist eine allgemeine Strategie, die bereits angewendet werden kann.
- Wir haben mit Hilfe der Analysen aus letzteren Kapiteln die kleinsten oberen Schranken für die maximal benötigte Anzahl der Schritte verbessert.

6 WANN IST GREEDY OPTIMAL?

Im letzten Kapitel haben wir die Optimalität von Greedy bei Mastermind als ein Spiel betrachtet. Jetzt wollen wir die Optimaltiät von Greedy bei Seitenkanalattacken untersuchen. Dabei ist bereits bekannt, dass Greedy für einen Schritt genauso gut ist wie die optimale Strategie [6]. Aber es ist auch möglich Beispiele [6] anzugeben, in denen Greedy schlechter als die optimale Strategie sein kann.

6.1 WORSTCASE FÜR GREEDY

Dabei betrachten wir folgende drei Partitionen (Blöcke entsprechen []-Klammern):

- $P_1 = \{[1], [2], [3,4,5]\}$
- $P_2 = \{[1], [2,3,4], [5]\}$
- $P_3 = \{[1,2,3], [4,5]\}$

Das Maß, das Greedy für den Informationsgehalt verwendet, ist (wie im Kapitel 2.4) die Feinheit einer Partition. Deswegen wird Greedy im ersten Schritt P_1 oder P_2 wählen, da diese jeweils drei Blöcke beinhalten. O.B.d.A: Greedy nimmt die erste Partition. Wenn der Schlüssel sich im letzten Block befindet, dann muss dieser weiterverfeinert werden. Aber im nächsten Schritt wird Greedy den Block $[3,4,5]$ entweder mit Blöcken $[2,3,4], [5]$ der zweiten Partition oder mit Blöcken $[1,2,3], [4,5]$ der dritten Partition scheiden können. In beiden Fällen entsteht ein Block der zwei Schlüsseln enthält.
Die optimale Strategie nach zwei Schritten hätte im ersten Schritt P_3 ausgesucht und dann im zweiten Schritt den Block $[1,2,3]$ mit Blöcken $[1], [2], [3,4,5]$ der ersten Partition bzw. den Block $[4,5]$ mit Blöcken $[2,3,4], [5]$ der zweiten Partition geschnitten. Und somit hätte die optimale Strategie bereits nach zwei Schritten alle Blöcke auf die Größe eins minimiert. Also kann Greedy schlechter sein als die optimale Strategie. Es gibt komplexere Beispiele [6], in denen Greedy beliebig schlechter als die optimale Strategie sein kann.

6.2 HINREICHENDE BEDINGUNG

Aufgrund der obigen Beobachtung interessieren wir uns für die Fälle, in denen Greedy auf jeden Fall so gut wie die optimale Strategie nach allen n Schritten ist. Wir suchen nach hinreichenden Kriterien, die anhand der Struktur der Partitionen sicherstellen, dass Greedy optimal ist. Also muss die Struktur so sein, dass die lokale Optimalität gleich der globalen Optimalität wird. In Figure 6 ist eine solche Situation dargestellt. Die Blöcke $B_1^{(i)}$ sind paarweise disjunkt, dadurch ist es gleichgültig welche Partition man am Anfang aussucht (weil alle Partitionen genau zwei Blöcke enthalten, also gleich fein sind) und in welcher Reihenfolge man den entsprechenden B_2 minimiert.
Also wenn wir Partitionen bekommen, die diese Struktur haben, wissen

wir bereits, dass die Greedy-Strategie optimal ist. Wir müssen nur noch das hinreichende Kriterium beweisen.

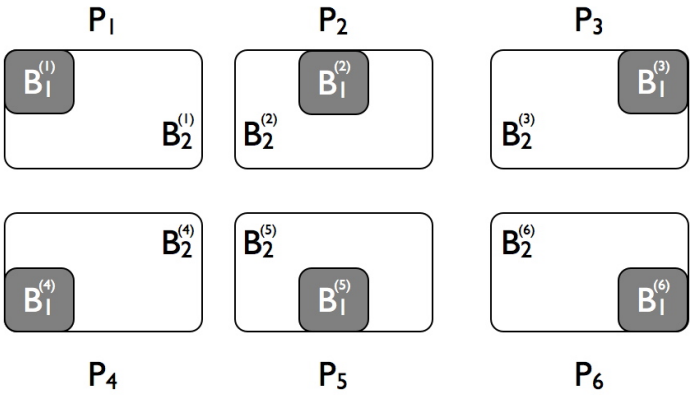

Figure 6: Situtation, in der Greedy optimal ist

6.3 FORMALISIERUNG

Sei D: $P \to \{1,2\}$ eine Verteilungsfunktion, die jedem Block eine Zahl k $\in$ {1,2} zuordnet. (Im folgendem kürzen wir $D(B^{(i)}) = k$ mit $B_k^{(i)}$ ab).

Theorem 2. *Wenn es eine Funktion* D *gibt sodass:*

$$\forall i \neq j \in \{k \in \mathbf{N} | 0 < k <= \#\text{Partitionen}\} : B_1^{(i)} \cup B_2^{(i)} = P_i \wedge B_1^{(i)} \subset B_2^{(j)} \tag{6.1}$$

gilt, dann ist Greedy $\forall$ $n \in \mathbf{N}$ *Schritte optimal.*
Bemerkung*: Das Maß, das wir für den Informationsgehalt verwenden, ist die Feinheit der Partition.*

Aber bevor wir dies beweisen, werden wir ein paar Lemmas benötigen.
Mit den oben genannten Voraussetzungen

Lemma 1.

$$\forall B_1 : \textit{Sie sind paarweise disjunkt} \tag{6.2}$$

Beweis. Wir beweisen es per Widerspruch. Man nehme an:

$$\exists i, j : i \neq j \wedge B_1^{(i)} \cap B_1^{(j)} \neq \emptyset \tag{6.3}$$

Dann wäre die verlangte Vorraussetzung nicht mehr erfüllt, weil dann z.B. $B_1^{(i)} \not\subset B_2^{(j)}$ gilt. Und somit zum Widerspruch führt. □

Lemma 2. *Wenn ein Block verfeinert wird, dann wird er immer zweigeteilt.*

Beweis. Jede Partion besitzt genau zwei Blöcke, wobei einer von denen im zu verfeinernden Block enthalten ist. Somit können wir es maximal zweiteilen. □

Lemma 3. *Nur einer dieser Blöcke kann weiterverfeinert werden.*

Beweis. Wir können nach Lemma 2 annehmen, B_2 in zwei Teile T_1 und T_2 zerlegt zu haben. Um dies zu erreichen, haben wir eine Partition P_k benutzt. Für $B_1^{(k)}$ galt $B_1^{(k)} \subset B_2$. Also ist einer der entstandenen Teilblöcke (z.B. T_1) genau $B_1^{(k)}$.
Nur wollen wir zeigen, dass T_1 nicht mehr weiterverfeinert werden kann (es folgt bereits implizit aus Lemma 1). Also nehmen wir wieder folgendes an:
$\exists B_1^{(j)}, B_2^{(j)} : B_1^{(j)} \cap T_1 \neq \emptyset \neq B_2^{(j)} \cap T_1$ damit wir verfeinern können.
$\Rightarrow B_1^{(j)} \not\subset B_2^{(k)}$ was Widerspruch zur Vorraussetzung ist, weil $\emptyset \neq B_1^{(j)} \cap B_1^{(k)}$. □

Jetzt haben wir alles was wir brauchen um den eigentlichen Beweis zu führen.

Beweis. Nach Lemma 3 können wir immer nur einen von beiden bereits verfeinerten Teilblöcken weiterverfeinern. Dadurch entsteht ein vollkommen unbalancierter Baum (Figure 7). Entweder erreichen wir direkt das maximale Ergebnis (z.B. wenn der Schlüssel im Block B_1 liegt) oder wir gehen rechts im Baum runter, bis wir nicht mehr weiterkommen (linken Pfad wählen). Die einzelnen Ebenen im Baum können beliebig permutiert werden, da die Blöcke, mit denen wir verfeinern, paarweise disjunkt sind (folgt aus dem Lemma 1).
Also ist es gleichgültig in welcher Reihenfolge wir uns durcharbeiten. Die optimale Strategie ist genau so gut wie Greedy, weil die Reihenfolge der Abarbeitung gleichgültig ist. □

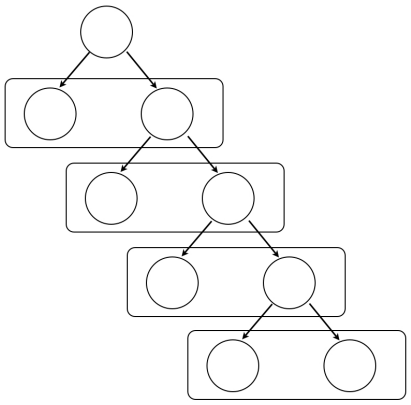

Figure 7: Der Verlauf der Strategie. Sobald wir das linke Kind nehmen, liegt der Schlüssel im einem Block $B_1^{(i)}$ und wie können nicht mehr weiter verfeinern.

Bemerkung: Sollten die Blöcke nicht disjunkt sein, dann könnte es Schritte geben, die lokal optimal, aber nicht global optimal sind. Gleichzeitig würde Greedy keinen Unterschied sehen, da wir immer dieselbe Anzahl an Blöcken erzeugen.

Im nächsten Kapitel betrachten wir eine spezielle Seitenkanalattacke auf das VISA-Karten Verifikationsverfahren und versuchen das gesammelte Wissen anzuwenden.

7 SEITENKANALATTACKEN

In diesem Kapitel wollen wir das VISA PIN Verifikations-Verfahren genauer betrachten. Für dieses Verfahren gibt es eine starke Seitenkanalattacke [8], die wir auch betrachten. Und anschliessend wollen wir gegen diese Attacke eine Schutzmassnahme entwickeln und diese dann analysieren.

7.1 VISA PIN VERIFIKATION

Üblicherweise werden die VISA-Karten mit einer Geheimzahl geschützt. Die Geheimzahl ist eine sehr sensible Information und sollte in keiner Datenbank gespeichert werden. Deswegen muss die Bank einen Verifikationswert, der vorallem von der Geheimzahl abhängig ist, für jede Karte berechnen und diesen dann speichern. Vereinfacht kann man sich es auf folgende Weise vorstellen:

- Jeder Kunde besitzt eine VISA Karte mit einer bestimmten Kontonummer (PAN) und der zugehörigen Geheimzahl (PIN).
- Bevor die Karte ausgehändigt wird, berechnet die Bank einen PIN Verification Value (PVV), indem sie PAN und PIN zu einer Zahl "vereinigt" (z.B. Xoren oder Konkatenieren) und die entstehende Zahl verschlüsselt. Von der entstehenden Zahl werden üblicherweise die letzten vier Ziffer als PVV festgelegt und gespeichert.
- Dann wird die Karte an den Kunden übergeben.
- Bei einer Zahlung wird die Pseudo-PIN vom Kunden eingegeben und die PAN von der Karte ausgelesen.
- Da es sehr sensible Informationen sind, wird die PIN für die Übertragung zur Bank in den so genannten Encrypted Pin Block (EPB) verschlüsselt.
- Sobald die EPB mit PAN die Bank erreicht, wird die Pseudo-PIN wieder entschlüsselt.
- PAN wird mit Pseudo-PIN "vereinigt" und die entstehende Zahl wird wie am Anfang verschlüsselt und die letzten vier Stellen extrahiert. Es entsteht PVV'.
- Die Bank vergleicht die errechnete PVV' mit der PVV aus der Datenbank. Bei Übereinstimmung wird die Zahlung freigegeben.

7.2 ATTACKE

Die genaue Verschlüsselung ist für unsere Attacke irrelevant. Es ist entscheidend, dass wir den Zugang zu einem Gerät haben, der PVVs berechnet. Genauer: wir können die Funktion $pvv(\mathsf{EPB}, \mathsf{PAN})$ berechnen. Wir werden versuchen die Zufälligkeit (die durch die EPB entsteht und die PIN für uns unkenntlich macht) zu umgehen, indem wir das System gegen sich selber nutzen. Dabei ist die folgende Beobachtung

für das Verständnis der Attacke entscheidend.
Die pvv-Funktion ist deterministisch in Bezug auf PAN und PIN aus dem EPB. Wenn wir zu einer bestimmten PIN ein EPB erzeugen und danach mit beliebiger PAN den $PVV := pvv(EPB, PAN)$ berechnen, dann bekommen wir den PVV zu dieser Kombination. Jetzt bekommen wir einen neuen unbekannten EPB', darin steckt die PIN' (= PIN, wir wissen aber nicht, dass die PINs gleich sind. Wir können es nicht erkennen, da die Verschlüsselung es unkenntlich macht.) Jetzt nutzen wir die Funktion pvv. Sie kann die Verschlüsselung rückgängig machen und den neuen $PVV' := pvv(EPB', PAN)$ berechnen. Da sie deterministisch ist, werden wir den gleichen PVV' bekommen wie am Anfang. Und am $PVV = PVV'$ erkennen wir, dass die PIN' und PIN auch gleich sein müssen. Da wir die erste PIN selber ausgewählt haben, kennen wir auch die unbekannte PIN. Es ist wichtig diese Beobachtung zu verstehen, da sie den Kern der Attacke verkörpert.

Die Attacke besteht aus zwei Phasen: Training und Angriff. Beim Training passiert folgendes:

- Es werden 10'000 EPBs E_i ($i \in [0..9999]$) erzeugt (für jede PIN einen, da die PINs üblicherweise vierstellig sind).
- Man sucht irgendeine PAN A_x aus.
- Jetzt berechnet man $pvv(E_i, A_x)$ für alle i und speichert die Einträge in einer eigenen Datenbank.

Es entsteht eine Tabelle, die der aufgeführten entspricht.

PVV	PIN
...	...
1279	7227
1280	8341
1281	----
1282	9374
1283	1987, 4577
1284	8347
...	...

Figure 8: Die Datenbank des Angreifers

Beim Angriff bekommt man einen EPB E_k (mit darin versteckten PIN_k) und PAN A_k. Anstatt den $pvv(E_k, PAN_k)$ zu berechnen, machen wir folgendes:

- Berechnen $pvv(E_k, A_x)$. Sei das Ergebnis PVV_k. (Man beachte, dass wir die PAN zur Berechnung nehmen, mit der wir trainiert haben).
- Schauen in der Datenbank nach, welche PIN zum PVV_k gespeichert ist bzw. beim Training beobachtet wurde.

Diese PIN ist genau die gesuchte und geheime PIN_k.
Bemerkung: Wir versuchen nicht den PVV der Bank zu imitieren, es ist

für uns irrelevant. Wir wollen die PIN vom Kunden rausfinden.

Wie man sieht, ist die Attacke sehr stark. Wenn man die Zeit für das Trainieren vernachlässigt, kann eine PIN in sehr kurzer Zeit rausgefunden werden. Und das ist keineswegs eine "Einwegattacke". Man kann mit einem Training beliebig viele PINs entlarven.

7.3 SCHUTZ

Dabei muss man beachten, dass in der Tabelle zu jedem PVV im Schnitt weniger als zwei PINs gehören.
Als Gegenmassnahme (zum Schutz von PIN) verändern wir die pvv-Funktion so, dass für einen PVV mehrere PINs in Frage kommen. Somit reicht es nicht mehr mit einer fiktiven PAN zu trainieren. Es reicht auch nicht mit mehreren PANs zu trainieren, da man kein Wissen darüber hat, welche PANs man wählen muss um die PIN eindeutig zu identifizieren. Somit kann die Komplexität enorm gesteigert werden. Und das Modell, was wir jetzt bekommen entspricht den Modellen aus vorherigen Kapiteln. Wir suchen nach einem geheimen Schlüssel (PIN), dazu können wir Anfragen (PVV) generieren und als Beobachtung bekommen wir die PINs, die zu dem PVV gehören. Also versuchen wir die Blöcke soweit zu minimieren, bis nur noch eine PIN in Frage kommt.
Auf dieses Problem können wir die Greedy-Strategie anwenden und anhand der Ergebnisse urteilen, ob es wirklich einen Schutz darstellt oder nicht.

7.4 ANALYSE DES SCHUTZES

Für die Analyse müssen wir einige Vereinfachungen vornehmen. Anstatt zu verschlüsseln werden wir eine zufällige Permutation benutzen. Und aus Laufzeitgründen werden wir PANs der Länge zwei und PINs der Länge zwei einführen.
Wir xoren alle Ziffern des PANs mit denen des PINs. Dann permutieren und extrahieren wir die letzte Ziffer als PVV. Die Funktion die entsteht, kann mit Hilfe der Greedy-Strategie analysiert werden. Uns interessiert vorallem die Gestalt der Blöcke.
Nach dem wir die Partitionen berechnet haben, bemerken wir, dass alle Blöcke sehr ähnlich sind. Sie enthalten die selbe Anzahl an PINs und alle PINs haben eine gleiche Stelle. Das liegt daran, dass wir nur eine Stelle von xor(PAN,PIN) extrahieren und somit die Information über die andere Stelle verlieren. Deswegen stellen wir nach der Analyse fest, dass Greedy die Blöcke nicht verkleinern kann. Das liegt daran, dass alle Partitionen gleiche Blöcke beinhalten und dadurch nicht mehr minimiert werden können.
Also könnte der Angreifer nicht mehr zwischen verschieden PINs unterscheiden und somit wäre die PIN geschützt. Aber auf der anderen Seite, haben wir natürlich das System allgemein schwächer gemacht. Es gibt jetzt mehrere PINs, die auf denselben PVV abbilden. Der Angreifer weiss zwar nicht welche das sind, aber die Wahrscheinlichkeit, per Zufall eine akzeptable PIN zu erwischen, ist gestiegen. Somit sind weitere Analysen notwendig um die beste Wahl der pvv-Funktion zu bestimmen. Vorallem haben wir in unserer Analyse starke Vereinfachungen (Permutation anstatt Verschlüsselung) vorgenommen. Es wäre sehr

interessant zu erfahren inwieweit man dies auf Verschlüsselungen übertragen kann.

8 WEITERE ARBEITEN

Zum Schluss wollen wir die Ergebnisse, die bis jetzt existieren, zusammentragen. Dabei werden wir chronologisch vorgehen und nur einen kurzen Überblick geben.

8.1 ERGEBNISSE BEI MASTERMIND

Die erste Veröffentlichung war von Donald Knuth [5]. Dort analysiert er die Standard-Variante (6 Farben und 4-stellige Codes) von Mastermind und stellt eine Strategie auf, die den gesuchten Code in maximal fünf Spielzügen errät. Seine Strategie ist sehr ähnlich zu der Greedy-Strategie (Kapitel 2.4). Man kann Knuths-Strategie als einen Speziallfall der Greedy-Strategie betrachten. Die Ähnlichkeit wird man auch an den später berechneten Resultaten von Flood sehen.

Die chronologisch nächste Arbeit stammte von V. Chvátal [1]. Er unterscheidet zwischen zwei verschiedenen Arten von Mastermind: statisch und aktiv. Die statische Variante, stellt zuerst die Fragen und (am Ende) bekommt dann die Antworten. Also ist diese Variante schwieriger als die aktive, in der wir die Antworten unmittelbar nach der Frage bekommen. Für unsere Anwendungen ist die aktive Variante interessanter, weil diese näher am Prinzip von Seitenkanalattacken ist.

Chvátal stellt auch eine kleinste obere Schranke für die Anzahl der nötigen Anfragen bis zu dem Erraten des Codes auf, wenn die Anzahl der Farben relativ klein im Vergleich zu der Anzahl der Steckplätze ist. Dabei kann man seine Schranke korrigieren, wenn man die Anzahl der möglichen Antworten genauer zählt. Er behauptet, es gäbe nicht mehr als $\binom{n+2}{2}$ Antworten, wobei n für die Anzahl der Steckplätze steht. Kombinatorisch gesehen ist die Begründung richtig:

- Wir haben n Steckplätze und drei verschiedene Arten von Antwortmoglichkeiten: schwarz (S), weiss (W) und nichts (N).
- Die Reihenfolge ist gleichgültig.
- Man kann n-Mal S wählen, dann bleibt für die anderen Stifte nur eine Möglichkeit (W = 0 und N = 0).
- Bei (n-1) vielen S haben wir zwei Möglichkeiten für den Rest (entweder W = 1 oder N = 1).
- Bei (n-2) S haben wir drei Möglichkeiten für den Rest (entweder W = 2 und N = 0 oder W = 1 und N = 1 oder W = 0 und N = 2).
- Man iteriert den Prozess und kommt zum letzten Fall mit 0 vielen S und (n+1) Möglichkeiten für den Rest.
- Es entsteht folgende Summe: $\sum_{i=1}^{n+1} i = \frac{(n+1)*(n+2)}{2} = \binom{n+2}{2}$ nach der Gauss-Summenformel.

Aber nicht alle Kombinationen im Spiel sind zulässig. Dabei ist die Antwort (n-1) viele schwarze Stifte und ein weisser Stift nicht möglich.

Es würde heissen, dass alle Stifte bis auf einen auf der richtigen Position mit der richtigen Farbe positioniert sind, aber der letzte die richtige Farbe und die falsche Position besitzt. Was nicht geht, da nur eine Position nicht bestimmt ist. Deswegen muss man von der Anzahl der möglichen Antworten diese Antwort abziehen.
$\Rightarrow \binom{n+2}{2} - 1$ viele Antwortmöglichkeiten. Somit ändert sich die untere Schranke von $\frac{n*\log k}{\log \binom{n+2}{2}}$ zu $\frac{n*\log k}{\log(\binom{n+2}{2}-1)}$. Diese neue Schranke ist höher als die vorherige. Leider habe ich kein Beispiel gefunden, in dem dies eine entscheidende Rolle spielen würde. Aber das wäre eine interessante Aufgabe für weitere Forschungen.

Die nächste Arbeit ist von Merrill M. Flood [2]. Er beschäftigt sich mit Knuth's, Irving's und Neuwirth's Strategien, analysiert sie und optimiert die nichtdeterministischen Entscheidungen. Bei vielen Entscheidungen über den nächsten Schritt, sind bestimmte Anfragen gleich optimal für den einen Schritt. Aber im Verlauf des Spiels können sie unterschiedlich optimal sein. Genau solche Situationen werden hier optimiert. Nach seinen Berechnungen braucht Knuth's Strategie 4.640 Schritte im Schnitt um auf die Lösung zu kommen (und die Greedy Strategie 4.648) . Wenn man aber Neuwirth's Strategie optimiert, kommt man auf 4.365 Schritte im Schnitt, was eine Verbesserung zu vorherigen Ergebnissen ist.

Anschliessend kommen wir zu der Arbeit von Kenji Koyama und Tony W. Lai [7]. Sie haben die schnellste Strategie entwickelt. Dabei war bis dato immer die Größe des Suchraums das Problem. Diese war so hoch, dass man nicht alle Fälle untersuchen konnte. Koyama und Lai haben einen Weg gefunden viele Codes gleich zu behandeln und dadurch den Suchraum entscheidend zu minimieren. So war eine komplette Analyse möglich. Ihre Strategie kommt auf ≈ 4.341 Schritte im Schnitt und ist die schnellste insgesamt. Da dies eine Bruteforce-Methode war, ist der Beweis durch die Methode erbracht worden.

Nun kommen wir zur letzten Veröffentlichung über Mastermind. Sie ist von Wayne Goddard [3]. Zuerst möchte ich bemerken, dass wir bis jetzt in diesem Kapitel immer das Standard-Spiel mit sechs Farben und vier Steckplätzen betrachtet haben. Es ist natürlich interessant, wie stark die sich Komplexität des Spiels mit steigenden Parametern verändert. Goddard widmet sich genau dieser Aufgabe. Er berechnet die durchschnittliche Anzahl der Schritte bis zur der Lösung als auch die obere Schranke an Schritten, die nötig sind um den Code zu erraten. Wir haben uns bereits mit seinen Ergebnissen beschäftigt (Kapitel 5).

8.2 ERGEBNISSE BEI SEITENKANALATTACKEN

Somit kommen wir zu den Ergebnissen auf dem Gebiet von Seitenkanalattacken. Hier möchte ich zwei Veröffentlichungen vorstellen, die für meine Arbeit sehr wichtig sind. Die erste ist von Boris Köpf und David Basin [6]. Sie präsentieren ein Model für die adaptiven Attacken und verschiedene Maße des Informationsgehalts sowie die Greedy-Strategie, die zwar schneller berechnet werden kann als die optimale, aber auch beliebig schlechter sein kann. Dabei ist die Greedy-Strategie

eine lokal-optimale Strategie. Sie sucht für den aktuellen Schritt den besten Zug aus um den maximalen Informationsgewinn für einen Schritt zu erreichen. Dabei wird der Informationsgewinn mit einem beliebigen aber festen Maß bestimmt. Die optimale Strategie ist dagegen global-optimal. Sie wählt die für eine bestimmte Anzahl der Züge k die beste Strategie aus um am Ende den höchsten Informationsgewinn zu erreichen.

Abschliessend kommen wir zum Paper von Mohammad Mannan und P.C. Oorschot [8]. Sie stellen unter anderem eine sehr übersichtliche Zusammenfassung von verschiedenen Seitenkanalattacken auf Sicherheit-APIs auf. Sie stellen auch eine Schutzmassnahme gegen die Attacken auf. Dabei ist das Vorgehen, anders als bei uns gewesen. Sie versuchen die Komplezität so zu erhöhen, dass die Attacken signifikant mehr Zeit brauchen.

9 FAZIT

Abschliessend möchte ich die Ergebnisse der Arbeit auflisten und die möglichen zukünftige Arbeiten vorstellen.

Wir haben Mastermind als eine Seitenkanalattacke mit Hilfe von der Greedy-Strategie analysiert und Ergebnisse bekommen, die wir mit bereits bestehenden Ergebnissen verglichen. Für die erwartete Anzahl der Schritte waren unsere Resultate schlechter als die, der besten Strategie, aber für die kleinste obere Schranke an notwendigen Anfragen konnten wir die bisjetzt bestehenden Resultate teilweise verbessern.

Daraufhin haben wir nach hinreichenden Bedingungen gesucht, bei denen Greedy-Strategie optimal ist. Wir haben eine gefunden und sie bewiesen. Aber es besteht noch sehr viel Potenzial in dieser Richtung. Man könnte untersuchen, welche weitere Bedingungen noch existieren und ob es auch notwendige Bedingungen gibt.

Am Ende entwickelten wir eine Schutzmassnahme gegen die Seitenkanalattacke auf VISA PIN Verifikationsverfahren. Hier könnte man untersuchen, ob es eine explizite Verschlüsselung gibt, die unsere Anforderung erfüllt und ob es weitere Attacken danach möglich sind.

Somit entstehen noch viele Möglichkeiten weiter auf diesem Gebiet zu forschen.

BIBLIOGRAPHIE

[1] V. Chvátal. Mastermind. *Combinatorica*, 3:325–329, 1983. (Cited on page 23.)

[2] M.M. Flood. Mastermind Strategy. *Journal of recreational mathematics*, (18):194–202, 1985. (Cited on page 24.)

[3] Wayne Goddard. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 2004. (Cited on pages 14 and 24.)

[4] Michael T. Goodrich. The Mastermind Attack on Genomic Data. (Cited on page 1.)

[5] D.E. Knuth. The Computer as Mastermind. *Journal of recreational mathematics*, (9):1–6, 1976. (Cited on page 23.)

[6] Boris Köpf and David Basin. Automaticaly Deriving Information-theoretic Bounds for Adaptive Side-channel Attack. 2009. (Cited on pages 1, 5, 16, and 24.)

[7] L. Koyama and T.W. Lai. An optimal Mastermind Strategy. *Journal of recreational mathematics*, (25):251–256, 1993. (Cited on page 24.)

[8] Mohammad Mannan and P.C. van Oorschot. Weighing Down "The Unbearable Lightness of PIN Cracking". 2008. (Cited on pages 1, 19, and 25.)

[9] Graham Steel. Formal Analysis of Pin Block Attacks. (Cited on page 1.)

[10] Jeff Stuckman and Guo-Qiang Zhang. Mastermind is NP-Complete. (Cited on page 1.)

KOLOPHON

Diese Arbeit war meine erste eingeständige wissenschaftliche Ausarbeitung. Sowohl das schreiben in LaTeXals auch das Programmieren in *Haskell* waren für mich wie die Entdeckung von Amerika für Christopher Kolumbus. Ich habe gemerkt, was es heisst etwas zu erforschen für ein Thema, das vorher von niemandem unter diesem Aspekt betrachtet wurde.

Es gab auch Zeiten, in denen ich dachte, dass ich nicht weiterkommen würde. Aber auch diese Zeiten vergehen. Ich habe viel gelernt über diese Zeit. Und an diesem Punkt, an dem ich jetzt stehe, wird mir bewusst, dass mit dem Abschluss dieser Zeilen auch ein Teil meines Studiums dem Ende naht.

So traurig es auch ist, freue mich darauf, das nächste Kapitel in meinem Leben und meiner Laufbahn aufzuschlagen und hoffentlich noch mehr Wissen auf der Spur zu sein.

EIDENSTATTLICHE ERKLÄRUNG

Ich versichere, die Bachelorarbeit selbständig und lediglich unter Benutzung der angegebenen Quellen und Hilfsmittel verfasst zu haben. Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht im Rahmen eines anderen Prüfungsverfahrens eingereicht wurde.

Saarbrücken, 21. Dezember 2009

Rajiv Lund