Gottfried Wilhelm Leibniz Universität Hannover
Institut für Theoretische Informatik

Kollisionen in kryptographischen Hashfunktionen

Bachelorarbeit

Max Mathes
Matrikelnr. 10021910

Hannover, den 21. August 2021

Erstprüfer: Dr. Arne Meier
Zweitprüfer: Prof. Dr. Heribert Vollmer
Betreuer: Dr. Arne Meier

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 21. August 2021

_____

Max Mathes

Inhaltsverzeichnis

1 Einleitung

IT-Sicherheit ist ein Thema, welches immer wieder Aufmerksamkeit erregt und durch die fortschreitende Digitalisierung und Entwicklungen wie das „Internet of Things“ nur an Wichtigkeit wächst. Einer der Hauptpfeiler von Sicherheit in der Informatik besteht in der Kryptographie, ohne die es nicht möglich wäre, Schutzziele wie Vertraulichkeit, Integrität und Authentizität zu erfüllen.

Die Möglichkeit zur verschlüsselten Kommunikation von Daten wird in unserer Gesellschaft immer essentieller, während sie unbemerkt schon fast Teil jedweder Interaktion mit oder über technische Geräte geworden ist. Doch während nicht nur die Menge der Anwendungsmöglichkeiten für kryptographische Verfahren wächst, so wächst auch die Menge an Daten, die verarbeitet werden muss. In der modernen Kryptographie herrscht ein stetiges Abwägen zwischen Sicherheit und Effizienz. Ist ein Verfahren nicht sicher genug, so sollte man es nicht verwenden, ist es nicht effizient genug, so will man es nicht verwenden.

Sichere kryptographische Hashfunktionen bieten die Möglichkeit, große wie auch kleinere Mengen an Daten durch kleine Werte zu repräsentieren, welche anschließend austauschbar mit ihren ursprünglichen Daten behandelt werden können. Die entstehenden kleinen Werte können einerseits genutzt werden, um die Menge der zu verarbeitenden Daten massiv zu reduzieren und andererseits eine Anonymisierung bieten, da von den kleinen Werten nicht auf die ursprünglichen Daten geschlossen werden kann. Hashfunktionen finden in fast allen kryptographischen Protokollen des Internets Anwendung. Sie werden aber auch genutzt, um die Integrität von Daten einfach zu überprüfen oder Nutzer-Passwörter sicher speichern zu können. Schlicht gesagt ist es in unserer Gesellschaft kaum möglich, einen Tag zu verbringen, ohne von der Existenz sicherer Hashfunktionen abhängig zu sein.

Die Sicherheit einer Hashfunktion hängt maßgeblich davon ab, dass die Zuordnung zwischen ursprünglichen Daten und Hashwert eindeutig bleibt, da der Hashwert repräsentativ für die Daten verwendet wird. Sobald mehrere Daten vorliegen, die denselben Hashwert ergeben, sogenannte Kollisionen, können daher die Daten untereinander ausgewechselt werden, ohne das Verfahren, die nur den Hashwert der Daten betrachten, eine Möglichkeit haben, diese Wechsel festzustellen. Dies kann für einige Anwendungsfälle selbst dann zum Problem werden, wenn die Daten keinerlei logisch strukturierten Aufbau besitzen [CW03].

Eine Hashfunktion zu entwerfen, für die keine Kollisionen gefunden werden können, ist eine herausfordernde Aufgabe, da nicht nur die Sicherheit des Funktionsaufbaus sondern auch die Sicherheit der intern in der Hashfunktion verwendeten kryptographischen Verfahren

relevant sind. Zusätzlich ist die Existenz von Kollisionen für jede Hashfunktion garantiert, da es wesentlich mehr mögliche Eingaben als Ausgaben gibt.

In dieser Arbeit wird zuerst ein Überblick über Hashfunktionen, ihre Konstruktion mit Hilfe der Merkle-Damgård-Konstruktion und unterschiedliche Angriffsmöglichkeiten gegen Hashfunktionen gegeben. Anschließend wird die Frage thematisiert, ob es möglich ist, eine sichere Hashfunktion zu entwickeln, die auf unsicheren kryptographischen Verfahren aufbaut. Dafür werden Hashfunktionen implementiert, die auf gebrochenen Kryptosystemen aufbauen und es wird versucht Kollisionen in diesen Funktionen zu finden. Abschließend werden die Erkenntnisse aus diesen Ansätzen zusammengefasst und ein Ausblick über weitere Möglichkeiten, diese Arbeit fortzusetzen, gegeben.

2 Grundlagen

In diesem Kapitel werden zuerst die Grundlagen von Hashfunktionen und deren Konstruktion mit Hilfe des Merkle-Damgård Schemas vorgestellt. Anschließend wird in unterschiedliche Angriffe gegen die Sicherheit von Hashfunktionen eingeleitet. Vieles auftauchende Wissen ist in diversen Quellen zu finden, diese Arbeit orientiert sich aber an der *Einführung in die Kryptographie* von Johannes Buchmann [Buc16].

Praktische Unmöglichkeit bedeutet im Verlauf dieser Arbeit, dass kein probabilistischer Algorithmus bekannt sein darf, der das zugrunde liegende Problem in polynomieller Laufzeit löst. Sämtliche auf Angreiferseite zur Verfügung stehenden Algorithmen sollen also probabilistische Polynomialzeit-Algorithmen sein. Wenn etwas als praktisch unmöglich gilt, dann kann ein solcher Algorithmus es nicht in annehmbarer Zeit lösen.

Eine praxisbezogene Einschätzung können Sicherheitsparameter bieten. Dabei bedeutet ein Sicherheitsparameter a, dass 2^a Operationen zum Brechen des zugrunde liegenden Kryptosystems nötig sind [Len04]. In Tabelle 2.1 ist zu erkennen, welche Sicherheitsparameter bis zu welchem Jahr als sicher angesehen werden können [oEiCI12].

Schutz bis zum Jahr	Sicherheitsparameter a
2030	112
2040	128
absehbare Zukunft	256

Tabelle 2.1: Sicherheitsparameter

Für eine ausführlichere mathematische Analyse der praktischen Unmöglichkeit sei auf die Werke *Kryptographie* [Wä18, S. 11-13] von Dietmar Wätjen und *Moderne Kryptographie* [KW11, S. 194-196] von Ralf Küsters und Thomas Wilke verwiesen.

Wenn in dieser Arbeit von zufälligen Werten die Rede ist, so sind, wenn nicht ausdrücklich gekennzeichnet, pseudo-zufällige Werte gemeint, die leicht mit einem Computer generiert werden können.

Der Operator $\|$ steht in dieser Arbeit für die Konkatenation zweier Werte. Für $X =$„abc“ und $Y =$„def“ gilt $(X \| Y) =$ „$abcdef$“.

Der Operator $\oplus$ steht in dieser Arbeit für das exklusive Oder zweier Werte. Für $X = 0110$ und $Y = 1010$ gilt $(X \oplus Y) = 1100$.

2.1 Hashfunktionen

Beginnen wir zunächst mit der Definition einer Hashfunktion und den Charakteristiken, die eine kryptographische Hashfunktion ausmachen.

Definition 2.1. Bei einer *Hashfunktion* handelt es sich um eine Funktion, die von einer Eingabe x beliebiger Länge auf eine Ausgabe $h(x)$ fester Länge abbildet. Dabei ist, wie in der folgenden Arbeit, Σ ein Alphabet.

$$h\colon \Sigma^* \to \Sigma^n, \quad n \in \mathbb{N}$$

Beispiel 2.1. Die Funktion $h(x) = x_0$ für $x = x_0x_1...x_n$, welche sämtliche Eingaben auf ihre erste Stelle abbildet, kann also, dieser Definition folgend, bereits als Hashfunktion nach 2.1 betrachtet werden.

Da der Definitionsbereich einer Hashfunktion zwingend größer ist als der Bildbereich, kann es sich nie um eine injektive Funktion handeln. Es muss also verschiedene Eingabewerte geben, die auf denselben Ausgabewert (im Folgenden auch Hashwert genannt) abgebildet werden. Eine solche Abbildung unterschiedlicher Eingaben auf denselben Hashwert wird Kollision genannt.

Definition 2.2. Eine *Kollision* innerhalb einer Hashfunktion tritt auf, wenn für zwei Eingaben x und x' gilt:

$$x \neq x' \text{ und } h(x) = h(x'), \quad x, x' \in \Sigma$$

Um nun zu verstehen, warum eine solche Kollision potenziell ein Problem für die Verwendung der Hashfunktion darstellen kann, sehen wir uns zunächst eine praktische Einsatzmöglichkeit an.

Beispiel 2.2. Angenommen, Alice speichert ihre tägliche Arbeitszeit zu Abrechnungszwecken in einem Dokument ab. Da der Computer außerhalb ihrer Arbeitszeit von anderen Personen verwendet wird und sie kontrollieren möchte, dass das Dokument nicht verändert wurde, notiert sie sich abends den Hashwert der Datei und überprüft morgens, ob die Datei immer noch denselben Hashwert ergibt. Nehmen wir nun an, dass Alice die Hashfunktion aus Beispiel 2.1 verwendet. Trotz ihrer Sicherheitsvorkehrungen wird Alice eine Veränderung von „Arbeitszeit Montag: 10 Stunden“ zu „Arbeitszeit Montag: 2 Stunden“ nicht bemerken, da die Hashfunktion für beide Variationen des Dokuments den Wert „A“ zurückgeben wird. Die Verwendung dieser Hashfunktion, für die das Finden von Kollisionen trivial ist, kann für Alice also potenziell fatale Folgen haben.

Es muss also für die Verwendung von Hashfunktionen, in sicherheitsrelevanten Kontexten, noch weitere Anforderungen geben, welche wir uns im Folgenden ansehen werden.

Eine Hashfunktion heißt *Einwegfunktion*, wenn es für einen gegebenen Hashwert y praktisch

unmöglich ist, eine Eingabe $x \in \Sigma$ zu finden, für die gilt $h(x) = y$.

Des Weiteren gilt eine Hashfunktion als *schwach kollisionsresistent*, wenn es für eine gegebene Eingabe x praktisch unmöglich ist, eine zweite Eingabe x' mit $x \neq x'$ zu finden, für die gilt $h(x) = h(x')$.

Darüber hinaus ist eine Hashfunktion als *stark kollisionsresistent* anzusehen, wenn es praktisch unmöglich ist, zwei beliebige Eingaben x, x' mit $x \neq x'$ zu finden, für die gilt $h(x) = h(x')$. Die starke Kollisionsresistenz schließt immer auch die schwache Kollisionsresistenz mit ein [Buc16, S. 235].

Die *ideale Hashfunktion* ist als Zufallsorakel definiert. Sie gibt für eine Eingabe entweder eine aus ihrem Ausgabebereich perfekt zufällig ausgewählte Ausgabe oder den Wert, der vorher bereits für diese Eingabe ausgegeben wurde, aus.

2.2 Merkle-Damgård-Konstruktion

Sehen wir uns nun an, wie eine Hashfunktion als Funktion, die Eingaben jedweder Länge akzeptiert, aufgebaut werden kann. Dafür beginnen wir mit der Definition einer Kompressionsfunktion $k(x)$.

Definition 2.3. Eine *Kompressionsfunktion* ist eine Funktion, die von einer Eingabe x fester Länge auf eine Ausgabe $k(x)$ kürzerer Länge abbildet. b bezeichnet dabei die Blockgröße der Kompressionsfunktion.

$$k\colon \Sigma^{a+b} \longrightarrow \Sigma^{a}, \quad a, b \in \mathbb{N}, \quad a, b > 0$$

Die Ausgabe einer Kompressionsfunktion wird im Folgenden H genannt. Die Merkle-Damgård-Konstruktion beschreibt ein Verfahren, wie man durch iteratives Anwenden einer solchen Kompressionsfunktion eine Hashfunktion, die Eingaben beliebiger Länge akzeptiert, konstruieren kann. Für das Verfahren wird die zu verarbeitende Eingabe m zuerst durch ein sogenanntes Padding auf eine Länge erweitert, die einem Vielfachen der Blockgröße b der Kompressionsfunktion entspricht. Ein mögliches Padding besteht darin, an die Eingabe eine beliebige Anzahl an 0*en*, gefolgt von der ursprünglichen Länge der Eingabe, in Binärdarstellung, anzuhängen. Das Anhängen der ursprünglichen Länge ist wichtig, damit die verschiedenen Eingaben m und $m \,||\, 0$ auch nach dem Padding unterscheidbar bleiben. Die Eingabe m ist also nach der Anwendung des Paddings definiert als $m' = m \,||\, 0^k \,||\, |m|, \quad \text{mit } 0 \leq k \leq b - 1.$

Beispiel 2.3. Angenommen, wir haben eine Blockgröße von 8 und m = „101“, so wäre $m' = m \,||\, 0^k \,||\, |m| = 101 \,||\, 000 \,||\, 11 = 10100011$.

Nachdem die Eingabe auf die benötigte Länge gebracht wurde, wird sie nun in Blöcke aufgeteilt, deren Länge jeweils der Blockgröße b entspricht. Wenn die Eingabe m also in n Blöcke der Größe b aufgeteilt wird, ergibt sich $m' = m_1 \,||\, m_2 \,||\, ... \,||\, m_n$ mit $|m_i| = b$.

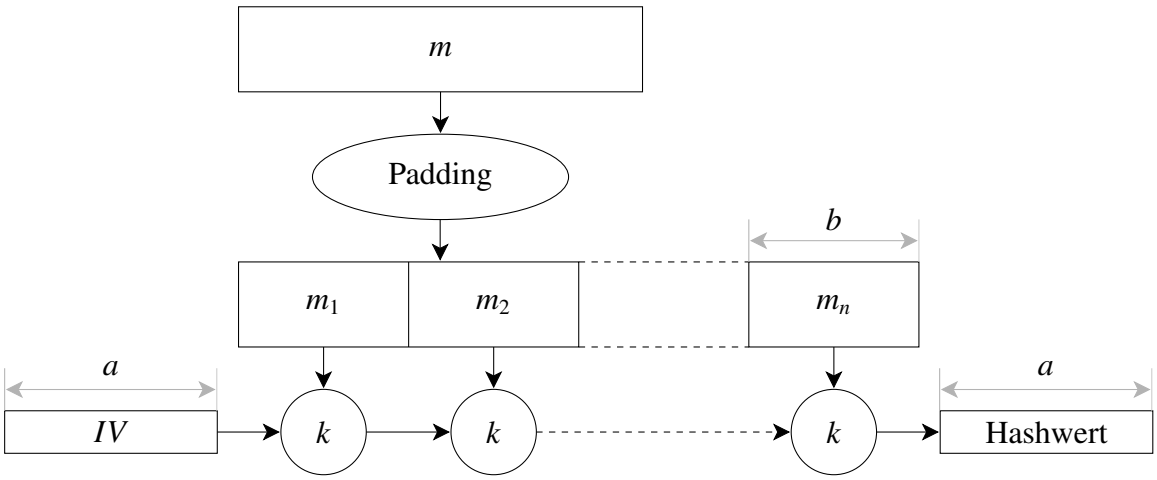

Abbildung 2.1: Merkle-Damgård-Konstruktion

Anschließend wird nun iterativ die Kompressionsfunktion auf die Ausgabe H_{i-1} der vorherigen Iteration und den nächsten Block der Eingabe m_i aufgerufen. Für den Wert H_i gilt $H_i = k(m_i, H_{i-1})$. Nachdem der letzte Block der Eingabe verarbeitet wurde, liegt als H_n das Ergebnis der konstruierten Hashfunktion vor. Sämtliche Hashwerte H_1 bis H_{n-1}, die nicht ausgegeben werden, werden in dieser Arbeit interne Hashwerte genannt. Da es den Wert H_0 nicht gibt, es gibt keine 0*te* Iteration, wird stattdessen ein Initialisierungsvektor IV verwendet. Dieser kann einen beliebigen Wert enthalten. Wird in dieser Arbeit eine Hashfunktion $h(v, x)$ mit zwei Parametern aufgerufen, so ist v der Initialisierungsvektor und x die Eingabe.

In Abbildung 2.1 ist das Verfahren graphisch dargestellt. Es ist leicht zu erkennen, dass die Anzahl der benötigten Iterationen stark mit der Blockgröße b der Kompressionsfunktion zusammenhängt.

Der besondere Vorteil dieser Konstruktion einer Hashfunktion aus dem iterativen Anwenden einer Kompressionsfunktion besteht nun darin, dass die resultierende Hashfunktion $h(x)$ beweisbar stark kollisionsresistent ist, wenn die verwendete Kompressionsfunktion $k(x)$ stark kollisionsresistent ist.

Satz 2.1. *Eine Hashfunktion nach der Merkle-Damgård-Konstruktion, ist beweisbar stark kollisionsresistent, wenn die verwendete Kompressionsfunktion stark kollisionsresistent ist.*

Beweis. Um dies zu beweisen, zeigen wir, dass man aus einer Kollision in h auch eine Kollision in k herleiten kann, was der starken Kollisionsresistenz von k widerspricht.

Nehmen wir nun an, (x, x') sei eine Kollision in h und $x_1, ..., x_t, x'_1, ..., x'_{t'}$ seien die dazugehörigen Zerlegungen in Blöcke der Größe b. Die entsprechenden Folgen von Hashwerten seien $H_1, ..., H_t, H'_1, ..., H'_{t'}$, da (x, x') eine Kollision ist, gilt $H_t = H'_{t'}$ und $x \neq x'$.

Wir vergleichen nun H_{t-i} mit $H'_{t'-i}$ für $i = 1, 2, ..., t-1$, bis wir einen Block finden für den gilt

$$H_{t-i} = H'_{t'-i} \text{ und } x_{t-i} \neq x'_{t'-i}$$

So ein Block existiert, da $x \neq x'$. Daraus folgt nun

$$H_{t-i-1} \circ x_{t-i} \neq H'_{t'-i-1} \circ x'_{t'-i}$$

und

$$k(H_{t-i-1} \circ x_{t-i}) = H_{t-i} = H'_{t'-i} = k(H'_{t'-i-1} \circ x'_{t'-i})$$

was eine Kollision in k ist. Für eine ausführlichere Darstellung des Beweises sei auf J. Buchmann verwiesen [Buc16, S. 237-239]. ■

Die Sicherheit, der resultierenden Hashfunktion, hängt also von der Sicherheit der verwendeten Kompressionsfunktion ab. Da nicht bekannt ist, ob es Kompressionsfunktionen gibt, die tatsächlich kollisionsresistent sind, werden in der Praxis Verfahren angewendet, deren Kollisionsresistenz noch nicht widerlegt ist. Solche Kompressionsfunktionen können zum Beispiel aus Blockchiffren konstruiert oder direkt für die Verwendung in einer Hashfunktion erstellt werden.

Das beschriebene Verfahren wurde in R. Merkle's Dissertation vorgestellt [Mer79, S. 13-15] und die Tatsache, dass die resultierende Hashfunktion stark kollisionsresistent ist, wenn die verwendete Kompressionsfunktion stark kollisionsresistent ist, von R. Merkle [Mer90] und I. Damgård [Dam90] unabhängig voneinander bewiesen, weshalb das Verfahren als Merkle-Damgård-Konstruktion (MD-Konstruktion) bekannt ist.

2.2.1 Erweiterungen

Im Folgenden werden kurz zwei Erweiterungen der Merkle-Damgård-Konstruktion vorgestellt, die gewisse strukturelle Schwächen der Konstruktion gegen einige Angriffsmethoden abmildern.

Beim Wide-Pipe Verfahren wird der interne Zustand der Hashfunktion erweitert, sodass er nicht mehr der Größe des ausgegebenen Hashwertes entspricht, sondern einen höheren Wert annimmt, beispielsweise doppelt so groß. Um vom letzten internen Wert auf den auszugebenden Hashwert zu kommen, muss also zusätzlich noch weiter komprimiert werden, was zum Beispiel durch einfaches Weglassen von Teilen der Ausgabe möglich ist. Das Wide-Pipe Verfahren erhöht die Sicherheit gegen Multikollisions- und Length-Extension-Angriffe [Luc04].

Das Fast-Wide-Pipe Verfahren erweitert die MD-Konstruktion derart, dass in jeder Iteration die Ausgabe der Kompressionsfunktion nicht direkt weitergegeben, sondern vorher mit der Ausgabe der vorherigen Iteration XOR-Verknüpft wird. Das Verfahren erzielt fast die doppelte Geschwindigkeit des Wide-Pipe Verfahrens [NP10].

2.3 Angriffe auf Hashfunktionen

Im Folgenden werden einige Angriffe auf Hashfunktionen vorgestellt. Ein Angriff ist dabei ein Verfahren, welches die Eigenschaften der Hashfunktion als Einwegfunktion oder als schwach oder stark kollisionsresistente Funktion außer Kraft setzt. Also aus einem Hashwert den Eingabewert zurückgewinnt, eine zweite Eingabe mit identischem Hashwert zu einem gegebenen Eingabe/Hashwert-Paar ermittelt oder eine beliebige Kollision in der Hashfunktion findet. Es ist außerdem vorausgesetzt, dass ein Angriff eine geringere Komplexität besitzt als der Sicherheitsanspruch der Hashfunktion vorgibt.

Beispiel 2.4. In einer Hashfunktion, die auf 8 Bit lange Hashwerte abbildet, wäre nach $2^8 + 1$ unterschiedlichen Eingaben zwingend eine Kollision gefunden, da die Menge der unterschiedlichen Hashwerte erschöpft ist. Ein Verfahren, das nach 2^{11} Operationen eine Kollision in dieser Hashfunktion findet, kann also nicht als Angriff bezeichnet werden.

Das Zurückgewinnen des Eingabewertes einer Hashfunktion stellt zum Beispiel für die Speicherung von Passwörtern als Hashwerten eine Bedrohung dar, da die verwendeten Passwörter trotz dieser Sicherheitsmaßnahme von einem Angreifer ermittelt werden könnten. Für die Auswirkungen von Kollisionen müssen wir einen Blick auf die praktischen Anwendungen von Hashfunktionen werfen. Zum Beispiel sind digitale Signaturverfahren nicht in der Lage, große Mengen an Daten effizient zu signieren, weshalb viele Implementationen den Hashwert der vorgelegten Daten bilden und diesen signieren. Gelingt es nun, zwei Dokumente A und B zu erzeugen, die denselben Hashwert besitzen, dann ist die Signatur von Dokument A auch für Dokument B gültig. Auf diese Weise ließen sich zum Beispiel TLS-Zertifikate fälschen [SSA+09].

Bei sämtlichen Angriffen muss für den praktischen Gebrauch unterschieden werden, ob der Angriff nur theoretisch durchführbar ist oder ob die Komplexität tatsächlich gering genug ist, um auf einem real existierenden System in absehbarer Zukunft ausgeführt zu werden.

2.3.1 Geburtstagsparadoxon und Brute-Force

Der wohl simpelste Ansatz zum Finden einer Kollision besteht darin, so lange neue Hashwerte zu berechnen, bis ein schon bekannter gefunden wurde. Es ist trivial zu erkennen, dass dies bei einer Hashfunktion mit 2^n Ausgabemöglichkeiten spätestens nach $2^n + 1$ Versuchen der Fall sein muss. Unintuitiv erscheint dagegen die Tatsache, dass mit einer Wahrscheinlichkeit von 50% bereits nach $2^{n/2}$ Versuchen eine Kollision gefunden wurde. Um diesen Zusammenhang zu verstehen, sehen wir uns das Geburtstagsparadoxon an, was sich mit der Frage: "Wie viele Personen müssen in einem Raum sein, damit mit einer Wahrscheinlichkeit von mindestens $\frac{1}{2}$ wenigstens zwei von ihnen am gleichen Tag Geburtstag haben?" beschäftigt.

Um die Wahrscheinlichkeit für einen gleichen Geburtstag zu berechnen, sieht man sich die

Gegenwahrscheinlichkeit an. Bei n Personen wäre die Wahrscheinlichkeit für nur unterschiedliche Geburtstage

$$\frac{365}{365} \cdot \frac{365-1}{365} \cdot \ldots \cdot \frac{365-(n-1)}{365}$$

Sobald $n \geq 23$ ist wird diese Wahrscheinlichkeit kleiner als 50% und somit die Wahrscheinlichkeit, dass zwei oder mehr Personen am gleichen Tag Geburtstag haben, größer als 50%. Allgemein gilt, dass bei n verschiedenen Ausgabewerten, seien es Geburtstage oder Hashwerte, nur etwas mehr als $\sqrt{n}$ Eingabewerte, seien es Personen oder Eingaben einer Hashfunktion, benötigt werden, damit mit hoher Wahrscheinlichkeit eine Dopplung der Ausgaben vorliegt. Für eine ausführlichere Darstellung sei auf Buchmann [Buc16, S. 16-17] verwiesen. Eine Kollision in einer Hashfunktion ist also immer schon nach $2^{n/2}$ Brute-Force Versuchen zu erwarten, weshalb der Sicherheitsanspruch einer kryptographischen Hashfunktion mit 2^n Ausgabemöglichkeiten immer durch $2^{n/2}$ definiert ist. Die Ausgabelänge einer Hashfunktion muss also immer doppelt so groß sein wie in Tabelle 2.1 vorgegeben, was in Tabelle 2.2 gezeigt wird.

Schutz bis zum Jahr	mindeste Hashlänge
2030	224
2040	256
absehbare Zukunft	512

Tabelle 2.2: mindeste Hashlänge gegen Brute-Force Angriffe [oEiCI12]

Das Geburtstagsparadoxon wirkt sich nur auf das Finden von vollständig beliebigen Kollisionen aus. Das Bestimmen eines zweiten Eingabewertes, der denselben Hashwert ergibt wie eine vorgegebene Eingabe, ist davon nicht betroffen, da die gefundenen Übereinstimmungen dafür nicht mehr beliebig sein können, sondern mit dem konkreten Hashwert übereinstimmen müssen. Ein Geburtstagsangriff beschreibt im Folgenden einen Brute-Force Angriff, der aufgrund der beschriebenen Zusammenhänge eine geringere Komplexität benötigt, als auf den ersten Blick anzunehmen wäre.

2.3.2 Chosen-Prefix-Angriff

Die meisten Kollisionen sind für einen Angriff auf reale Anwendungen von Hashfunktionen nicht nutzbar, da die zugehörigen Eingaben nur aus unstrukturierten Bitfolgen bestehen. Ein Chosen-Prefix Angriff hat das Ziel, zu zwei beliebigen Präfixen P_1 und P_2 zwei Eingaben X_1 und X_2 zu finden, sodass $h(P_1 \parallel X_1) = h(P_2 \parallel X_2)$ gilt [Sch20]. Dieser Angriff ist wesentlich mächtiger, da so Kollisionen, die einer in der Praxis verwendeten Struktur folgen, gefunden werden können.

Beispiel 2.5. Angenommen unser Angreifer Bob findet eine Kollision mit

$$h(\text{„Alice Public Key ist:“} \parallel \text{„A6h7FH3z“}) = h(\text{„Bobs Public Key ist:“} \parallel \text{„1-ghZ7d9q“})$$

Bob kann nun den Schlüssel K_{Bob} von einer CA zertifizieren lassen. Da beide Dokumente denselben Hashwert ergeben, ist die Signatur von K_{Bob} auch für K_{Alice} gültig. Anschließend kann Bob sich mit der kopierten Signatur und dem zertifizierten Schlüssel K_{Alice} als Alice ausgeben.

Im Jahr 2020 konnte die erste praktische Anwendung eines Chosen-Prefix Angriff gegen SHA-1 durchgeführt werden. Bei dem Angriff wurden zwei PGP-Schlüssel mit unterschiedlichen Identitäten, die dasselbe SHA-1 Zertifikat ergeben, erstellt [LP20].

2.3.3 Length-Extension-Angriff

Ein Length-Extension Angriff gibt dem Angreifer die Möglichkeit, aus einem bekannten Hashwert $h(x)$ und dem Wissen über die Länge von x einen neuen Hashwert $h(x \parallel x')$ zu berechnen, ohne dafür den Inhalt von x zu kennen. Dieser Angriff ist möglich, da die Hashfunktion intern iterativ abläuft und der Angreifer einfach den Hashwert von x' mit dem bekannten Hashwert $h(x)$ als Initialisierungsvektor berechnen kann, anstatt die Eingabe $x \parallel x'$ zu hashen. Das Wissen über die Länge von x ist notwendig, um ein gültiges Padding für die konstruierte Eingabe $x \parallel x'$ zu erstellen.

In der Praxis ist so ein Angriff zum Beispiel auf einige Formen von Message Authentication Codes (MACs) anwendbar, bei denen zur Authentifizierung zusätzlich zur Nachricht der Hashwert $h(secret + message)$ der Nachricht und eines geheimen Schlüssels übertragen wird. Da die Länge der Nachricht mit Übertragung öffentlich wird, muss der Angreifer nur die Länge des Secrets erraten, um anschließend für neue Nachrichten $message \parallel evilAppendix$ gültige Authentifizierungen $h(secret+message+evilAppendix)$ zu erzeugen. Hashfunktionen, die der Merkle-Damgård-Konstruktion folgen und deren ausgegebener Hashwert identisch zum letzten internen Hashwert ist, sind anfällig gegen diesen Angriff.

2.3.4 Multikollisionsangriff

Wenn eine Kollision zwei unterschiedliche Eingabewerte bezeichnet, die denselben Hashwert ergeben, so bezeichnet eine a-Kollision a unterschiedliche Eingabewerte, die alle denselben Hashwert ergeben. Wenn die verwendete Hashfunktion ideal ist, so werden mindestens $2^{(a-1)n/a}$ Operationen benötigt, um eine a-Kollision zu finden [Luc04, S. 4]. Ein Multikollisionsangriff bedient sich des iterativen Aufbaus einer Hashfunktion, um eine 2^a-Kollision in nur a mal längerer Zeit als eine normale Kollision zu finden. Eine 2^a-Kollision ist also nach $a \cdot 2^{n/2}$ Operationen zu erwarten.

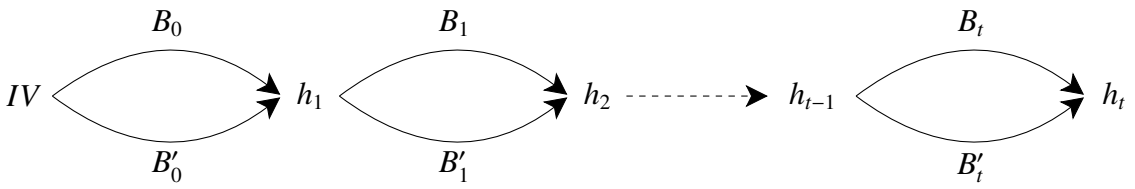

Abbildung 2.2: Konstruktion von Multikollisionen

Für den Angriff wird ein Algorithmus C verwendet, der mit Eingabe von einem Initialisierungsvektor IV für h zwei Eingabeblöcke B und B' findet, sodass $h(IV, B) = h(IV, B')$ gilt. Der Algorithmus kann dafür beispielsweise einen Geburtstagsangriff ausführen. Nachdem nun C mit einem beliebigen IV ausgeführt wurde und zwei Eingabeblöcke B_0 und B'_0 gefunden wurden, für die $h(IV, B_0) = h(IV, B'_0) = y$ gilt, wird C anschließend mit y als Initialisierungsvektor ausgeführt. C findet nun wieder zwei Eingabeblöcke B_1 und B'_1, für die $h(y, B_1) = h(y, B'_1)$ gilt. Aus diesen zwei Kollisionen lässt sich nun bereits die folgende 4-Kollision herleiten:

$$h(h(IV, B_0), B_1) = h(h(IV, B_0), B'_1) = h(h(IV, B'_0), B_1) = h(h(IV, B'_0), B'_1)$$

Aufgrund des iterativen Aufbaus von h lässt sich dieses Verfahren beliebig vergrößern. Das liegt daran, dass, sobald in einer Iteration einmal ein identischer interner Hashwert erreicht wurde, eine Veränderung des finalen Hashwertes nur durch unterschiedliche Eingabeblöcke möglich ist. Abbildung 2.2 zeigt das von A. Joux vorgestellte Verfahren [Jou04].

Nehmen wir an, $h(x)$ ist eine Hashfunktion, die zur Steigerung der Sicherheit längere Hashwerte durch Konkatenation von mehreren einzelnen Hashfunktionen erzeugt $h(x) = f(x) \parallel g(x)$. Solange $f(x)$ und $g(x)$ iterative Hashfunktionen sind, lässt sich mit diesem Angriff zeigen, dass die Sicherheit von $h(x)$ nicht wesentlich größer als die alleinige Sicherheit von $f(x)$ oder $g(x)$ ist, obwohl $h(x)$ eine wesentlich längere Ausgabe besitzt [Jou04, S.310-312].

2.3.5 Herding-Angriff

Die von Kelsey und Kohno vorgestellten Herding-Angriffe [KK06] erweitern Chosen-Prefix-Angriffe derart, dass zu einem gegebenen Hashwert H und einer vorgegebenen Eingabe P eine weitere Eingabe S gefunden werden muss, sodass $h(P \parallel S) = H$ gilt. Es ist damit also möglich, den Hashwert einer Eingabe durch Anhängen eines Suffixes zu jedem beliebigen Hashwert zu verändern. Ein Herding-Angriff wird durch die Ausgabelänge n und den Wert k parametrisiert.

Im ersten Schritt des Angriffs werden wiederholt Kollisionen in der Hashfunktion gefunden (zum Beispiel mit einem Geburtstagsangriff) und aus diesen eine Diamant-Struktur angelegt.

Eine Diamant-Struktur ist dabei ein Binärbaum, dessen Knoten Hashwerten und dessen

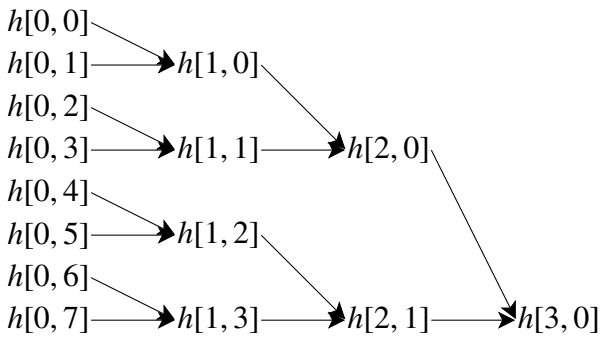

Abbildung 2.3: Diamant-Struktur der Breite $k = 3$

Kanten Eingabeblöcken entsprechen. Eine Kante von einem Knoten K_1 zu einem Knoten K_2 stellt dabei die Eingabe dar, die benötigt wird, um vom Hashwert des Knotens K_1 zum Hashwert des Knotens K_2 zu gelangen.

Mit hoher Wahrscheinlichkeit ist nach $2^{k/2+n/2+2}$ Anwendungen der Kompressionsfunktion, der verwendeten Hashfunktion, eine Diamant-Struktur mit $2^{k+1} - 2$ internen Hashwerten erstellt. Im zweiten Schritt werden so lange Werte für eine Eingabe S' ausprobiert, bis $h(P \parallel S')$ einem der internen Hashwerte der Diamant-Struktur entspricht. Dafür müssen 2^{n-k} Möglichkeiten für S' ausprobiert werden. Anschließend kann aus der Diamant-Struktur eine Folge von Eingabeblöcken Q entnommen werden, sodass mit $S = S' \parallel Q, \quad h(P \parallel S) = H$ gilt. Sämtliche Hashfunktionen, die der Merkle-Damgård-Konstruktion folgen, können auf diese Weise durch wiederholtes Finden von Kollisionen angegriffen werden. Tabelle 2.3 zeigt einige praktische Beispiele dieses Angriffs.

Hashfunktion	Ausgabelänge	Breite der Diamant-Struktur k	Länge von S in Blöcken	Operationen
MD5	128	41	48	2^{87}
SHA-1	160	52	59	2^{108}
Tiger	192	63	70	2^{129}
SHA256	256	84	92	2^{172}
Whirlpool	512	169	178	2^{343}
	n	$\lceil (n-5)/3 \rceil$	$\lceil k + ld(k) + 1 \rceil$	2^{n-k}

Tabelle 2.3: Praktische Anwendung von Herding-Angriffen [KK06]

2.3.6 Long-Message-Angriff

Ein Long-Message-Angriff [MOVR01] verletzt die schwache Kollisionsresistenz von Hashfunktionen. Dafür wird zuerst eine sehr lange Eingabe M_{long} aus $2^R + 1$ Blöcken gehasht und die internen Hashwerte gespeichert. R ist dabei eine sehr große natürliche Zahl. An-

schließend wird über Brute-Force ein Eingabeblock M_{link} ermittelt, sodass $h(M_{link})$ einem der gespeicherten internen Hashwerte an einem beliebigen Index j entspricht. Da insgesamt 2^R interne Hashwerte vorliegen, ist eine Übereinstimmung nach 2^{n-R} Versuchen zu erwarten. Der Block M_{link} stellt eine direkte Verbindung vom Initialisierungsvektor der Hashfunktion zum internen Hashwert H_j der ursprünglichen Eingabe M_{long} dar und somit ist $h(M_{long}) = h(M_{link} \parallel M_{long,j+1} \parallel M_{long,j+2} \parallel ...)$. Dieser Angriff ist gegen Hashfunktionen, deren Padding die Länge der Eingabe enthält, wirkungslos, da M_{long} länger ist als $(M_{link} \parallel M_{long,j+1} \parallel M_{long,j+2} \parallel ...)$. Er kann aber mit einem Expandable-Message-Angriff kombiniert werden.

2.3.7 Expandable-Message-Angriff

Ein Expandable-Message-Angriff [KS05] bedient sich fester Punkte in der Kompressionsfunktion, um unterschiedlich lange Nachrichten zu generieren, die dieselben internen Hashwerte ergeben. Ein fester Punkt in der Kompressionsfunktion k ist dabei ein Paar (H_{fix}, M_{fix}), sodass $H_{fix} = k(H_{fix}, M_{fix})$ gilt. Solche festen Punkte sind in Kompressionsfunktionen, die auf der Davies-Meyer-Konstruktion [Dav85] aufbauen, leicht zu finden [MOVR01]. Um einen festen Punkt in der Hashfunktion anwenden zu können, finden wir zuerst $2^{n/2}$ feste Punkte und ermitteln dann über Brute-Force einen Eingabeblock M, sodass $h(M)$ vom Initialisierungsvektor zu einem der festen Punkte führt. Also $h(M) = H_{fix}$ gilt. Sobald dieser feste Punkt gefunden ist, kann der entsprechende Eingabeblock M_{fix} des festen Punktes beliebig oft wiederholt werden, ohne den internen Hashwert der Eingabe bis vor dem Padding zu verändern. Das Padding verändert den finalen Hashwert, da sich mit Einfügen von Blöcken M_{fix} die Länge der Eingabe verändert hat. Wir können auf diese Weise also mit $2^{n/2+1}$ Operationen Eingaben aus M, gefolgt von beliebig vielen Blöcken M_{fix}, erzeugen, die vor dem Padding alle dieselben internen Hashwerte ergeben.

Dieses Verfahren erweitern wir ähnlich zum Multikollisionsangriff, indem wir zuerst ein Paar ermitteln, das aus einer 1 Block langen und einer $2^{k-1} + 1$ Blöcke langen Eingabe besteht, die beide denselben internen Hashwert H_{tmp} ergeben. Hierbei sei 2^k die maximale Eingabelänge der Hashfunktion in Blöcken. Theoretisch ist die Länge der Eingabe einer Hashfunktion unbeschränkt, praktisch gibt es aber eine maximale Länge, die die Hashfunktion verarbeiten kann. Nun benutzen wir H_{tmp} als neuen Initialisierungsvektor und finden ein Paar mit den Längen 1 und $2^{k-2} + 1$, als nächstes ein Paar mit den Längen 1 und $2^{k-3} + 1$, bis wir bei den Längen 1 und 2 angekommen sind. Mit dieser Liste lässt sich nun eine Eingabe flexibel zwischen k und $k + 2^k - 1$ Blöcke verlängern, indem die Differenz aus momentaner und gewünschter Länge in binär aufgeschrieben und aus jedem Paar abhängig vom entsprechenden Bit die kurze oder die lange Eingabe gewählt wird. Zur Konstruktion dieser Liste von Paaren musste die Kompressionsfunktion in etwa $2^k + k \cdot 2^{n/2+1}$ mal ausgeführt werden.

Wenn wir nun einen Long-Message-Angriff mit einer Expandable-Message kombinieren,

können wir eine Eingabe M_{link} finden, welche den letzten internen Hashwert der Expandable-Message mit einem internen Hashwert der ursprünglichen Eingabe M_{long} verbindet. Die Länge der Expandable-Message kann nun so angepasst werden, dass $|(M_{expandable} \parallel M_{link} \parallel M_{long,j+1} \parallel M_{long,j+2} \parallel ...)| = |M_{long}|$ ist. Das führt dazu, dass die Hashwerte der beiden Eingaben trotz Padding identisch sind, weil die Eingaben die gleiche Länge und somit das gleiche Padding haben. Die schwache Kollisionsresistenz ist verletzt, da der Angriff mit $k \cdot 2^{n/2+1} + 2^{n-k+1}$ Operationen zum Finden der Expandable-Message und $3 \cdot 2^{n/2+1} + 2^{n-k+1}$ Operationen zum Finden von M_{link} insgesamt weniger als 2^n Operationen benötigt, wenn realistische Werte für k verwendet werden. Beispielsweise wäre für SHA512 $n = 512$ und $k = 118$ [KS05, S. 483], damit ergibt sich eine Komplexität K von

$$\begin{aligned} K &= 118 \cdot 2^{512/2+1} + 2^{512-118+1} + 3 \cdot 2^{512/2+1} + 2^{512-118+1} \\ &= 118 \cdot 2^{257} + 2^{395} + 3 \cdot 2^{257} + 2^{395} \\ &< 128 \cdot 2^{257} + 4 \cdot 2^{257} + 2^{395} + 2^{395} \\ &= 2^{264} + 2^{259} + 2^{396} \\ &< 2^{397}, \end{aligned}$$

was deutlich weniger als 2^{512} ist.

2.3.8 Boomerang-Angriff für Hashfunktionen

Der Boomerang-Angriff auf Blockchiffren baut darauf auf, die Differenz zweier unterschiedlicher Eingaben mit der Differenz der dazugehörigen Ausgaben zu vergleichen. Dabei werden die Wahrscheinlichkeiten für Paare von Eingabe- und Ausgabedifferenz betrachtet. Er wird im Folgenden kurz erläutert. Angenommen, wir haben eine differenzielle Charakteristik auf der ersten Hälfte der Chiffre, die besagt, dass eine Eingabedifferenz $\triangle$ mit der Wahrscheinlichkeit p_1 zu einer Ausgabedifferenz $\triangle'$ führt und eine weitere Charakteristik für die zweite Hälfte der Chiffre, die besagt, dass eine Eingabedifferenz $\triangledown$ mit Wahrscheinlichkeit p_2 zu einer Ausgabedifferenz $\triangledown'$ führt. Nun verschlüsseln wir die Texte P_1 und $P_2 = P_1 \oplus \triangle$ und erhalten die Chiffrate C_1 und C_2. Anschließend berechnen wir $C_1' = C_1 \oplus \triangledown$ und $C_2' = C_2 \oplus \triangledown$ und entschlüsseln zu P_1' und P_2'. Der Kerngedanke des Angriffs besteht in der Feststellung, dass, wenn das Paar (P_1, P_2) dem Differentialpfad $\triangle$ und die Entschlüsselungen dem Differentialpfad $\triangledown$ folgen, die Zwischenwerte P_1' und P_2' die Differenz $\triangle'$ haben. Dadurch lässt sich mit Wahrscheinlichkeit $p_1^2 p_2^2$ zwischen einer Blockchiffre und einer zufälligen Permutation unterscheiden [Wag99].

Die Anwendung des Angriffs auf Hashfunktionen wurde von Joux und Peyrin vorgestellt [JP07] und verläuft etwas anders. Der wesentliche Unterschied liegt darin, dass bei einer Hashfunktion nicht entschlüsselt werden kann. Wenn entschlüsselt werden könnte, wäre eine Hashfunktion keine Einwegfunktion. Mit Hilfe des Angriffs wird die Komplexität zum Finden

von Kollisionen gesenkt. Die Grundlage bildet ein Hauptdifferentialpfad mit Eingabedifferenz $\triangle$, sodass die Eingaben M und $M \oplus \triangle$ mit Wahrscheinlichkeit $p_\triangle$ eine Kollision darstellen. Dabei werden die sogenannten frühen Schritte, bei denen Teile der Eingabe M unabhängig voneinander gewählt werden können, nicht berücksichtigt. Von da an werden die restlichen Schritte in die mittleren Schritte, mit der Wahrscheinlichkeit p_M und die späten Schritte, mit der Wahrscheinlichkeit p_S, aufgeteilt, sodass sich $p_\triangle = p_M * p_S$ ergibt. Ziel des Angriffs ist es nun, p_M zu verbessern. Dafür werden Hilfsdifferentialpfade verwendet, die die frühen und mittleren Schritte abdecken. Nehmen wir einen solchen Hilfsdifferentialpfad an, der besagt, dass mit Wahrscheinlichkeit p_δ zwei Eingaben M und $M \oplus \delta$ nach den mittleren Schritten zwei Zwischenwerte mit einer vorgegebenen Differenz ergeben. Nehmen wir nun zwei Eingaben M und $M' = M \oplus \triangle$ an, die dem Hauptdifferentialpfad auf den frühen und mittleren Schritten folgen, also mit Wahrscheinlichkeit $p_\triangle$ eine Kollision darstellen. Zusätzlich nehmen wir an, dass sowohl $(M, M \oplus \delta)$ als auch $(M', M' \oplus \delta)$ mit dem Hilfsdifferentialpfad konform sind, also nach den mittleren Schritten mit Wahrscheinlichkeit p_δ eine vorgegebene Differenz ergeben. Für das Paar $(M \oplus \delta, M' \oplus \delta)$, das $(M \oplus \delta, M \oplus \triangle \oplus \delta)$ entspricht, heben sich dann die durch δ entstandenen Zustandsdifferenzen auf und es stimmt auch mit dem Hauptdifferentialpfad bis zum Beginn der späten Schritte überein. Unter der Annahme der Unabhängigkeit der Differentialpfade ist dieses Paar mit der Wahrscheinlichkeit $p_\delta^2 * p_S$ eine Kollision. Ein Problem besteht darin, dass, wenn kein Paar (M, M') vorgegeben wird, nicht weiter konstruiert werden kann. Allerdings können wegen der geringen Anzahl an Schritten, viele Hilfsdifferentialpfade gefunden und so der Basisblock immer wieder benutzt werden.

2.3.9 Angriffe auf zugrundeliegende Chiffren

Neben Angriffen auf die Konstruktion einer Hashfunktion kann auch ein Angriff auf die zugrundeliegende Kompressionsfunktion Folgen für die Sicherheit der Hashfunktion haben. Viele der vorangegangen vorgestellten Angriffe befassen sich nicht mit dem Finden von Kollisionen, sondern nehmen zum Finden einer Kollision die Komplexität eines Geburtstagsangriffs $2^{n/2}$ an. Soll nun aber eine Hashfunktion angegriffen werden, für die $2^{n/2}$ praktisch unmöglich zu berechnen ist, ist es nötig, Angriffe, die eine Kollision mit weniger Operationen ermitteln, zu finden. Solche Angriffe bedienen sich in den meisten Fällen einer Kryptoanalyse der verwendeten Kompressionsfunktion, um Abhängigkeiten zwischen Eingabe und Ausgabe zu ermitteln und so die Wahrscheinlichkeit zum Finden einer Kollision zu erhöhen. Ein praktisches Beispiel dafür liefert der *Shattered* Angriff auf die SHA-1 Hashfunktion [SBK+17]. Wenn die Komplexität zum Finden einer Kollision von solch einem Angriff unter $2^{n/2}$ gesenkt werden kann, werden damit einhergehend auch alle Angriffe, die bisher $2^{n/2}$ Operationen zum Finden einer Kollision angenommen haben, effizienter.

3 Selbst implementierte Hashfunktionen

In diesem Kapitel werde ich selbst implementierte Hashfunktionen, die der Merkle-Damgård-Konstruktion folgen, vorstellen. Die verwendeten Kompressionsfunktionen basieren dafür auf den bereits gebrochenen Kryptosystemen der Vigenere Chiffre und der Blockchiffre FEAL. Anschließend werde ich Möglichkeiten, auf welche Weise die implementierten Hashfunktionen angreifbar sind, vorstellen und auch praktische Beispiele für gefundene Kollisionen präsentieren.

Sämtliche Zeitangaben sind dabei auf mein System bezogen, welches über einen 3,5 GHz AMD Ryzen 5 1500X Quad-Core Prozessor und 16 GB DDR4 RAM verfügt.

3.1 Vigenere-Chiffre

Die Vigenere-Verschlüsselung ist nach der Cäsar-Verschlüsselung die wohl best bekannte und gleichzeitig auch älteste Verschlüsselungsmethode. Sie baut auf einem ähnlichen Prinzip auf, allerdings werden nicht alle Buchstaben um denselben Wert verschoben, sondern es wird ein Schlüsselwort verwendet, das die Verschiebung der einzelnen Buchstaben codiert. Eine Verschlüsselung des Buchstaben „C“ mit dem dem Schlüssel „B“ entspricht einer Verschiebung von „C“ um 1, da „B“ an 1-ter Stelle des Alphabets steht (A steht an 0-ter Stelle).

Beispiel 3.1. Vigenere-Verschlüsselung von „lorem ipsum“ mit dem Schlüssel „Hallo“

Klartext	L	O	R	E	M	I	P	S	U	M
Schlüssel	H	A	L	L	O	H	A	L	L	O
Chiffrat	S	O	C	P	A	P	P	D	F	A

Der gewählte Schlüssel kann wesentlich kleiner sein als der Klartext und wird in so einem Fall entsprechend oft wiederholt, um die gewünschte Länge abzudecken.

Das Verfahren als Kompressionsfunktion zu verwenden, bedeutet, dass der letzte interne Hashwert, jeweils mit dem nächsten Block der Eingabe als Schlüssel, verschlüsselt wird. Dafür

ist es zunächst sinnvoll, die Schlüssel- und damit Blockgröße entsprechend der gewünschten Größe des Hashwertes zu wählen, da so die Kompressionsfunktion die größte mögliche Kompression erreicht.

Für die Verwendung als Hashfunktion Erweitern wir zusätzlich das Alphabet, sodass nicht nur Groß- und Kleinbuchstaben, sondern auch Zahlen und Satzzeichen verwendet werden können. Eine Verkleinerung des Alphabets auf 0 und 1, und damit verbunden die Unterstützung sämtlicher Eingaben als Bits, wäre prinzipiell auch möglich. Allerdings ist die Vigenere-Verschlüsselung auf diesem Alphabet äquivalent zur XOR-Operation, welche nicht einmal als gebrochenes Kryptosystem angesehen werden kann.

Natürlich ist das One-Time-Pad, welches aus der XOR-Operation besteht sogar beweisbar sicher. Allerdings setzt es voraus, dass der Schlüssel zufällig gewählt und nicht mehrfach benutzt wird, was bei einer Verwendung als Hashfunktion und der damit verbundenen Gewinnung der Schlüssel aus der Eingabe definitiv nicht der Fall wäre.

Das Padding der Eingaben erfolgt durch anhängen von A's gefolgt von der ursprünglichen Länge der Eingabe. Ein Initialisierungsvektor kann beliebig gewählt werden. Die Ausgabelänge ist nicht durch das Kryptosystem vorgegeben, es kann also auch hier jeder gewünschte Wert verwendet werden. Bei einer Ausgabelänge (und damit Blockgröße) von beispielsweise 16 sind also 95^{16} unterschiedliche Hashwerte möglich, da das verwendete Alphabet 95 Zeichen hat.

Bei Verwendung der Vigenere-Hashfunktion (VHash) fallen direkt erste strukturelle Schwächen auf. So führt eine kleine Veränderung der Eingabe nur zu einer kleinen Veränderung des Hashwertes. Genauer betrachtetet fällt auf, dass bei einer Blockgröße von k die i-te Stelle des Hashwertes nur von allen $(i + x \cdot k)$-ten Stellen der Eingabe abhängt. Der Avalanche Effekt [WT86] welcher besagt, dass die Veränderung eines einzelnen Eingabebits bei jedem einzelnen Ausgabebit mit einer Wahrscheinlichkeit von 50% zu einer Veränderung des Wertes führt, ist also nicht erfüllt.

Beispiel 3.2. Bei einer Blockgröße von 8 wird die 3-te Stelle des Hashwertes nur durch die 3-te, 11-te, 17-te, ... Stelle der Eingabe beeinflusst. Eine Veränderung der Eingabe an jeder anderen Stelle hat also keine Auswirkung auf die 3-te Stelle des Hashwertes.

Es ist auch möglich, eine 1-Block Eingabe zu berechnen, die zu einem gegebenen Hashwert führt, indem vom Hashwert direkt mit dem Initialisierungsvektor entschlüsselt wird. Hat diese Eingabe kein korrektes Padding, so muss die ursprüngliche Eingabe, die zum gegebenen Hashwert führt, länger gewesen sein. Hat sie ein korrektes Padding, so handelt es sich entweder um die ursprüngliche Eingabe oder bereits um eine Kollision zur ursprünglichen Eingabe.

3.1.1 Geburtstagsangriff

Ein Geburtstagsangriff auf VHash deckt weitere Mängel der Hashfunktion auf. So fällt auf, dass ein iteratives Ausprobieren aller Eingaben ($a \parallel a \parallel ... \parallel a$ gefolgt von $a \parallel a \parallel ... \parallel a \parallel b$) sehr viel ineffizienter zum Finden einer Kollision ist, als es ein Geburtstagsangriff sein müsste. Das liegt daran, dass die Eingaben den Initialisierungsvektor um einen festen Wert verschieben. VHash produziert also keine zufällig verteilten Hashwerte. Durch die Auswahl zufällig verteilter Eingaben lässt sich die Komplexität auf das Niveau eines Geburtstagsangriffs senken. Das Finden einer Kollision für Ausgabelänge $n = 6$ benötigt dementsprechend 95^3 getestete Eingaben und ist in ca. 20 Sekunden erledigt. Für $n = 8$ werden 95^4 Eingaben und auf meinem System knapp eine Stunde benötigt. Der Zeitaufwand steigt um mehr als das 95 fache, da mit größerer Zahl an Eingabe/Hashwert Kombinationen, die gespeichert werden müssen, die Performance leicht abnimmt.

Einen Brute-Force Angriff auf die anfangs vorgeschlagene Ausgabelänge von 16 durchzuführen ist mit diesem naiven Ansatz also bereits sehr aufwendig. Die in Beispiel 3.2 festgestellten Zusammenhänge lassen sich aber nutzen, um die Komplexität zu verringern. Bei $n = 16$ können zwei Eingaben der Länge 17 sich nur an erster und letzter Stelle unterscheiden, um kollidierende Hashwerte besitzen zu können. Jede andere Stelle der Hashwerte hängt nur von jeweils einer Stelle der Eingabe und einer Stelle des identischen Paddings beider Eingaben ab, diese Stellen der beiden Eingaben müssen also identisch sein, um denselben Hashwert zu ergeben.

Setzt man nun die, vorher vollständig zufälligen, Eingaben an allen Stellen außer der ersten und letzten auf einen festen Wert, wird die Anzahl der noch erreichbaren Hashwerte auf 95^1 limitiert. Es stehen aber noch 95^2 unterschiedliche Eingaben zur Verfügung. Auf diese Weise lassen sich Kollisionen in vernachlässigbarer Zeit finden und die Komplexität zum Finden einer Kollision steigt nicht mehr mit größerem n, sondern bleibt für alle Ausgabelängen konstant. Zusätzlich erhält man mit diesem Verfahren keine zufälligen Kollisionen, sondern kann, ähnlich wie bei einem Chosen-Prefix-Angriff, große Teile der Eingaben beliebig wählen.

Beispiel 3.3. In Anlehnung an Beispiel 2.2 kann für $n = 10$ nach zwei Eingaben gesucht werden, die sich nur an den Stellen X und Y unterscheiden und trotzdem kollidieren.

„Arbeitszeit Montag: X Stunden Y0 Minuten“

50 Kollisionen zu „Arbeitszeit Montag: 8 Stunden 10 Minuten“ lassen sich auf diese Weise in weit unter einer Sekunde finden. Dafür werden nur ca. $95^{1,85}$ Versuche benötigt anstatt der für schwache Kollisionsresistenz vorausgesetzten 95^{10} für nur eine einzige Kollision.

3.1.2 Konstruktion von Kollisionen

Die bisher verwendeten Ansätze können weiter verfolgt werden, um beliebige Kollisionen direkt zu konstruieren. Jedes Zeichen im verwendeten Alphabet bekommt den Wert seiner

Position im Alphabet zugeordnet, $W(E[i])$ bezeichnet den Wert der i-ten Stelle der Eingabe. Der Wert der i-ten Stelle des Hashwertes entspricht bei k Eingabeblöcken und einer Blockgröße von b der Summe aller $(i + xb)$-ten Werte und der entsprechenden Stelle des Initialisierungsvektors

$$W(H[i]) = \left(IV[i] + \sum_{x=0}^{x=k-1} W(E[i + xb]) \right) mod\ 95$$

Für eine Kollision muss also nur die Summe, der entsprechenden Zeichen einer Eingabe, mit der Summe der entsprechenden Zeichen einer anderen Eingabe, modulo 95, übereinstimmen.

Eine Art Kollisionen zu erzeugen, besteht darin, Zeichen, die dieselbe Stelle des Hashwertes beeinflussen, beliebig zu vertauschen. Innerhalb von Zahlen lassen sich so bereits sehr große inhaltliche Veränderungen hervorrufen. Soll ein Zeichen der Eingabe beliebig verändert werden, so muss die Verschiebung innerhalb des Alphabets nur durch eine Verschiebung in Gegenrichtung, die auf beliebig viele andere Zeichen, die dieselbe Stelle des Hashwertes beeinflussen, aufgeteilt werden kann, ausgeglichen werden. Eine Verschiebung in dieselbe Richtung, die modulo 95 denselben Wert ergibt, ist auch möglich.

Aus diesem Grund ist eine Veränderung der VHash Funktion, sodass die Blockgröße der Kompressionsfunktion kleiner ist als die Ausgabelänge der Hashfunktion, nicht förderlich. Jedes Zeichen des Hashwertes würde anschließend von mehr Zeichen der Eingabe abhängen, was einen Ansatz wie in Beispiel 3.3 erschweren würde. Allerdings würde es auch bedeuten, dass noch mehr Zeichen zum Ausgleich einer gewünschten Veränderung in Frage kommen. Die Veränderung würde die Hashfunktion also durch mehr Aufrufe der Kompressionsfunktion ineffizienter machen, ohne dabei die Sicherheit zu erhöhen.

Beispiel 3.4. Konstruktion einer Kollision für ein X.509-Zertifikat
Nehmen wir an, ein Angreifer Bob verfügt über ein gültiges X.509-Zertifikat [BSP+08] und das Subject-Feld, dieses Zertifikates, enthält den Eintrag „CN=Bob12.com/Email=ABCDE@example.com“. Bob möchte nun den Inhalt des Zertifikats so verändern, dass es für „CN=Alice.com/...“ gilt. Nach der Veränderung soll der mit 16 Block großem VHash berechnete Hashwert beider Zertifikate identisch sein. Wenn wir vereinfachend davon ausgehen, dass mit dem CN Eintrag ein neuer Block beginnt, so beeinflussen die Stellen des Eintrags den resultierenden Hashwert wie in Tabelle 3.1 angegeben.

Hashstelle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Eingabestellen	C	N	=	B	o	b	1	2	.	c	o	m	/	E	m	a
	i	l	=	A	B	C	D	E	@	e	x	a	m	p	l	e
	.	c	o	m												

Tabelle 3.1: Zusammenhang zwischen Eingabe und Hashwert

altes Zeichen	B	o	b	1	2
neues Zeichen	A	l	i	c	e
Differenz	1	3	-7	83	82

Tabelle 3.2: Zeichendifferenz

Es ist leicht zu erkennen, dass eine Veränderung von „Bob12.com“ zu „Alice.com“ innerhalb der ersten Stelle der Email-Adresse ausgeglichen werden kann. Die auszugleichenden Differenzen sind in Tabelle 3.2 angegeben.

Verrechnet Bob nun dieses Differenzen mit den ersten Stellen seiner Adresse „ABCDE@example.com“ so erhält er die Adresse „BEvrr@example.com“, unter deren Angabe er trotz des Austauschs der Domain, für das resultierende Zertifikat denselben Hashwert wie für sein ursprüngliches Zertifikat erhält. Mit dem für „Alice.com“ gültigen Zertifikat und der für dieses Zertifikat gültigen Signatur seines ursprünglichen Zertifikates kann Bob sich nun als Alice ausgeben. Ein Angriff dieser Art konnte auf tatsächlich verwendete MD5 basierte Zertifikate durchgeführt werden [SLdW07].

3.2 FEAL

FEAL (Fast data Encipherment ALgorithm) [SM88] ist eine Blockchiffre, die erstmals im Jahr 1987 vom japanischen Telefonkonzern NTT veröffentlicht wurde. Ziel war es, eine Alternative zum DES zu bieten, die effizienter in Software zu implementieren ist. Sämtliche Versionen von FEAL bauen auf der Struktur eines Feistelnetzwerks auf und haben eine Blockgröße von 64 Bit.

In einem Feistelnetzwerk [MOVR01] wird der Klartext zuerst in zwei (meist) gleich große Teile aufgeteilt $P = L_0 \parallel R_0$. Anschließend werden diese Teile in beliebig vielen Runden verarbeitet. Ein Teil ist dabei die Eingabe für eine Rundenfunktion F und der andere Teil wird mit der Ausgabe der Rundenfunktion F verknüpft. Nach einer Runde werden beide Teile neu zugewiesen, sodass $L_{i+1} = R_i$ und $R_{i+1} = L_i \oplus F(R_i, K_i)$ ist. K_0 bis K_n sind dabei aus dem Schlüssel K abgeleitete Rundenschlüssel. Im Fall von FEAL werden sie durch eine Verschlüsselung des ursprünglichen Schlüssels mit sich selbst, in einer leicht abgewandelten Version von FEAL, generiert und sind 16 Bit lang. Abbildung 3.1 zeigt eine generelle Darstellung eines Feistelnetzwerks.

Die Rundenfunktion F von FEAL (Abbildung 3.2) setzt sich aus 4 XOR-Verknüpfungen und zwei unterschiedlichen Arten von Substitutionsboxen S_0 und S_1 zusammen, von denen jede zweimal vorkommt. Beide S-Boxen sind definiert als $S_i(x, y) = Rotl2((x+y+i) \bmod 256)$. Die Funktion $Rotl2(x)$ entspricht dabei einer 2 Bit Links-Rotation von x.

Es ist zu erkennen, dass die modulare Byte Addition innerhalb der S-Boxen die einzige nicht lineare Operation der Funktion ist. Sie ist also für die Sicherheit von FEAL z. B. gegen

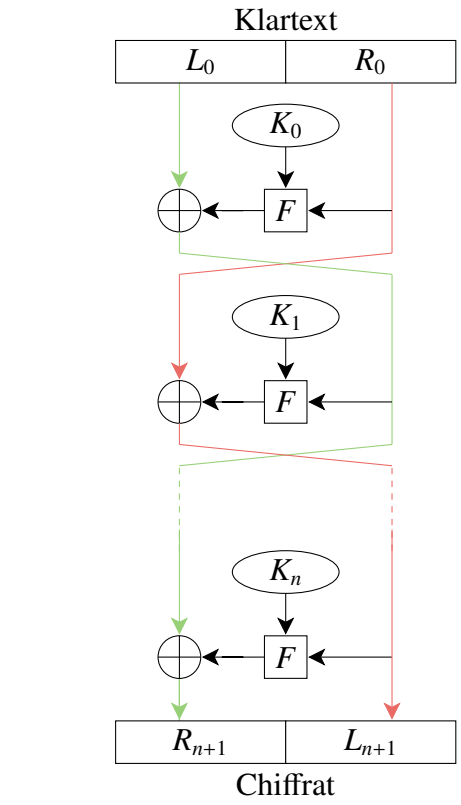

Abbildung 3.1: Allgemeiner Aufbau eines Feistelnetzwerks

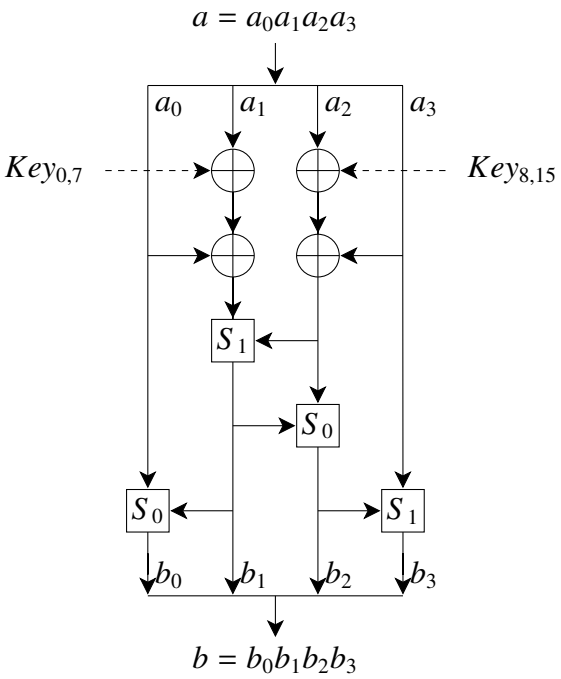

Abbildung 3.2: Rundenfunktion F von FEAL

eine differenzielle Kryptoanalyse von großer Wichtigkeit.

Vor der ersten und nach der letzten Runde der Verschlüsselung werden in FEAL zusätzlich die linke und rechte Hälfte der Daten miteinander und zusätzlich mit jeweils 4 weiteren Rundenschlüsseln XOR-Verknüpft. Dieses Verfahren ist eine Form des Key-Whitenings und wird verwendet, um die genauen Ein- und Ausgaben für die Funktionen der ersten und letzten Runde zu verbergen. Auf diese Weise wird ein Meet-in-the-Middle-Attack erschwert. Verfahren dieser Art werden auch von modernen Chiffren wie z. B. dem AES eingesetzt.

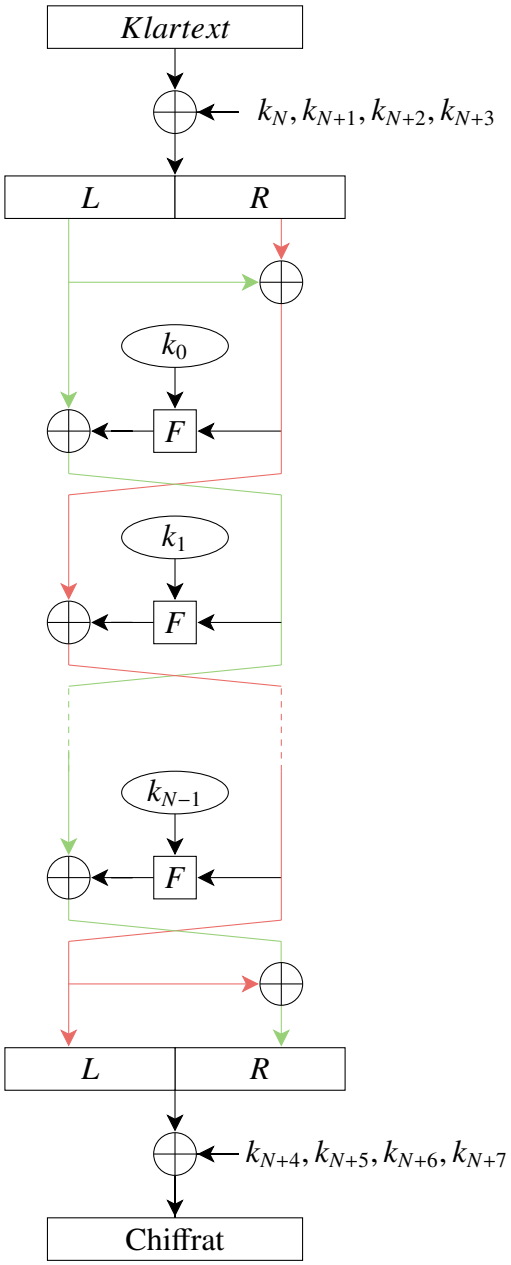

Abbildung 3.3: Key-Whitening und Verwendung der Rundenschlüssel k_0 bis k_{N+7} für N Runden FEAL

FEAL war anfangs auf 4 (FEAL4) und anschließend auf 8 (FEAL8) Runden beschränkt, um

möglichst effizient zu sein. Beide Varianten konnten aber schnell gebrochen werden [dB88] [GC91]. Die letzte Variante FEAL-NX ließ eine variable (gerade) Anzahl von N Runden zu und erweiterte die Schlüsselgröße von 64 auf 128 Bit. Von Biham und Shamir wurde allerdings nicht nur FEAL-N für $N \leq 32$ gebrochen, sondern auch gezeigt, dass FEAL-NX, mit einem 128 Bit langem Schlüssel, nicht sicherer ist als FEAL-N, mit einem 64 Bit langem Schlüssel [BS91b].

Auf FEAL aufbauend wurde die Hashfunktion N-Hash entwickelt [MOI90]. Sie hat eine Blockgröße von 128 Bit und eine leicht andere Rundenfunktion als sie in FEAL verwendet wird. Die Kompressionsfunktion von N-Hash setzt nur 8 Runden ein und N-Hash wurde ebenfalls von Biham und Shamir gebrochen [BS91b].

In meiner Implementation der auf FEAL basierenden Hashfunktion (FEALHash) habe ich die exakte Rundenfunktion von FEAL mit einem 64 Bit langem Schlüssel verwendet. Die Rundenzahl ist variabel, wenn nicht anders angegeben habe ich aber 16 Runden verwendet. Da FEAL eine Blockgröße von 64 Bit hat entstehen 64 Bit große Hashwerte, welche ich im Folgenden als 16 Stellige Hexadezimalwerte angebe. Als Initialisierungsvektor habe ich angelehnt an SHA-2 die ersten 8 Bits der Nachkommastellen der Quadratwurzeln der ersten 8 Primzahlen 2-19 verwendet. Solche Werte werden für Konstanten verwendet, um zu zeigen, dass es sich bei den zufälligen Werten tatsächlich nur um zufällige Werte handelt und nicht um eine versteckte Schwachstelle. Selbst wenn keine versteckte Schwachstelle nachgewiesen werden kann, können unbegründete Zufallswerte ein Grund zur Besorgnis sein [Sch07].

Die Kompressionsfunktion von FEALHash verwendet den Eingabeblock als Klartext und den letzten internen Hashwert als Schlüssel für die Verschlüsselung. Der nächste interne Hashwert ist nicht die Ausgabe der FEAL Verschlüsselung (das Chiffrat), sondern die XOR-Verknüpfung aus Eingabe, Chiffrat und verwendetem Schlüssel. Dieses Verfahren ist eines der 4 sicheren Verfahren eine Hashfunktion iterativ aufzubauen [PGV94].

Beispiel 3.5. FEALHash Beispielwert
Eingabe: „The quick brown fox jumps over the lazy dog“
Hashwert: „20CF4D8B E1C1E08A“

Bei Verwendung von FEALHash lässt sich erkennen, dass der Avalanche Effekt erfüllt ist. Wird ein einzelnes Bit der Eingabe verändert so verändert sich der resultierende Hashwert fast vollständig.

Beispiel 3.6. Avalanche Effekt
Eingabe: „The quick brown fox jumps over the lazy dof“
Hashwert: „1A0FABF6 BBFBD34E“

3.2.1 Geburtstagsangriff

FEALHash produziert 64 Bit lange Hashwerte ein Geburtstagsangriff benötigt also 2^{32} unterschiedliche Eingaben um mit einer Wahrscheinlichkeit von > 50% eine Kollision zu finden. Dieser Wert scheint gegen die $2^{61,2}$ Operationen für das Finden einer Kollision in SHA-1 verschwindend gering zu sein [LP20]. Allerdings muss man beachten, dass für diesen Angriff 900 Nvidia GTX 1060 GPUs insgesamt zwei Monate benötigt haben.

Auf meinem System werden, bei einer Implementierung in Python, zur Berechnung von 2^{20} FEALHash Werten knapp 100 Sekunden benötigt. Eine Reduzierung der Rundenzahl auf 8 oder 4 reduziert diesen Wert auf 60 oder 40 Sekunden. Bei einer Rundenzahl von 16 werden dementsprechend knapp 5 Tage benötigt, um mit einer Wahrscheinlichkeit von > 50% eine Kollision zu finden. Für eine Wahrscheinlichkeit von > 75% sind es knapp 8 Tage. Ohne Zweifel sind diese Werte zu gering, um die Hashfunktion ansatzweise als sicher zu bezeichnen. Trotzdem sind sie noch wesentlich zu hoch, um effizient Angriffe durchzuführen für die viele Kollisionen gefunden werden müssen oder auch nur verlässlich eine einzelne Kollision zu berechnen. Durch Verwendung des Just-in-time Compilers PyPy lässt sich die Geschwindigkeit für 1 Millionen Hashes auf 33 Sekunden reduzieren, sie ist aber immer noch bedeutend langsamer als eine C Implementierung.

Eine Implementierung in C benötigt nur 1,8 Sekunden, um 1 Millionen Hashwerte zu berechnen. Die Eingaben sind dabei auf zufällig generierte 64 Bit Blöcke ohne Padding beschränkt, die jeweils nur ein einziges mal mit dem Initialisierungsvektor verschlüsselt werden. Auf diese Weise lassen sich effizienter Kollisionen finden, die trotzdem beliebig erweiterbar sind, da sie nach nur einer Iteration der MD-Konstruktion identische interne Hashwerte ergeben. Werden also die gefundenen kollidierenden Eingaben um einen identischen Suffix erweitert, so ergeben die resultierenden Eingaben ebenfalls kollidierende Hashwerte. Nach Finden einer Kollision lassen sich also weitere Kollisionen aus den kollidierenden Präfixen konstruieren.

Durch die große Zeitersparnis einer C Implementierung ist es theoretisch möglich, eine Kollision mit > 50% Wahrscheinlichkeit in nur 2 Stunden zu ermitteln. Allerdings treten bei der Durchführung Speicherprobleme auf. Selbst wenn für jeden ermittelten Hashwert nur 8 Bytes Eingabe und 8 Bytes Hashwert gespeichert werden, würden für 2^{32} Werte insgesamt $\frac{2^4 \cdot 2^{32}}{2^{30}}$ = 64 GB Speicher benötigt werden und das ohne Berücksichtigung eines Overheads durch die verwendeten Datenstrukturen. Solche Datenmengen sind einerseits schwer zu verwalten, da nicht im Vorhinein feststeht, wie viele Werte tatsächlich berechnet und gespeichert werden müssen, bis eine Kollision gefunden wird. Andererseits wird das Programm mit zunehmender Größe der Daten langsamer, da 64 GB nicht nur die 16 MB L3-Cache meines Prozessors, sondern auch meine 16 GB Arbeitsspeicher weit übersteigen. Die Daten müssen also auf der Festplatte gespeichert werden, was die Lese- und Schreibzugriffe massiv verlangsamt. Werden z. B. 1 Millionen Hashwerte berechnet und samt Eingabe in

einer .txt Datei gespeichert, so benötigt das Programm hierfür ca. 1 Sekunde länger als für das Berechnen und Speichern in einem Array (insgesamt 2,8 Sekunden).

Ein weiteres Problem ist die Suche nach Duplikaten in den berechneten Hashwerten, ein einfaches Vergleichen mit sämtlichen Werten läuft in $O(n^2)$ und ist daher inakzeptabel. Die Werte in einer Hashtable zu speichern erlaubt das Finden von Duplikaten in $O(1)$. Allerdings sind Hashtables nur effizient, solange keine Kollisionen in der von ihnen verwendeten Hashfunktion auftreten [CW03]. Die berechneten 64 Bit FEALHashwerte können aber nicht ohne weitere Komprimierung als Schlüssel für die (Key, Value)-Paare der Hashtable verwendet werden, da sonst der Schlüsselraum zu groß wird. Sie mit Hilfe der Hashfunktion der Hashtable weiter zu reduzieren, ist aber auch nicht möglich, da selbst wenn diese Hashfunktion perfekt ist, Kollisionen in ihr gefunden werden müssen, bevor eine Kollision in FEALHash gefunden wird. Werden die 64 Bit Werte beispielsweise auf 32 Bit reduziert, so ist nach 2^{16} Eingaben eine Kollision, in der Hashfunktion der Hashtable, zu erwarten, bis also die 2^{32} Eingaben für eine Kollision in FEALHash gespeichert sind, wären 2^{16} Kollisionen zu erwarten und das schon ohne in Betracht zu ziehen, dass Kollisionen immer wahrscheinlicher werden, je mehr Werte vorliegen. Eine Hashtable würde also schnell ineffizient werden.

Aufgrund dieser Schwierigkeiten und durch eine weitaus bessere Alternative habe ich das Finden einer Kollision in FEALHash durch einen reinen Brute-Force Ansatz nicht weiter verfolgt.

3.2.2 Differenzielle Kryptoanalyse

Die differenzielle Kryptoanalyse [BS91a] ist ein Verfahren, welches die Auswirkung einer bestimmten Differenz zwischen zwei verschiedenen Klartexten auf die Differenz zwischen den zwei korrespondierenden Chiffraten analysiert. Mit Differenz zwischen zwei Werten ist in diesem Zusammenhang die XOR-Verknüpfung beider Werte gemeint. Durch dieses Verfahren lassen sich bei Anwendung auf eine Blockchiffre die verwendeten Rundenschlüssel ermitteln. Bei Anwendung auf die Kompressionsfunktion einer Hashfunktion kann es genutzt werden, um eine Kollision schneller als über einen Geburtstagsangriff zu finden. Ein Paar bezeichnet im Folgenden zwei unterschiedliche Werte, die eine bestimmte Differenz zueinander haben. Die Differenz eines Paares ist die Differenz beider Werte zueinander.

Die meisten Operationen innerhalb von FEAL, wie Rotationen oder XOR-Verknüpfungen mit bestimmten Werten verändern die Differenz ihrer Eingaben nicht, die entsprechenden Ausgaben haben also dieselbe Differenz wie die Eingaben. Solche Operationen sind linear, die in FEAL verwendeten S-Boxen sind aber nicht linear. Die Kenntnis über die Eingabedifferenz eines Paares gibt also nicht zwingend Kenntnis über die Ausgabedifferenz eines Paares, die vorliegt, nachdem beide Werte die S-Box durchlaufen haben. Jede mögliche Eingabedifferenz einer S-Box führt aber mit einer bestimmten Wahrscheinlichkeit zu jeder möglichen Ausgabedifferenz. Durch eine Analyse der Wahrscheinlichkeiten lassen sich nun

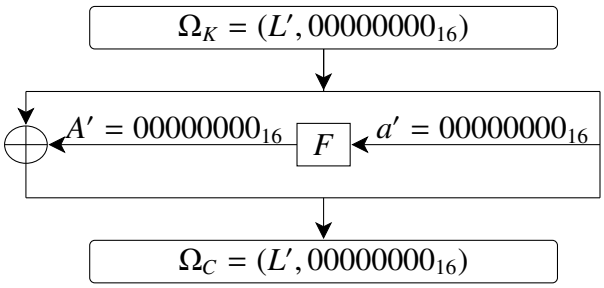

Abbildung 3.4: 1-Runden Charakteristik mit Wahrscheinlichkeit $\frac{1}{1}$

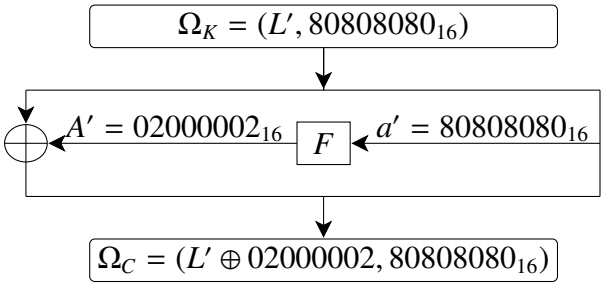

Abbildung 3.5: 1-Runden Charakteristik mit Wahrscheinlichkeit $\frac{1}{1}$

jene Eingabedifferenzen auswählen, die garantiert oder mit hoher Wahrscheinlichkeit zu einer bestimmten Ausgabedifferenz führen. Aus ihnen können anschließend Charakteristiken für das Verhalten von unterschiedlichen Differenzen innerhalb eines Paares über mehrere Runden der Verschlüsselungsfunktion abgeleitet werden.

Eine N Runden Charakteristik benennt eine Eingabedifferenz Ω_K und gibt eine Wahrscheinlichkeit dafür an, dass nach N Runden der Verschlüsselungsfunktion die Ausgabedifferenz Ω_C beträgt. Zwei Charakteristiken Ω^1 und Ω^2 können aneinander gehängt werden, wenn Ω^1_C identisch mit den vertauschten Hälften von Ω^2_K ist. Die Wahrscheinlichkeit der resultierenden Charakteristik Ω entspricht dann dem Produkt der Wahrscheinlichkeiten von Ω^1 und Ω^2. Eine Charakteristik wird iterativ genannt, wenn sie beliebig oft hintereinander angewandt werden kann, also Ω_C identisch mit den vertauschten Hälften von Ω_K ist. Ein Paar, dessen Eingabedifferenz Ω_K entspricht und das nach N Runden der Verschlüsselungsfunktion tatsächlich eine Differenz von Ω_C besitzt, wird als wahres Paar bezeichnet. Wahre Paare treten abhängig von der Wahrscheinlichkeit einer Charakteristik häufiger oder seltener auf.

Biham und Shamir haben einige Charakteristiken für die Rundenfunktion F von FEAL ermittelt [BS91b], darunter zwei 1-Runden Charakteristiken mit einer Wahrscheinlichkeit von 1 und eine 4-Runden Charakteristik mit einer Wahrscheinlichkeit von $\frac{1}{256}$.

Die Kleinbuchstaben beschreiben jeweils die Differenz vor der Rundenfunktion F. Die Großbuchstaben beschreiben die Differenz nach der Anwendung der Rundenfunktion. Aus den beiden 1-Runden Charakteristiken Abbildung 3.4 und Abbildung 3.5 lässt sich die 3-Runden

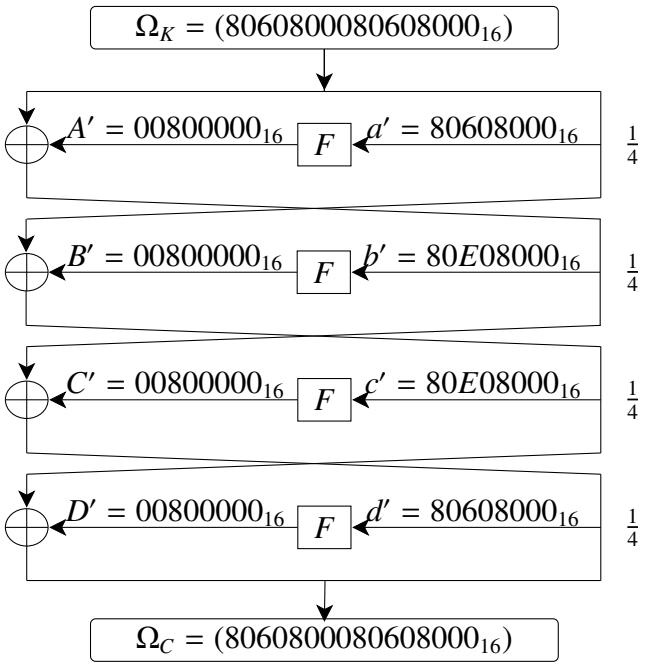

Abbildung 3.6: 4-Runden iterative Charakteristik mit Wahrscheinlichkeit $\frac{1}{256}$

Charakteristik Abbildung 3.7 konstruieren.

Es existiert noch eine 8-Runden Charakteristik mit Wahrscheinlichkeit $\frac{1}{128}$ und 3 leicht variablen Ausgabestellen. Sie ist für die Anwendung zum Finden einer Kollision allerdings schlechter geeignet als die iterative Charakteristik 3.6. Um eine Charakteristik auf eine N Runden Variante von FEAL anwenden zu können, muss sie entweder ebenfalls eine Rundenzahl von N haben oder es muss sich um einer iterative Charakteristik handeln, deren Rundenzahl die Zahl N restlos teilt. Die Verwendung von Charakteristiken, die um 1, 2 oder 3 Runden kürzer sind als die gesamte Verschlüsselung, ist zwar gegen Blockchiffren möglich, kann bei einem Angriff auf eine Hashfunktion allerdings nicht genutzt werden [RP95].

In unserem Fall bedeutet das, dass für einen Angriff gegen 16 Runden FEALHash nur die iterative 4-Runden Charakteristik 3.6 verwendet werden kann. Für die 16 Runden Variante dieser Charakteristik ergibt sich die Wahrscheinlichkeit $\frac{1}{2^{32}}$. Nachdem 2^{32} zufällige Paare ausprobiert wurden wäre also ein wahres Paar für diese Charakteristik gefunden.

Da die Kompressionsfunktion von FEALHash die XOR-Verknüpfung aus der Eingabe, dem berechneten Chiffrat und dem Schlüssel bildet, ist ein wahres Paar immer eine Kollision in der Hashfunktion. Das kommt daher, dass für wahre Paare $\Omega_K = \Omega_C$ ist, die Differenz zwischen Eingabe und berechnetem Chiffrat ist also für beide Elemente des Paares identisch. Der Schlüssel wird für beide Elemente des Paares aus demselben Initialisierungsvektor erzeugt, er ist also ebenfalls identisch.

Für diesen Angriff werden insgesamt 2^{32} Paare benötigt. Er ist aber bereits ohne weitere Optimierung eine Verbesserung gegenüber Brute-Force. Einerseits, da falsche Paare nicht

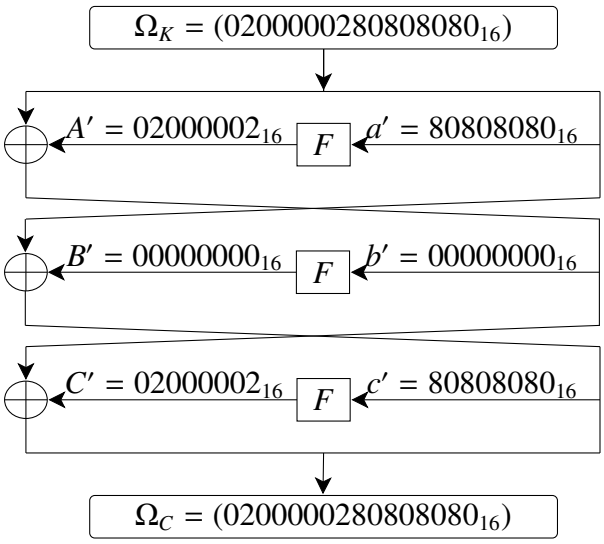

Abbildung 3.7: Konstruierte 3-Runden Charakteristik mit Wahrscheinlichkeit $\frac{1}{1}$

gespeichert werden müssen, was den Speicheraufwand massiv senkt und andererseits, da die Wahrscheinlichkeit für das Finden einer Kollision gegen 100% geht, was höher als die 50% eines Geburtstagsangriffs ist. Um den Angriff effizienter zu machen, kann nach jeder vierten Runde überprüft werden, ob die Charakteristik für das ausgewählte Paar noch gültig ist. Durch das frühzeitige Verwerfen von falschen Paaren kann die Geschwindigkeit um den Faktor $\frac{3N}{7}$ erhöht werden. Die Verwendung von bestimmten Schlüsseln (im Fall von FEALHash Initialisierungsvektoren), um die Wahrscheinlichkeit einer Charakteristik zu erhöhen, ist im Fall von FEAL nicht anwendbar.

Des Weiteren kann der Angriff optimiert werden, indem, anstatt komplett zufällige Paare zu verwenden, nur solche ausgewählt werden, für die die Charakteristik in der ersten Runde garantiert werden kann [PGV93]. In dieser Auswahl ist die Wahrscheinlichkeit für ein wahres Paar, über 16 Runden, um das 256-fache höher und somit werden nur insgesamt 2^{24} Paare benötigt, um ein wahres Paar und damit eine Kollision in FEALHash zu Finden. Die Charakteristik ist in der ersten Runde garantiert, wenn die Eingabedifferenzen $E0_{16}$ und 80_{16} innerhalb der zweiten S-Box zu einer Ausgabedifferenz von 80_{16} führen. Schreiben wir die Bytes des Klartextes als $P[0]$ bis $P[7]$ und die Rundenschlüssel als $K0$ bis $K24$, so können die Eingaben dieser S-Box wie folgt geschrieben werden:

$$S2a = P[0] \oplus P[1] \oplus P[4] \oplus P[5] \oplus K2a$$
$$S2b = P[2] \oplus P[3] \oplus P[6] \oplus P[7] \oplus K2b,$$

mit

$$K2a = K16[0] \oplus K16[1] \oplus K18[0] \oplus K18[1] \oplus K0[0]$$
$$K2b = K17[0] \oplus K17[1] \oplus K19[0] \oplus K19[1] \oplus K0[1].$$

Eine Möglichkeit zum Generieren eines wahren Paares, mit höherer Wahrscheinlichkeit, besteht darin, $S2a = S2b = A0_{16}$ zu wählen. Anschließend lassen sich beide Gleichungen zu einer Variablen auflösen und die Charakteristik wird für die erste Runde erfüllt sein. Da 6 Bytes des Klartextes zufällig gewählt werden können, gibt es 2^{48} unterschiedliche Klartexte, die diese Gleichung erfüllen.

Dieser Angriff benötigt auf meinem System in etwa fünf Minuten, um eine Kollision für 16 Runden FEALHash mit einem vorgegebenen IV zu Finden. Für die Generation der benötigten zufälligen Eingabewerte habe ich einen Mersenne-Twister-Generator von Takuji Nishimura und Makoto Matsumoto verwendet [NM04].

Beispiel 3.7. Kollision in 16 Runden FEALHash
Eingabe 1: „61CFE810 81D98E26“
Eingabe 2: „E1AF6810 81D98E26“
IV: „398B1E89 140C67BC“
Hashwert: „9A9ED6B4 61C5FFB4“

Für den anfangs für FEALHash angegebene IV wird im Verhältnis zu anderen IVs eine längere Zeit benötigt, bis eine Kollision gefunden ist (10 Minuten). Selbst ein IV, für den die Charakteristik nie erfüllt wäre, bietet aber trotzdem keinen Schutz gegen diesen Angriff, da einfach der letzte interne Hashwert als Schlüssel verwendet wird. Es können also ausgehend vom IV so lange beliebige Eingabeblöcke gehasht werden, bis nach einem Schritt ein interner Hashwert erreicht ist, der die Charakteristik erfüllt. Werden dann an diesen Eingabeblock kollidierende Blöcke angehängt, wäre eine Kollision auch für einen IV der die Charakteristik nicht erfüllt gefunden.

Beispiel 3.8. Kollision aus 2-Block Eingaben für originalen IV: 6ABB3CA5 519B1F5B
Eingabe 1: „01234567 89ABCDEF 15B07A3E 88A0F4EB“
Eingabe 2: „01234567 89ABCDEF 95D0FA3E 88A0F4EB“
Hashwert: „80B8D7A8 4C0ED789“

Es ist anzumerken, dass die Key-Whitening Prozesse von FEAL keinerlei erhöhte Schwierigkeit für diesen Angriff darstellen. Die XOR-Verknüpfung mit den Rundenschlüsseln geschieht für beide Elemente eines Paares mit denselben Werten, da die Rundenschlüssel für beide aus demselben Initialisierungsvektor gewonnen werden. Die Differenz eines Paares wird durch diese XOR-Verknüpfung also nicht verändert. Die initiale und abschließende

XOR-Verknüpfung der rechten und linken Hälfte lässt sich einfach umgehen, indem Paare mit einer Differenz von 8060800000000000_{16} gewählt werden. So liegt zu Beginn der Charakteristik immer die gewünschte Differenz von 8060800080608000_{16} vor und falls es sich um ein wahres Paar handelt, so ergibt sich am Ende wieder die Differenz von 8060800000000000_{16} und es wäre eine Kollision gefunden.

Der beschriebene Angriff funktioniert in dieser Form nur, da die Blöcke der Eingabe als Klartext und der IV als Schlüssel für FEAL verwendet wird. Wird diese Zuordnung vertauscht verliert der Angreifer die Kontrolle über die Differenz der Klartexte und der Angriff ist nicht mehr möglich. Allerdings entstehen dadurch andere Schwächen. Einerseits wäre es ohne großen Aufwand möglich, feste Punkte in der Rundenfunktion zu ermitteln und diese zum Finden einer Kollision anzuwenden [PGV94]. Andererseits wäre es auch denkbar, einen dualen differenziellen Angriff zu entwickeln, in dem die Differenz unterschiedlicher Schlüssel zu bestimmten Ausgaben von FEAL führt. Die Schlüsselgeneration von FEAL ist sehr ähnlich aufgebaut wie die Blockchiffre selbst. Es besteht also Grund zur Annahme, dass sich auch in ihr verwendbare Charakteristiken finden lassen.

Eine zuverlässige Möglichkeit, FEALHash gegen differenzielle Kryptoanalyse zu schützen, besteht darin, die Rundenzahl soweit zu erhöhen, dass die Wahrscheinlichkeit der Charakteristik einen höheren Aufwand verlangt als er für einen Geburtstagsangriff nötig wäre. Tabelle 3.3 zeigt, dass das ab 24 Runden der Fall ist. Ein Erhöhen der Rundenzahl hat aber auch großen Einfluss auf die Effizienz der Hashfunktion.

Rundenzahl	Komplexität	Zeitaufwand auf meinem System
4	2^6	-
8	2^{12}	< 0.02 Sekunden
12	2^{18}	< 2 Sekunden
16	2^{24}	< 10 Minuten
20	2^{30}	< 12 Stunden
24	2^{36}	-

Tabelle 3.3: Komplexität von differenzieller Kryptoanalyse gegen bestimmte Rundenzahlen von FEALHash

3.2.3 Multikollisionsangriff

Ein Multikollisionsangriff ist, mit dem durch die differenzielle Kryptoanalyse zur Verfügung stehendem Verfahren, zum Finden von Kollisionen, einfach auf FEALHash anwendbar. Dafür muss zuerst eine Kollision für den spezifizierten IV gefunden werden, die kollidierenden Eingaben bezeichnen wir als P_1 und P'_1. Anschließend kann der identische Hashwert beider Eingaben als neuer Initialisierungsvektor angegeben und erneut eine Kollision gefunden

werden. Bezeichnen wir die neuen kollidierenden Eingaben als P_2 und P'_2, so ist bereits die folgende 4-Kollision gefunden:

$$h(P_1 \parallel P_2) = h(P'_1 \parallel P_2) = h(P_1 \parallel P'_2) = h(P'_1 \parallel P'_2)$$

Für FEALHash beispielsweise mit:

$$\begin{aligned} IV &= 6ABB3CA5\ 519B1F5B \\ P_1 &= CFA521CE\ 025C4BCF \\ P'_1 &= 4FC5A1CE\ 025C4BCF \\ P_2 &= E40F0611\ 6484098B \\ P'_2 &= 646F8611\ 6484098B \\ Hashwert &= 4F257A66\ A6D6786B \end{aligned}$$

Dieser Angriff benötigt auf meinem System ca. 12 Minuten, um die angegebene 4-Kollision zu finden. Selbst wenn für eine einzelne Kollision 10 Minuten veranschlagt werden, so benötigt dieser Angriff generell nur $a \cdot 10$ Minuten um eine 2^a-Kollision zu finden.

4 Fazit und Ausblick

In dieser Arbeit wurden die Grundlagen von kryptographischen Hashfunktionen und der Merkle-Damgård-Konstruktion vorgestellt. Es wurde erläutert, in welcher Weise Kollisionen in Hashfunktionen gefunden werden können und welche weiteren Angriffsmöglichkeiten nach Finden einer Kollision zur Verfügung stehen.

Hashfunktionen, die auf der Vigenere Verschlüsslung und der Blockchiffre FEAL, zwei gebrochenen Kryptosystemen, aufbauen, wurden implementiert und ihre Sicherheit durch das Finden von Kollisionen aberkannt. Die Komplexität des jeweiligen Kryptosystems hat dabei Auswirkungen auf die Komplexität der benötigten Angriffe gezeigt. Sämtliche Angriffe konnten aber tatsächlich auf einem durchschnittlichen Desktop-Computer durchgeführt werden. Gegen das aufwendigere der beiden Kryptosysteme, die Blockchiffre FEAL, konnte eine differenzielle Kryptoanalyse zum Finden von Kollisionen genutzt werden. Die Effizienz dieses Verfahrens wurde durch unterschiedliche Maßnahmen, einerseits innerhalb der Programmierung und andererseits in der Auswahl der Eingabewerte, deutlich erhöht. Anschließend war die Komplexität zum Finden einer Kollision gering genug, um nicht nur eine Kollision in wenigen Minuten zu finden, sondern es war darüber hinaus auch möglich einen Multikollisionsangriff gegen die implementierte Hashfunktion durchzuführen.

Als allgemein lehrreich lässt sich die Erfahrung bewerten, bis zu welchem Grad selbst kleine Schwachstellen in einem Kryptosystem mit verhältnismäßig geringen Mitteln ausnutzbar sind. Die Arbeit hat gezeigt, dass gebrochene Chiffren keine Basis für eine sichere Hashfunktion bieten können, da sich ihre Schwachstellen auch bei einer Verwendung als Hashfunktion ausnutzen lassen. Die aufwendigen Standardisierungsprozesse in diesem Bereich sind vollends begründet. Eine sichere Hashfunktion zu schreiben ist eine schwere wie auch wichtige Aufgabe und nur die Wenigsten sollten sich ernsthaft selbst daran versuchen, wie sich immer wieder zeigt [CT18]. Der Grundsatz „Don't roll your own Crypto“ mag einem trivial erscheinen. Doch beim Versuch, dies zu tun, wird einem klar, welch schier unlösbare Aufgabe dahinter steckt. Die von einem breiten wissenschaftlichen Publikum über Jahre analysierten SHA-Funktionen sind ein Luxus, der jedem zur Verfügung steht. Sie sollten deshalb für eine tatsächliche Verwendung immer einer eigenen Implementierung vorgezogen werden.

Nicht näher eingegangen wurde in dieser Arbeit einerseits auf die unterschiedlichen Anwendungsformen der MD-Konstruktion [PGV94] und andererseits auf andere Verfahren zur Konstruktion einer Hashfunktion. Nachdem im SHA-3 Wettbewerb eine Funktion ausgewählt

wurde, die nicht nach dem Merkle-Damgård Schema konstruiert ist, stellt sich die Frage nach der generellen Zukunftsfähigkeit der MD-Konstruktion zum Beispiel im Vergleich zur Sponge-Konstruktion. Die MD-Konstruktion wurde bereits häufig angepasst und erweitert, um gegen neue Angriffsvarianten geschützt zu sein oder höhere Sicherheitsansprüche zu erfüllen [CDMP05]. Ein Vergleich unterschiedlicher Konstruktionen könnte Aufschluss darüber liefern, in welcher Weise diese Veränderungen qualitativ mit von Grund auf anderen Ansätzen zu vergleichen sind. Auch ein Blick auf die Sicherheit unterschiedlicher Ansätze gegen ausreichend entwickelte Quantencomputer könnte in diese Vergleiche mit einbezogen werden.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[BS91a] Eli Biham and Adi Shamir. Differential cryptanalysis of des-like cryptosystems. In Alfred J. Menezes and Scott A. Vanstone, editors, *Advances in Cryptology-CRYPTO' 90*, pages 2–21, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[BS91b] Eli Biham and Adi Shamir. Differential cryptanalysis of feal and n-hash. In Donald W. Davies, editor, *Advances in Cryptology — EUROCRYPT '91*, pages 1–16, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[BSP+08] Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and Dave Cooper. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, May 2008.

[Buc16] Johannes Buchmann. *Einführung in die Kryptographie*. Springer Spektrum, Berlin, Heidelberg, 6 edition, 2016.

[CDMP05] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-damgård revisited: How to construct a hash function. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, pages 430–448, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[CT18] Michael Colavita and Garrett Tanzer. A cryptanalysis of iota's curl hash function, 2018.

[CW03] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C., August 2003. USENIX Association.

[Dam90] Ivan Bjerre Damgård. A design principle for hash functions. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 416–427, New York, NY, 1990. Springer New York.

[Dav85] Donald W Davies. Digital signature-an update. In *Proc. International Conference on Computer Communications, Sydney*, pages 843–847. Elsevier, 1985.

[dB88] Bert den Boer. Cryptanalysis of f.e.a.l. In D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmüller, J. Stoer, N. Wirth,

and Christoph G. Günther, editors, *Advances in Cryptology — EUROCRYPT '88*, pages 293–299, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[GC91] Henri Gilbert and Guy Chassé. A statistical attack of the feal-8 cryptosystem. In Alfred J. Menezes and Scott A. Vanstone, editors, *Advances in Cryptology-CRYPTO' 90*, pages 22–33, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[Jou04] Antoine Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In Matt Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, pages 306–316, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[JP07] Antoine Joux and Thomas Peyrin. Hash functions and the (amplified) boomerang attack. In Alfred Menezes, editor, *Advances in Cryptology - CRYPTO 2007*, pages 244–263, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[KK06] John Kelsey and Tadayoshi Kohno. Herding hash functions and the nostradamus attack. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006*, pages 183–200, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[KS05] John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, pages 474–490, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[KW11] Ralf Küsters and Thomas Wilke. *Moderne Kryptographie*. Vieweg+Teubner, 2011.

[Len04] Arjen K. Lenstra. Key length. contribution to the handbook of information security, 2004.

[LP20] G. Leurent and Thomas Peyrin. Sha-1 is a shambles - first chosen-prefix collision on sha-1 and application to the pgp web of trust. In *IACR Cryptol. ePrint Arch.*, 2020.

[Luc04] Stefan Lucks. Design principles for iterated hash functions. Cryptology ePrint Archive, Report 2004/253, 2004. `https://eprint.iacr.org/2004/253`.

[Mer79] Ralph C. Merkle. *Secrecy, Authentication, and public key systems*. PhD thesis, Stanford University, 1979.

[Mer90] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 218–238, New York, NY, 1990. Springer New York.

[MOI90] Shoji Miyaguchi, Kazuo Ohta, and Masahiko Iwata. 128-bit hash function (n-hash). In *Proceedings of SECURICOM90*, pages 123–137, 1990.

[MOVR01] Alfred J. Menezes, Paul C. Van Oorschot, Scott A. Vanstone, and R. L. Rivest. *Handbook of Applied Cryptography*. CRC Press, 5 edition, 2001.

[NM04] Takuji Nishimura and Makoto Matsumoto. A c programm for mersenne twister. `http://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/emt.html`, 2004. Eingesehen: 03.08.2021.

[NP10] Mridul Nandi and Souradyuti Paul. Speeding up the wide-pipe: Secure and fast hashing. In Guang Gong and Kishan Chand Gupta, editors, *Progress in Cryptology - INDOCRYPT 2010*, pages 144–162, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[oEiCI12] ICT-2007-216676 ECRYPT II European Network of Excellence in Cryptology II. Yearly report on algorithms and keysizes, 2012.

[PGV93] Bart Preneel, Rene Govaerts, and Joos Vandewalle. Differential cryptanalysis of hash functions based on block ciphers. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, page 183–188, New York, NY, USA, 1993. Association for Computing Machinery.

[PGV94] Bart Preneel, René Govaerts, and Joos Vandewalle. Hash functions based on block ciphers: a synthetic approach. In Douglas R. Stinson, editor, *Advances in Cryptology — CRYPTO' 93*, pages 368–378, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[RP95] Vincent Rijmen and Bart Preneel. Improved characteristics for differential cryptanalysis of hash functions based on block ciphers. In Bart Preneel, editor, *Fast Software Encryption*, pages 242–248, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[SBK+17] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full sha-1. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 570–596, Cham, 2017. Springer International Publishing.

[Sch07] Bruce Schneier. Did nsa put a secret backdoor in new encryption standard? `https://www.wired.com/2007/11/securitymatters-1115/`, 2007. Eingesehen: 19.07.2021.

[Sch20] Jörg Schwenk. *Sicherheit und Kryptographie im Internet*. Springer Vieweg, Wiesbaden, 5 edition, 2020.

[SLdW07] Marc Stevens, Arjen Lenstra, and Benne de Weger. Chosen-prefix collisions for md5 and colliding x.509 certificates for different identities. In Moni Naor, editor, *Advances in Cryptology - EUROCRYPT 2007*, pages 1–22, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[SM88] Akihiro Shimizu and Shoji Miyaguchi. Fast data encipherment algorithm feal. In David Chaum and Wyn L. Price, editors, *Advances in Cryptology — EUROCRYPT' 87*, pages 267–278, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[SSA+09] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. Short chosen-prefix collisions for md5 and the creation of a rogue ca certificate. Cryptology ePrint Archive, Report 2009/111, 2009. `https://eprint.iacr.org/2009/111`.

[Wag99] David Wagner. The boomerang attack. In Lars Knudsen, editor, *Fast Software Encryption*, pages 156–170, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[WT86] A. F. Webster and S. E. Tavares. On the design of s-boxes. In Hugh C. Williams, editor, *Advances in Cryptology — CRYPTO '85 Proceedings*, pages 523–534, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[Wä18] Dietmar Wätjen. *Kryptographie*. Springer Vieweg, Wiesbaden, 3 edition, 2018.