LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

# Grundlagenpraktikum: Rechnerarchitektur

Gruppe 199 – Abgabe zu Aufgabe A505
Sommersemester 2022

Dorian Zedler Finn Dröge Thomas Florian

## 1 Einleitung

Der Begriff „Hashing“ bedeutet ursprünglich „zerhacken und mischen“. Das bedeutet, eine Hashfunktion zerhackt und mischt Informationen und leitet daraus ein Hash-Ergebnis ab [**?**]. 1989 hat Ron Rivest die Hashfunktion Message-Digest 2 Algorithm (MD2) veröffentlicht. Für eine lange Zeit war der MD2-Funktion eine der meist verwendeten Hashfunktionen. Die Hauptanwendungsgebiete sind Passwortsicherung, Nachrichtenauthentifizierungscodes, digitale Signaturen und somit auch Zertifikate [**?**]. 2004 wurde die Hashfunktion im Paper „The MD2 Hash Function is Not One-Way“ von Frédéric Muller als unsicher erklärt [**?**].

In dieser Arbeit wird auf die Eigenschaften von Hashfunktionen eingegangen, sowie deren Anwendungsbereich. Schwachpunkte der MD2-Funktion werden aufgezeigt und alternative Algorithmen werden genannt. Insbesondere wird die Berechnung des MD2-Hashes erklärt und die Performance verschiedener Implementierungen analysiert.

### 1.1 Die Eigenschaften kryptografischer Hashfunktionen

Sichere Hashfunktionen müssen 3 Kriterien erfüllen:

#### 1.1.1 Preimage-Resistance

Preimage-Resistance besagt, dass eine Hashfunktion nur sehr schwer umkehrbar sein darf. Das heißt, dass die zu einem Hash gehörige Eingabe nur mit sehr hohem Rechenaufwand bestimmt werden kann. [**?**]
Mit gegebenem $y = H(x)$ sollte es schwer sein, ein $x' \neq x$ zu finden, sodass gilt:
$H(x') = y$. [**?**]

#### 1.1.2 Second-Preimage-Resistance

Second-Preimage-Resistance ist dann gewährleistet, wenn es für eine gegebene Ein- und Ausgabe einer Hashfunktion sehr schwierig ist, eine weitere Eingabe zu finden, die dieselbe Ausgabe produziert. [**?**]
Mit gegebenem $x$ und $y = H(x)$ sollte es schwer sein, ein $x' \neq x$ zu finden, sodass gilt:
$H(x') = y$. [**?**]

### 1.1.3 Collision-Resistance

Die Collision-Resistance ist dann erfüllt, wenn es nur sehr wenige gleiche Hashes für unterschiedliche Eingaben gibt. Bei einer Hashfunktion mit einer hohen Collision-Resistance ist die Second-Preimage-Resistance auch hoch. Wenn es nur wenige Kollisionen des Hashes gibt, ist es folglich auch schwerer, eine Eingabe zu finden, die denselben Hash produziert. [**?**]
Es sollte schwer sein, ein $x$ und $x'$ zu finden, sodass gilt: $x \neq x'$ und $H(x) = H(x')$. [**?**]

## 1.2 Angriffe auf Hashfunktion

Zu jedem der oben genannten Kriterien gibt es einen entsprechenden Angriff. Der Preimage-Angriff und der Second-Preimage-Angriff kann über eine Brute-Force-Implementierung mit einer Komplexität von $2^n$ durchgeführt werden, wobei n der Anzahl an Bits des Hashes entspricht. Da die MD2-Funktion einen 128 Bit großen Hash erstellt, hat dieser eine Komplexität von $2^{128}$. Aufgrund des Geburtstagsparadoxons hat der Brute-Force-Kollisionsangriff die Komplexität $2^{n/2}$ (für MD2: $2^{64}$). Eine Hashfunktion gilt als sicher, wenn es keine effizienteren Angriffe als diese drei Brute-Force-Angriffe gibt. [**?**]

### 1.2.1 Preimage-Angriff

Bei einem Preimage-Angriff soll zu einem gegebenen Hash die dazu passende Nachricht gefunden werden. Der aktuell schnellste Preimage-Angriff auf MD2 wurde im Jahr 2008 von Søren S. Thomsen veröffentlicht [**?**]. Dort wird beschrieben, wie die vorherigen Zwischenergebnisse, anhand eines entdeckten Musters in der Berechnung, wiederhergestellt werden können. Es kann durch die bijektive Eigenschaft von XOR aus zwei Elementen auf das dritte geschlossen werden [**?**]. Dadurch gibt es weniger mögliche Permutationen, welche die Eingangsnachricht annehmen kann. Insgesamt reduziert dieses Verfahren die Komplexität des Preimage-Angriffs von $2^{128}$ auf $2^{73}$.

### 1.2.2 Kollisionsangriff

Basierend auf ähnlichen Mustererkennungen kann auch ein Kollisionsangriff auf den MD2-Hash durchgeführt werden. Somit verringert sich die Komplexität des oben beschriebenen Kollisionsangriffs von $2^{64}$ auf $2^{54}$ mithilfe einer „Collision Attack with Arbitrary Chaining Input“ [**?**].

## 1.3 Alternativen zu MD2

Wegen der oben genannten Angriffe gilt die MD2-Hashfunktion seit 2004 als unsicher. Mögliche Alternativen wären die Nachfolger von MD2, die auch von Ron Rivest entwickelt wurden: MD4 und MD5. MD4 wurde 1991, nur ein Jahr nach Veröffentlichung, bereits als unsicher erklärt und gilt daher nicht als Alternative [**?**]. 2005 wurde MD5 als unsicher gegen Kollisionsattacken deklariert [**?**]. Gängige Alternativen sind die Secure

Hashing Algorithms (SHA-1, SHA-2, SHA-256, etc.) [**?**]. Insbesondere SHA-256 gilt als sicher (Stand: August 2015) [**?**] und wird daher als weitverbreitete Alternative genutzt.

## 1.4 Anwendungen von kryptografischen Hashfunktionen

Das heutzutage wohl wichtigste Anwendungsgebiet von Hashes ist das Sichern von Passwörtern zur Authentifizierung an computergestützten Systemen [**?**]. Die Passwörter werden gehasht für den Fall, dass eine Datenbank mit den Passwörter-Hashes in die Hände Dritter gelangt. Durch das Hashing ist es praktisch unmöglich, den Klartext der Passwörter festzustellen. Um eine Wörterbuch-Attacke vorzubeugen, wird oft zusätzlich vor dem Hashing noch ein „salt“ (zufällige Daten) an das Passwort angehangen.
Eine weitere gängige Anwendung ist die digitale Signierung von Daten. Hierbei wird der aus den Daten gebildete Hash signiert [**?**]. So kann sichergestellt werden, dass jene Daten nach der Signierung nicht mehr verändert wurden, da jede kleine Änderung an den Daten den Hash stark verändern würde.
Auch wenn es darum geht, eine Kette von Ereignissen kryptografisch zu fixieren, ist Hashing ein wichtiges Werkzeug. In einer sogenannten Blockchain enthält jedes Element (z.B. eine Überweisung) den Hash des vorherigen Elementes [**?**]. Dadurch ist es möglich, die gesamte Kette an Überweisungen zu verifizieren. Ein praktisches Beispiel dafür sind Kryptowährungen wie Bitcoin.

## 1.5 Berechnung von Prüfsumme und Hashwert in der MD2-Hashfunktion

Die MD2-Hashfunktion lässt sich in drei Schritte gliedern: Padding hinzufügen, Prüfsumme berechnen und einer finalen Berechnung.

### 1.5.1 Padding

Damit eine Nachricht mit MD2 gehasht werden kann, muss die Länge der Nachricht zunächst ein Vielfaches von 16 sein. MD2 benutzt dafür das Paddingverfahren PKCS#7. Dieses fügt an eine Nachricht N Bytes mit dem Wert N an.
Im folgenden Beispiel ist N = 5:

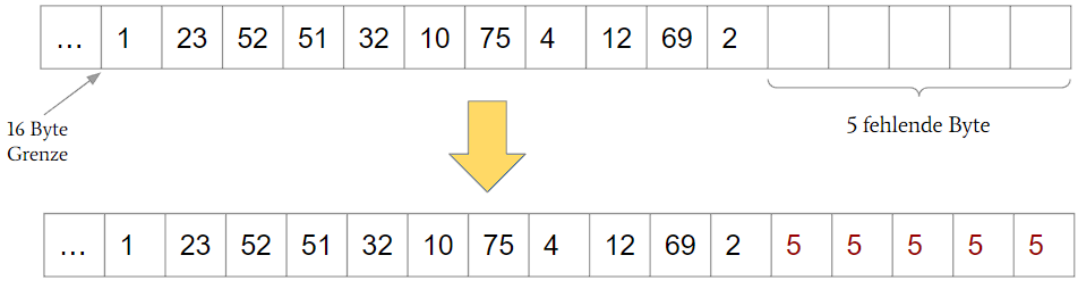


Abbildung 1: Padding Beispiel

Bei dem Verfahren muss mindestens ein Byte eingefügt werden. Somit werden genau 16 Bytes mit dem Wert 16 angefügt, falls die Nachricht bereits ein Vielfaches von 16 ist. Damit stellt man sicher, dass das letzte Byte eindeutig zum Padding gehört.
Zudem wird ein trivialer Preimage-Angriff verhindert:
Zur Vereinfachung wird angenommen, dass das erwähnte Paddingverfahren die Nachrichtenlänge auf ein Vielfaches von 8 bringt.
Sei $m_1 = [01, 01, 01, 01, 01]$ die Eingabe. Dann wird $m_1$ mit diesem Paddingverfahren zu $p_1 = [01, 01, 01, 01, 01, 03, 03, 03]$ erweitert.
Existiert die Bedingung der Mindestpaddinggröße nicht, so könnte nun $m_2 = [01, 01, 01, 01, 01, 03, 03, 03]$ als Nachricht gewählt werden. Dann ist $m_2$ inklusive Padding: $m_2 = p_1 = [01, 01, 01, 01, 01, 03, 03, 03]$. Dadurch kommt es zu $hash(m_1) = hash(m_2)$, wobei $m_1 \neq m_2$. Mit der Mindestpaddinggröße ergibt, sich $p_2' = [01, 01, 01, 01, 01, 03, 03, 03, 08, 08, 08, 08, 08, 08, 08, 08]$, was den Preimage-Angriff verhindert. In der Regel gilt dann $hash(m_1) \neq hash(m_2)$, weshalb kein trivialer Preimage-Angriff möglich ist.
Beim MD2-Hash gilt zusätzlich, dass für den Fall einer leeren Nachricht ein Padding mit 16 Bytes angefügt wird. Die Prüfsumme und der Hash können damit ohne eine Fallunterscheidung gebildet werden **[?]**, **[?]**.

### 1.5.2 Substitutionsbox

Zur weiteren Berechnung wird eine Substitutionsbox verwendet, die sich aus Nachkommastellen von $\pi$ zusammensetzen lässt. Hierbei werden die Nachkommastellen mithilfe des Durstenfeld-Shuffle ausgewählt. **[?]**. Die Zahlen der Box werden als Zufallszahlen angesehen und werden verwendet, um das Ergebnis zu streuen.

### 1.5.3 Prüfsumme

An die Padding-Bytes werden 16 weitere Bytes angehängt, die sogenannte Prüfsumme. Diese Bytes werden zunächst mit 0 initialisiert. Für die Berechnung der Prüfsumme wird für jedes n-te Byte der Prüfsumme die XOR-Operation auf das Byte selbst mit einem Wert von der Substitutionsbox $\pi$ ausgeführt:

$$Prüfsumme[n] = Prüfsumme[n] \oplus \pi[k]$$

Der Index k der Substitutionsbox $\pi$ in dem i-ten Schleifendurchlauf ist das Ergebnis der XOR-Operation von dem $(i * 16 + n)$-ten Byte der Nachricht (inklusive Padding) mit dem vorherigen Byte der Prüfsumme:

$$k = Nachricht[i * 16 + n] \oplus Prüfsumme[n - 1 \mod 16]$$

Hierbei gilt es zu beachten, dass das erste Byte keinen Vorgänger hat. Hier wird das letzte Byte als Vorgänger verwendet. Deswegen ist die Formel für den Vorgänger von n als $n - 1 \mod 16$ definiert.
In der folgenden Grafik wird die Berechnung des achten Bytes der Prüfsumme veranschaulicht:

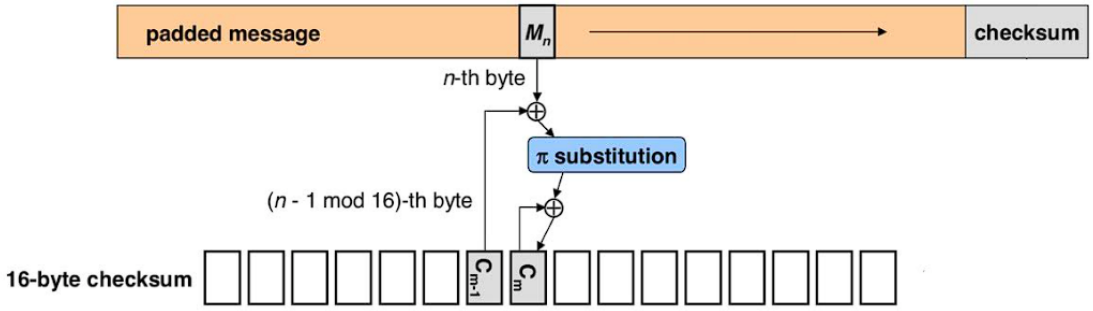


Abbildung 2: MD2-Prüfsumme [?]

Beispielsweise wird im dritten Schleifendurchlauf das oben dargestellte achte Byte der Prüfsumme wie folgt berechnet:

$$Prüfsumme[8] = Prüfsumme[8] \oplus \pi[k]$$

mit

$$k = Nachricht[3 * 16 + 8] \oplus Prüfsumme[8 - 1 \mod 16]$$

### 1.5.4 Finale Berechnung

Zur finalen Berechnung werden zunächst weitere 48 Byte Speicher alloziert und mit dem Wert 0 initialisiert. Im Folgenden werden diese 48 Bytes in 3 Blöcke unterteilt: Block A beschreibt die ersten 16 Byte, Block B die mittleren 16 Byte und Block C die letzten 16 Byte. Die Daten des zuvor erzeugten Buffers aus Nachricht, Padding und Prüfsumme werden mithilfe von weiteren Operationen verarbeitet und in die entsprechenden Blöcke geschrieben.
Als Erstes wird vom Buffer ein 16-Byte Block in Block B kopiert. Anschließend wird in Block C das Ergebnis der XOR-Operation in Block A und B geschrieben. Danach werden alle Bytes der drei Blöcke 18 Mal durchlaufen. Zur Berechnung eines Bytes wird die XOR-Operation auf ein Element der Substitutionsbox $\pi$ und das Byte selbst angewandt:

$$Byte[k] = Byte[k] \oplus \pi[t]$$

Das ausgewählte Element der Substitutionsbox wird durch das vorherige Byte und den Schleifendurchlauf j bestimmt:

$$t = (Byte[k - 1] + j) \mod 256$$

Somit ist jedes Byte von seinem Vorgänger abhängig. Da das erste Byte keinen Vorgänger hat, wird hier das letzte Byte als Vorgänger verwendet:

$$Byte[0] = Byte[0] \oplus \pi[t]$$

mit

$$t = (Byte[47] + j) \mod 256$$

im ersten der 18 Schleifendurchläufe wird außerdem $t = 0$ gesetzt.

Dieses Verfahren wird für jeden 16-Byte Block des Buffers wiederholt. Nach allen Iterationen steht der Hash in Block A. Das beschriebene Verfahren wird in Abbildung 3 dargestellt.

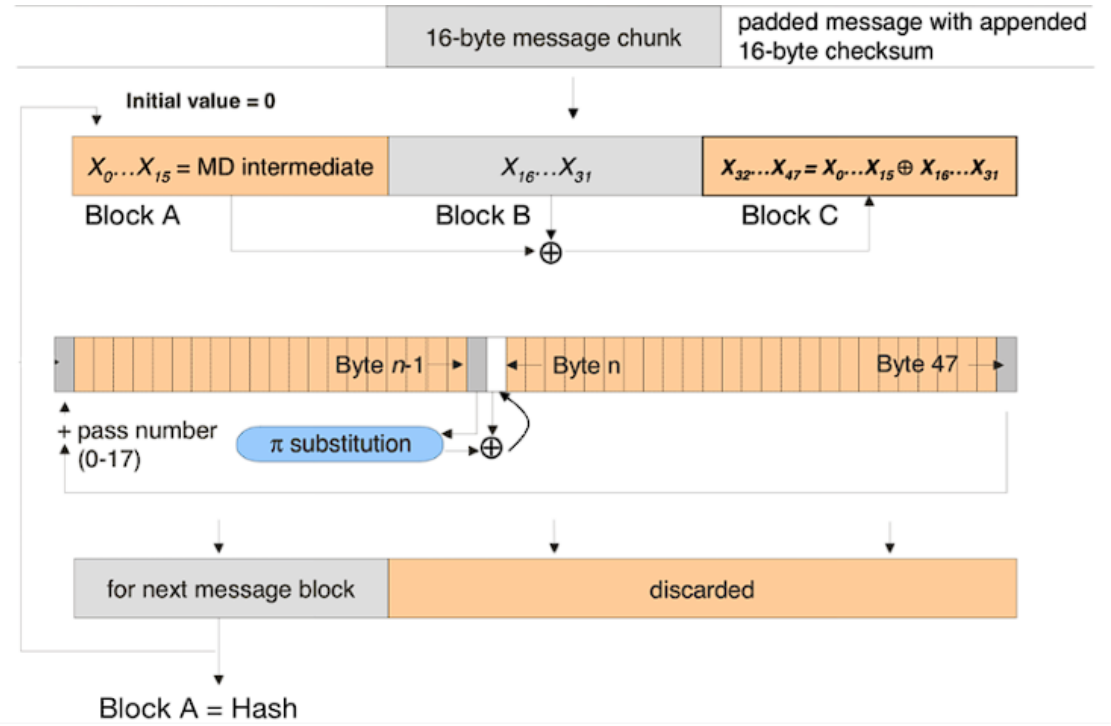


Abbildung 3: MD2 Finale Hash Berechnung [?]

## 2 Lösungsansatz

### 2.1 V0 - Basis-Implementierung

Die Basis-Implementierung wurde mithilfe des Pseudocodes, welcher auf in der RFC1319 Spezifikation [?] zu finden ist, erstellt. Sie folgt genau dem oben beschriebenen Schema (siehe Kapitel 1.5).

### 2.2 V1 - Optimierte Implementierung mit SIMD

In dieser Implementierung wird der Code mittels SIMD-Operationen optimiert. Eine for-Schleife zur Berechnung des Hashes wird durch 5 Intrinsics-Operationen ersetzt. Beispielsweise wird in der Schleife eine XOR-Operation auf jedes Element eines Buffers ausgeführt. Diese vielen iterativen Berechnungen werden durch den Befehl `_mm_xor_si128(x,y)` ersetzt.
Das Einsetzen des Paddings wird mithilfe einer LUT gemacht. Hierbei verbessert sich

die Laufzeit jedoch nicht, weil trotzdem ein `memcpy()` aufrufen wird.
Kleinere Optimierungen werden auch vorgenommen, wie das Ersetzen des Modulo-Operators durch ein logisches UND.
Weitere SIMD-Optimierungen können nicht gemacht werden, da die Berechnungen von Elementen immer von der Berechnung ihres Vorgängers abhängen.

## 2.3 V2 - Implementierung mit partiellem Einlesen der Datei

In den anderen Implementierungen wird die gesamte Datei zu Beginn in den RAM geladen. Zusätzlich muss nochmals so viel Speicher alloziert werden, um Platz für das Padding und die Prüfsumme zu schaffen. Daher ist die Dateigröße auf maximal die Hälfte des Hauptspeichers abzüglich Padding und Prüfsumme beschränkt. Der Speicherbedarf ist also in $\mathcal{O}(n)$.
In dieser Implementierung wird die Datei in 16-Byte Blöcken geladen, welche jeweils einzeln verarbeitet und danach wieder aus dem Hauptspeicher gelöscht werden. Daher ist der Speicherbedarf hier in $\mathcal{O}(1)$ und es können beliebig große Dateien verarbeitet werden.
Bei der Implementierung dieser Optimierung konnten die Vorgaben aus der Aufgabenstellung nicht eingehalten werden, da sie das vollständige Laden der Datei in den Hauptspeicher voraussetzen. Auch die Funktionen

```
1 void md2_checksum(size_t len, uint8_t* buf)
```

und

```
1 void md2_hash(size_t len, const uint8_t buf[len], uint8_t out[16])
```

konnten aus diesem Grund nicht so umgesetzt werden, da auch sie das Vorhandensein der gesamten Datei im Hauptspeicher voraussetzen.

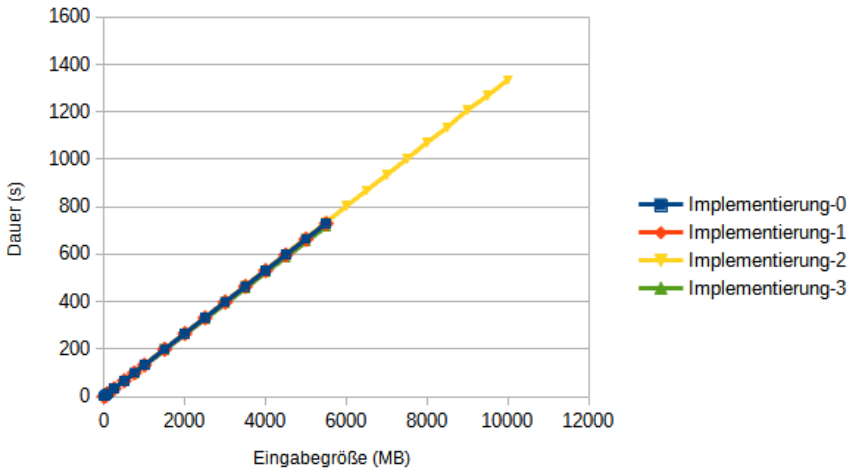


Abbildung 4: Vorteil der zweiten Implementierung

Die Vorteile dieser Implementierung zeigen sich in Abbildung 4. Nachdem die Größe der Eingabe ca. 6 Gigabyte überschreitet, funktioniert bei dem verwendeten Rechner nur noch diese Implementierung.

### 2.4 V3 - Optimierte Implementierung mit Threading

Bei dieser Implementierung soll die Berechnung des Hashes in zwei Threads aufgeteilt werden. Ein Thread soll sich um die Berechnung der Prüfsumme durchführen, während der zweite den Hash für die Nachricht berechnen soll.
Aus dem Main-Thread werden mithilfe der Funktion `pthread_create()` diese beiden Threads erstellt. Sobald diese mit ihrer Berechnung fertig sind, werden die Threads wieder mittels `pthread_join()` wieder in den Main-Thread zusammengeführt.
Durch das Aufteilen wurde die Prüfsumme noch nicht mit in den Hash gerechnet. Deswegen wird die Funktion zur Berechnung des Hashes ein weiteres Mal aufgerufen.
Auch in dieser Implementierung konnten die Vorgaben der Aufgabenstellung nicht eingehalten werden. Die Funktionsköpfe mussten so angepasst werden, dass diese für Threads geeignet sind.

### 2.5 V4 - Referenzimplementierung

Als vierte Implementierung wurde die Referenzimplementierung aus der RFC1319 Spezifikation [**?**] übernommen. Sie ist ein wichtiges Werkzeug, um die Korrektheit der anderen Implementierungen zu beurteilen.

## 3 Korrektheit

Um die Korrektheit der Implementierungen zu testen, wurden zwei Strategien angewendet:
Zuerst wurden Beispielwerte von der RFC1319 Spezifikation [**?**] getestet. Zusätzlich wurden einige Dateien mit zufälligen Daten getestet und mit den Ergebnissen der Referenzimplementierung (siehe 2.5) verglichen.
Da schon kleine Fehler im Algorithmus zu sehr großen Abweichungen im Ergebnis führen würden, kann davon ausgegangen werden, dass die Implementierungen korrekt sind.

## 4 Performance-Analyse

Getestet wurde auf einem System mit einem i7-2670QM Prozessor, 2.20GHz, 11GB Arbeitsspeicher, Ubuntu 20.04, 64Bit, Linux-Kernel 5.4. Kompiliert wurde mit GCC 9.3.0 mit der Option `-O3`. Die Berechnungen wurden jeweils 3-mal durchgeführt und das arithmetische Mittel für jede Eingabegröße wurde berechnet.

Um einen groben Überblick über die Performance zu bekommen, wurde zunächst die Laufzeit der gesamten Berechnung des Hashes gemessen.

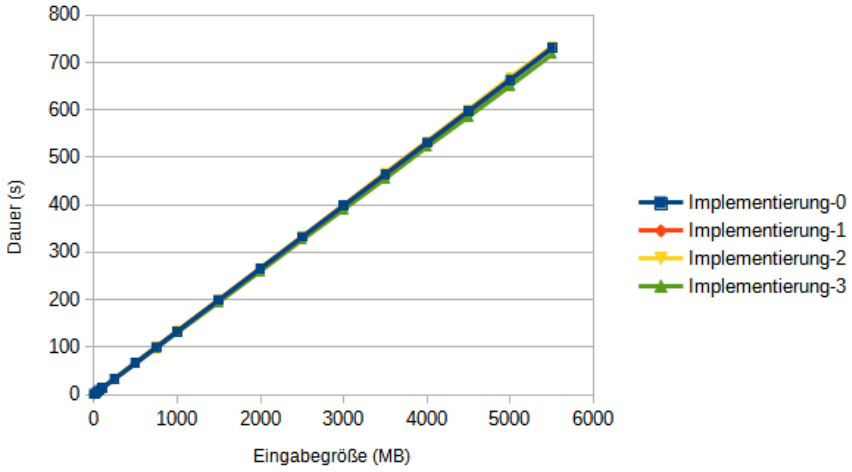


Abbildung 5: Laufzeitanalyse

In Abbildung 5 ist zu erkennen, dass die asymptotische Laufzeit aller Algorithmen in $\mathcal{O}(n)$ liegt. Dies lässt sich dadurch begründen, dass in verschachtelten for-Schleifen maximal eine der Schleifen über n iteriert. Alle anderen Schleifen iterieren über eine konstante Zahl.

Die größte Schwierigkeit bei der Optimierung des MD2-Algorithmus ist die zweite Schleife der finalen Berechnung. Diese Schleife lässt sich nicht optimieren und benötigt die meiste Zeit bei der Berechnung. Um diese Problematik genauer zu analysieren, wurde die zweite Performance-Analyse entwickelt. Hierzu wird die Gesamtlaufzeit in drei Teil-Laufzeiten gegliedert: die Berechnung der Prüfsumme, der ersten sowie der zweiten Schleife im finalen Schritt.
Das Verhältnis dieser verschiedenen Laufzeiten ist in Abbildung 6 dargestellt.

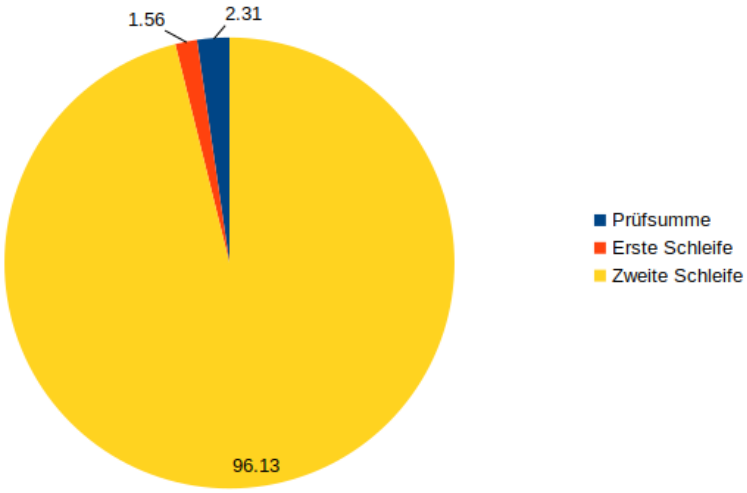


Abbildung 6: Verhältnis der Teil-Laufzeiten (in Prozent) [**?**]

Gemessen wurde mit 100 Megabyte Eingabedaten und dem Durchschnitt über zehn Wiederholungen.
Mit diesem Test ist feststellbar, dass die zweite Schleife im finalen Schritt den Großteil der gesamten Laufzeit ausmacht. Das erklärt, weshalb der Effekt der oben beschriebenen Optimierungen minimal ist.
In der dritten Implementierung (Threading) kann lediglich ein Speed-Up durch die Parallelisierung der Prüfsumme erreicht werden. Somit werden bei 100 Megabyte maximal 2,31% der Geschwindigkeit des Programms optimiert. In der ersten Implementierung (SIMD) wird die erste Schleife mithilfe von SIMD-Operationen optimiert. Hier kann der Speed-Up maximal 1,56% betragen. Diese beiden Erkenntnisse lassen auch aus den Abbildungen 7 und 8 ablesen.

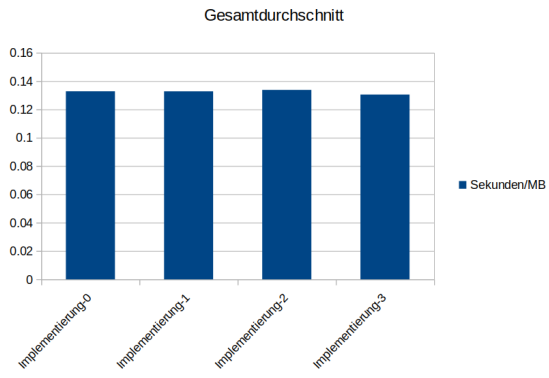


Abbildung 7: Normierte Laufzeitanalyse

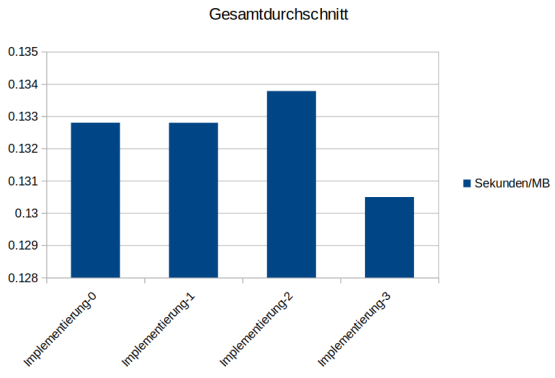


Abbildung 8: Vergrößerte Laufzeitanalyse

Im ersten Balkendiagramm ist der Speed-Up kaum zu erkennen. Dies lässt sich mit der nicht optimierbaren zweiten Schleife begründen. Beim Vergrößern des Diagramms (siehe Abbildung 8) wird der oben beschriebene, geringe Speed-Up in Implementierung 3 erkennbar. Implementierung 1 hat einen noch kleineren Speed-Up, da mit SIMD nicht die vollen 1,56% wegoptimiert werden können.

## 5 Zusammenfassung und Ausblick

Der MD2-Algorithmus wurde im Jahr 1989 entwickelt und war lange ein Standard-Hashverfahren für kryptografische Anwendungsbereiche. Er wurde 2004 aufgrund von Attacken als unsicher deklariert. Während dem Implementieren hat sich herausgestellt, dass sich der Algorithmus nur schwer optimieren lässt. Kleine Optimierungen können bei großen Datenmengen mittels Threading umgesetzt werden. Diese Schwerfälligkeit des Optimierens bringt aber auch Vorteile für den Algorithmus, denn Brute-Force-Angriffe müssen Permutationen ausprobieren, bis das gewünschte Ergebnis erreicht wird. Je zeitintensiver die Berechnung von Hashes sind, desto aufwendiger werden auch die entsprechenden Angriffe.
Weitere Optimierungsmöglichkeiten, welche in diesem Projekt nicht umgesetzt wurden, könnten möglicherweise aus dem Vorgehen der genannten Angriffe resultieren. Dabei könnte es möglich sein, das in den Angriffen erkannte Muster als Anlaufstelle für weitere Parallelisierungen im Code zu benutzen.
Durch Moore's Law kann davon ausgegangen werden, dass sich die Kosten eines Angriffes auf ein Kryptosystem alle 18 Monate halbieren [**?**]. Deswegen ist es nötig immer komplexere und aufwändigere Hashfunktionen zu entwickeln. In der Zukunft müssen auch die genannten Alternativen zu MD2, so wie MD2 heute, durch solche neuen Hashfunktionen abgelöst werden.

## Literatur

[1] Arjen K. Lenstra . *Key Lengths*. Citibank, N.A., and Technische Universiteit Eindhoven, 2006. `https://infoscience.epfl.ch/record/164539/files/NPDF-32.pdf`, besucht am 24.07.2022.

[2] D. Atkins. *PGP Message Exchange Formats*, August 1996. `https://www.rfc-editor.org/rfc/rfc1991`, besucht am 21.07.2022.

[3] B. Kaliski. *The MD2 Message-Digest Algorithm*. RSA Laboratories, April 1992. `https://datatracker.ietf.org/doc/html/rfc1319`, besucht am 16.07.2022.

[4] Bert den Boer, Antoon Bosselaers. *An Attack on the Last Two Rounds of MD4*, 1991. `https://web.archive.org/web/20030523231212/http://dsns.csie.nctu.edu.tw/research/crypto/HTML/PDF/C91/194.PDF`, besucht am 23.07.2022.

[5] Di Pierro, Massimo. What is the blockchain? *Computing in Science Engineering*, 19(5):92–95, 2017.

[6] F. Muller. *The MD2 Hash Function Is Not One-Way*, 2004. https://www.iacr.org/archive/asiacrypt2004/33290211/33290211.pdf, besucht am 20.07.2022.

[7] Praveen Gauravaram. Security analysis of salt||password hashes. In *2012 International Conference on Advanced Computer Science Applications and Technologies (ACSAT)*, pages 25–30, 2012.

[8] L. Knudsen et al. *Cryptanalysis of MD2*. Technical University of Denmark, 2010. https://link.springer.com/content/pdf/10.1007/s00145-009-9054-1.pdf, besucht am 23.07.2022.

[9] Xingquan Zhu Lianhua Chi. *Hashing Techniques: A Survey and Taxonomy*. January 2018.

[10] mikeazo. 2014. https://crypto.stackexchange.com/questions/11935/how-is-the-md2-hash-function-s-table-constructed-from-pi, besucht am 20.07.2022.

[11] National Institute of Standards and Technology. *Secure Hash Standard (SHS)*, August 2015. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf, besucht am 23.07.2022.

[12] R. Housley. *Cryptographic Message Syntax (CMS)*. Vigil Security, September 2009. https://datatracker.ietf.org/doc/html/rfc5652#section-6.3, besucht am 18.07.2022.

[13] RWTH Aachen. *Kommunikation und verteilte Systeme, Kapitel 2.3 Hash Functions*, 2007. http://www-i4.informatik.rwth-aachen.de/content/teaching/lectures/sub/sikon/sikonSS07/05_Hash_1P.pdf, besucht am 20.07.2022.

[14] S. Thomsen. *An improved preimage attack on MD2*. Technical University of Denmark, 2008. https://eprint.iacr.org/2008/089.pdf, besucht am 20.07.2022.

[15] S. Turner. *MD2 to Historic Status*. Independent Educational Consultants Association, March 2011. https://datatracker.ietf.org/doc/html/rfc6149#section-7, besucht am 23.07.2022.

[16] X. Wang et al. *How to Break MD5 and Other Hash Functions*. Shandong University, 2005. http://merlot.usc.edu/csac-f06/papers/Wang05a.pdf, besucht am 23.07.2022.