LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Gruppe 276 – Abgabe zu Aufgabe A505

Sommersemester 2023

Awar Satar Yiyang Xie Jingjing Yu

1 Einleitung

Im digitalen Zeitalter sind die Sicherheit und die Integrität von Daten von großer Bedeutung. Kryptographische Hashfunktionen spielen eine entscheidende Rolle in diesem Bereich und sind weitreichend in verschiedenen Domänen eingesetzt. Der Fokus dieser Arbeit liegt auf dem MD2 Hashing Algorithmus, welcher im RFC 1319 der Internet Engineering Task Force (IETF) beschrieben ist [1].
Obwohl der MD2 Algorithmus mittlerweile als obsolet gilt, bietet er einen interessanten Ansatz zur Untersuchung der Grundlagen und der Entwicklung kryptographischer Hashfunktionen.
Die zu verarbeitenden Daten in solchen Hashing Algorithmen müssen stets ein Vielfaches der Blocklänge sein, was durch Padding erreicht wird. Eines der Verfahren, das diese Anforderung erfüllt, ist das PKCS#7 [2].
Die erste Aufgabe besteht darin, dieses Verfahren genauer zu erforschen und in dieser Arbeit zu beschreiben.
Darüber hinaus gilt es, die genaue Funktionsweise der Berechnung von Prüfsumme und Hashwert zu erläutern. Insbesondere wird in beiden Schritten eine Substitutionsbox verwendet, deren Werte es zu klären gilt.
Letztendlich werden wir die Angriffe auf den MD2-Algorithmus sowie die Alternativen, die heute verwendet werden, diskutieren. Dadurch werden die Gründe für das Auslaufen des MD2 und die Fortschritte in der Kryptographie verdeutlicht.

2 Lösungsansatz

2.1 Kryptographischer Hintergrund

Eine kryptographische Hashfunktion, auch „Message Digest"-Funktion genannt ist eine Familie von Hashing Algorithmen , die eine Nachricht m auf einen „Digest" $h(m)$ reduzieren. Die Idee ist also eine Nachricht durch Hashing kompakt zu repräsentieren, was bspw. für Signaturen von langen Texten Anwendung findet [3] und da beim Hashing das Urbild der Funktion in der Mächtigkeit die Ergebnismenge deutlich übersteigt, ist es allerdings möglich sogenannte Kollisionen zu finden, also ein $m_k \neq m$ mit $h(m) = h(m_k)$.
Für kryptographische Hashfunktionen wurden entsprechend Eigenschaften festgelegt, die aus einer Sicherheitsperspektive erstrebenswert sind:[4]

(1) $\forall m : H(m)$ ist einfach zu berechnen.

(2) **Urbildresistenz:** Für ein gegebenes $h = H(m)$ ist das Bestimmen des Wertes m mit $m = H^{-1}(h)$ nicht effizient möglich, d.h. es ist nicht effizient möglich mittels $H(m)$ auf m zu schließen.

(3) **schwache Kollisionsresistenz:** Es ist nicht effizient möglich zu einem gegebenem m ein m' zu finden, sodass $H(m) = H(m')$.

(4) **starke Kollisionsresistenz:** Es ist nicht effizient möglich ein beliebiges Paar (m, m') zu finden, sodass $H(m) = H(m')$.

MD2 Hashing besteht aus 3 Schritten. Zuerst wird die Nachricht auf ein Vielfaches von 16 Byte ergänzt. Anschließend wird eine Prüfsumme berechnet und an das Ende der Nachricht angehängt. Zum Schluss wird die Nachricht in 18 Durchläufen komprimiert oder gehasht. Wir werden auf jeden dieser Schritte im Detail eingehen.

2.2 PKCS#7 Padding: Implementierung und Optimierung

In der kryptographischen Welt ist das Padding von Nachrichten eine weit verbreitete Technik, um die Größe der zu verarbeitenden Daten auf ein Vielfaches der Blocklänge zu bringen. Ein gängiges Verfahren dafür ist das PKCS#7 Padding, das hier genauer erörtert wird. Es handelt sich um ein standardisiertes Verfahren, das in RFC 2315 ausführlich beschrieben wird [2].
Die grundlegende Idee des PKCS#7 Paddings besteht darin, den fehlenden Platz in einem Block mit einer Sequenz von Bytes zu füllen, deren Wert der Länge der Sequenz entspricht. Bei einer Blocklänge von 16 Bytes kann also jeder Block, unabhängig von seiner ursprünglichen Größe, auf die erforderliche Länge gebracht werden.
Eine Optimierungsstufe kann durch den Einsatz von SIMD-Instruktionen erreicht werden, wie in der Funktion `padding_opt_V0` gezeigt. Durch die Verwendung der SIMD-Instruktion `_mm_storeu_si128` können mehrere Bytes gleichzeitig geschrieben werden, was zu einer deutlich verbesserten Leistung führt.
Insgesamt ermöglichen diese Optimierungen eine effizientere Anwendung des PKCS#7 Paddings, was in vielen kryptographischen Kontexten von großem Nutzen sein kann.

2.3 Berechnung der Prüfsumme

Die Berechnung der Prüfsumme und des Hashwerts sind zentrale Bestandteile des MD2-Algorithmus. Ohne dem Prüfsummen-Byte ist der MD2 unsicher [5]. In diesem Abschnitt wird die genaue Funktionsweise dieser Berechnungen erläutert und der Einsatz der Substitutionsbox (S-Box) in diesen Prozessen verdeutlicht. Die S-Box, die in diesen Berechnungen verwendet wird, ist ein wesentliches Element. Es handelt sich um ein festes, vordefiniertes Array von 256 Werten, das für die Substitution der Eingabewerte verwendet wird. Die Werte in der S-Box sind so gewählt, dass sie den kryptographischen Sicherheitsanforderungen genügen und das Ausgabebild stark verändern, selbst wenn nur kleine Änderungen an der Eingabe vorgenommen werden (Diffusion). Die Prüfsumme wird erstellt, indem eine Checksummen-Funktion auf die Nachricht angewendet

wird, wobei jedes Byte der Nachricht in 16 Byte Blöcken verarbeitet wird. Das aktuelle Byte des Blocks nennen wir von nun b_i und das 16 Byte Array der Checksumme s. Ein laufender Totalisator l beginnt am Anfang der Funktion zunächst bei Null $c = 0$. Anschließend wird der Index des nächsten Wertes aus der Substitutionsbox mit $t = b_i \oplus l$ berechnet. Die Prüfsumme wird an dem Index i dann auf $s_i \oplus t$ gesetzt und letztlich der Totalisator mit $l = s_i$ aktualisiert.
Die S-Box oder S-Tabelle wird in RFC 1319 [1] beschrieben wie folgt:

> This step uses a 256-byte "random"permutation constructed from the digits of π.

Wirft man allerdings einen Blick auf die S-Box ist es schwer den Zusammenhang zwischen den Werten in der Tabelle und π zu erkennen. Ron Rivest klärte dies in einer E-Mail an einen Forennutzer auf. Zur Generierung der S-Box wird π als Psuedozufallsgenerator verwendet und mittels einer modifizierten Form des Durstenfeld-Shuffles wird eine gleichmäßige Verteilung der Zufallszahlen in der S-Box ermöglicht.[6]

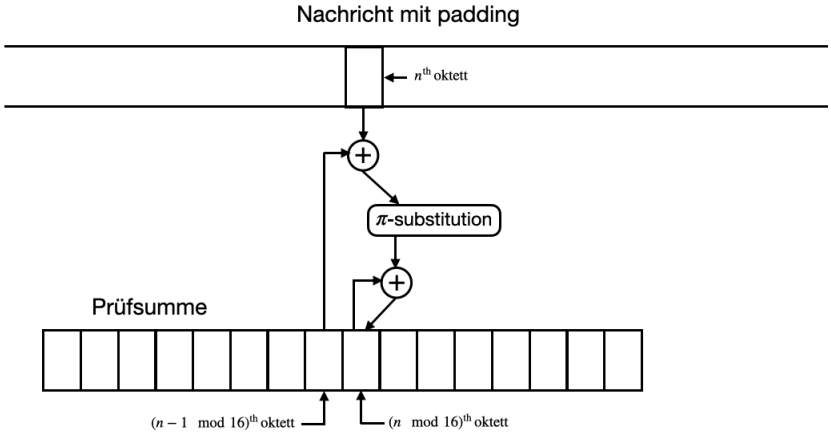

Abbildung 1: Darstellung zur Berechnung der Prüfsumme von MD2[7]

Diese genaue Berechnungsmethode, kombiniert mit der Verwendung der S-Box, trägt zur Sicherheit und Unveränderlichkeit der MD2-Hashfunktion bei.

2.4 Kompressionsfunktion

Bei der Kompressionsfunktion handelt es sich um einen iterativen Prozess, bei dem mehrere Berechnungsschritte ausgeführt werden, die schließlich zu einem 128-Bit-Hashwert führen. Zunächst wird der zu verarbeitende Nachrichtenpuffer in Blöcke von 16 Bytes aufgeteilt und diese sequenziell verarbeitet. Jeder Block wird in das interne Speicher-Array kopiert, das insgesamt 48 Bytes aufnehmen kann und in drei Segmente von je 16 Bytes unterteilt ist. Es ist hervorzuheben, dass die ersten beiden Segmente vor der Verarbeitung jedes Blocks bereits Daten enthalten können, die aus vorherigen Durchläufen stammen. Anschließend wird eine XOR-Operation (exklusives Oder) auf den

ersten und zweiten Segment des internen Arrays angewendet und das Ergebnis in das dritte Segment gespeichert. Das Ergebnis dieser Operation fügt eine weitere Ebene der Durchmischung der Daten hinzu und erhöht so die Diffusion der Hash-Funktion. Darauf folgen 18 Runden von Substitution und Permutation, die auf das gesamte interne Array angewendet werden. In jeder Runde wird jedes Byte des internen Arrays durch einen Wert aus der S-Box ersetzt, wobei der Ersatzwert durch eine XOR-Operation mit einem temporären Wert bestimmt wird. Nach jeder Substitution wird der temporäre Wert aktualisiert, um den nächsten S-Box-Index zu bestimmen. Obwohl diese Methode effektiv ist, bietet sie wenig Spielraum für Optimierungen. Dies liegt vor allem an der Natur des MD2-Algorithmus, der keine Parallelität in seinen Operationen zulässt. Jeder Schritt in der Berechnung ist voneinander abhängig und muss in einer strengen Reihenfolge ausgeführt werden. Daher sind die Möglichkeiten für Verbesserungen in Bezug auf die Geschwindigkeit und Effizienz des Algorithmus begrenzt.

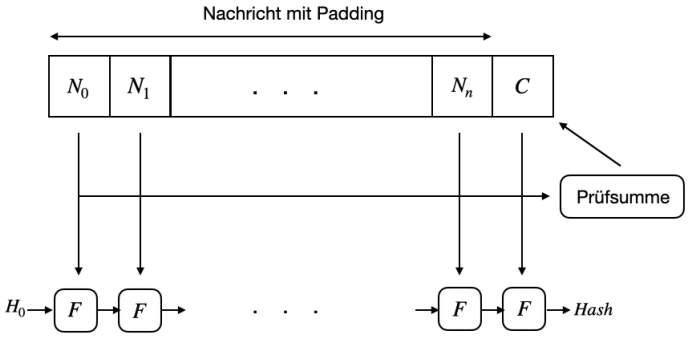

Abbildung 2: Berechnung des Hashwertes. H_0 beträgt bei MD2 0. Die Darstellung orientiert sich an [8]

2.5 Sicherheitsanalyse

In diesem Abschnitt setzten wir uns konkret mit der kryptographischen Sicherheit von MD2 Hashing aus. Hier betrachten wir neben Anwendungsmöglichkeiten auch Sicherheitsrisiken und Angriffsvektoren.

2.5.1 Anwendungsbereiche

Der MD2-Algorithmus besitzt in der Implementierung eine geringe Komplexität und ist zudem schnell berechenbar. Bevor man sicherere Algorithmen entwickelte, gab es viele Anwedungen für MD2, z.B.

- **Überprüfung von Benutzerpasswörtern**
  Der MD2-Algorithmus kann für die Passwortüberprüfung bei der Benutzeranmeldung verwendet werden, indem er vom Klartextabgleich zum Geheimtextabgleich übergeht. Hierbei kommt oft auch sog. „Salting “zum Einsatz. Diese Methode wird im Abschnitt 2.5.3 näher behandelt.

- **Überprüfung von Dateiintegrität**
  Lädt man eine Datei im Internet herunter, kann man oftmals den MD2-Wert der Datei auf der Webseite finden. Nachdem der Download abgeschlossen ist, kann der MD2-Wert der heruntergeladenen Datei berechnet und mit dem MD2-Wert der Quelle verglichen werden. Wenn sie übereinstimmen, ist die Datei vollständig und wurde nicht manipuliert. Dadurch kann man die Echtheit der Datei validieren. Natürlich muss hierfür der Hashwert auf der Webseite vertrauenswürdig sein.

2.5.2 Sicherheitslücken

Von der Verwendung von MD2 Hashing im Internet wird seit 2011 offiziell abgeraten, doch schon seit 2004 wurde MD2 weitesgehend von MD5 abgelöst. [9] Grund dafür sind verschiedene Sicherheitslücken im Algoritmus, die einen Angriff ermöglichen.
Die Länge des Hashwerts von MD2 beträgt nur 128 bit bzw 2^{65}. Wegen des Geburtstagsparadoxons sind nur ca. $2^{65/2} = 2^{64}$ Versuche notwendig um mit einer Wahrscheinlichkeit von mehr als 50% eine Kollision zu finden. Diese relativ kleine Anzahl an Versuchen ist heutzutage effizient leistbar. Folglich sind auch komplexere Kollisionsangriffe auf MD2 nicht nötig, da die rohe Rechenleistung von heutigen Supercomputern genügt, um ein Kollisionspaar zu finden.[4]
Das größere Sicherheitsproblem ist die schwache Urbildresistenz von MD2. Der französische Informatiker Frédéric Muller fand 2004 erstmals eine Möglichkeit das Urblid eines Hashwertes in besserer Laufzeit als Bruteforce zu ermitteln. Seine Methode beruht darauf zuerst verschiedene Urbilder der Kompressionsfunktion zu finden, was durch den Initialisierungsvektor 0 möglich ist und anschließend die Prüfsummen abzugleichen um ein korrektes Urbild zu ermitteln. Die Zeitkomplexität dieser Methode beträgt ca. 2^{104} [8], was unter den von NIST vorgeschriebenen 128-bit an Sicherheit liegt.[10]
Diese schwache Urbildresistenz wird in sogenannten Rainbow Tables zu Nutze gemacht.[11] Diese berechnen die Hashwerte für verschiedenste Passwortkombinationen vor und speichern sie in einer Kette, die zwischen Hashwerten und Passwörtern alterniert. Dafür muss natürlich die Hashfunktion bekannt sein und auch eine Reduktionsfunktion, auf deren Details wir im Rahmen dieser Ausarbeitung nicht weiter eingehen, auf die Hashwerte angewendet werden um wieder Klartexte zu erhalten. Mithilfe dieser vorberechneten Werte, erlaubt man sich einen Zeit-Speicher-Ausgleich.

2.5.3 Sicherheitsmaßnahmen

- **Salz zur Hashfunktion hinzufügen**
  Die Hashfunktion wird zu $hash(salt + str)$. [4] Da der Salt nicht öffentlich verfügbar ist, wird es für einen Angreifer schwieriger, Zeichenketten mit identischen Hashwerten zu konstruieren. Hierbei wird vor dem Hashing das Klartextwort um einen zufälligen ASCII string ergänzt. Dies führt durch Diffusion zu wesentlich anderen Hashwerten als sonst und schützt das System vor Angriffen durch Rainbow Tables. Zudem müsste ein Angreifen dann neben den Passwörtern auch alle möglichen Salts in einem Brute-Force Angriff beachten.

- **Hash-Kollisionen vermeiden**
  Die Verwendung von Rehash oder geschlossenem Hashing mit offener Adressierung sind wirksame Verfahren zur Vermeidung von Hash-Kollisionen.[12] Jedoch führen sie zu einem Anstieg der Rechenzeit bzw. des Speicherplatzes.

- **Modernere Hashfunktionen verwenden**
  Es wird von NIST nicht empfohlen, MD2, MD4, MD5, SHA-1, RIPEMD-Algorithmen zu verwenden, um sensible Informationen wie Benutzerpasswörter zu sichern. [10] SHA-512, SHA-3 oder andere fortschrittlichere Hashfunktionen wären bessere Optionen.

3 Korrektheit

Der MD2 Algorithmus ist deterministisch, berechnet also für gleiche Eingabewerte gleiche Hashwerte. Verschiedene Eingabewerte führen mit Ausnahmen von Kollisionen zu unterschiedlichen Ausgaben.
Da wir hier mit identischen Ausgaben arbeiten, eignet sich eine Diskussion zu Genauigkeit in diesem Abschnitt nicht.

3.1 Testfälle

Um die Korrektheit unserer Implementierung zu überprüfen, haben wir eine Reihe von Tests durchgeführt. Dabei haben wir die von uns berechneten Hashwerte mit denen verglichen, die von verschiedenen Online-MD2-Rechnern[13] erzeugt wurden.
Eine normale .txt-Textdatei wird ausgewählt und der Inhalt dieser Datei mit unserer Implementierung gehasht. Anschließend wird dieselbe Datei auf einem Online-MD2-Rechner hochgeladen und die beiden Hashwerte miteinander verglichen. Diese Tests sollen sicherstellen, dass unsere Implementierung den Inhalt der Textdatei nach MD2 korrekt hasht.

3.2 Fehleranalyse

Eine vorläufige Implementierung hat nicht zu den gewünschten Ergebnissen geführt. Unsere Ergebnisse weichten erheblich von den online generierten Ergebnissen ab. Ferner generierten verschiedene Online-MD2-Rechner unterschiedliche Hashwerte, wobei einige mit unseren übereinstimmten.

3.2.1 Nicht-Text-Dateien

MD2 behandelt eine rohe Datei als Texteingabe, was das Hashen anderer Dateitypen erlauben sollte.
Wir testeten unsere Implementierung mit verschiedenen Dateiformaten wie .o, .pdf, .png, .tar usw. erneut. Da aber dann der rohe Inhalt der Datei gehasht wird und nicht nur Text, muss der Inhalt der Datei mit `uint_8* buf` anstelle von `char* buf` dargestellt

werden. Operationen wie `buf[len] = \0;` können nun ausgelassen werden, da der `string` Datentyp nicht in Verwendung kommt.
Die Leselänge der Quelldatei wird durch die Erstellung der `stat`-Variablen $statbuf$ und `fstat(fileno(file), &statbuf)` ermittelt [14].
Nach der Verbesserung aller oben beschriebenen Punkte berechnet unser Algorithmus einen Hash, der in allen Fällen mit dem Wert des Online-MD2-Rechners übereinstimmt. Dies bestätigt die Korrektheit unserer Implementierung und erlaubt uns die Performanz der Implementierung zu testen.

4 Performanzanalyse

Das verwendete System hat folgende Spezifikationen: Intel(R) Core(TM) i7-10710U CPU, 1.10 GHz, 16 GB Arbeitsspeicher, Ubuntu 22.04.2 LTS, 64 Bit. Linux Kernel 5.19.0-45-generic. Kompiliert wurde mit GCC 11.3.0 mit der Option `-O2`.
Performance-Tests werden für V0 (die Implementierung mit Vektoroptimierung) und V1 (die Implementierung mit Wrapper-Funktionen für String-Operationen) durchgeführt.

4.1 Laufzeit-Tests

Wir verglichen die Laufzeit für Operationen zwischen Dateien mit unterschiedlichen Datenmengen, unterschiedlicher Entropie (Inhaltswiederholung), Datenlängen, die (nicht) ganzzahlige Vielfache von 16 sind und Dateien mit redundanten Datenlängen von 1 oder 15 usw.

4.1.1 Voraussetzungen für die Laufzeit-Tests

Wir gestalteten unsere Tests so, dass Sie die in der Vorlesung genannten Voraussetzungen [15] erfüllen:

- Um die Genauigkeit der Ergebnisse zu garantieren, wurde jeder Test 250.000 Mal wiederholt, so dass die Gesamtlaufzeit 1 Sekunde betrug.
- Aufgrund der Varianz der Zeittests führen wird derselben Test mindestens dreimal durchgeführt und der Durchschnitt berechnet.
- Die Tests wurden auf einem Personal Computer durchgeführt, da stärkere Varianzen beim Testen in der `lxhalle` über `ssh` beobachtet wurden.
- Der Computer sollte sich nicht im Energiespar-Modus befinden [16], sonst verlängert sich die Laufzeit und die Varianz der Ergebnisse nimmt zu.
- Die Optimierungsstufe sollte mindestens `-O2` sein.
- Es werden `clock_gettime` als funktion und `CLOCK_MONOTONIC` als Uhr gewählt, da sie präziser sind als andere Funktionen.

Die Zeittests erfordern ein sehr hohes Niveau an Genauigkeit und Präzision, [16] andernfalls sind die Ergebnisse gegenstandslos bzw. unwissenschaftlich.

4.1.2 Ergebnisse der zeitbasierten Tests

Der Unterschied in der Laufzeit zwischen $V0$ und $V1$ ist unter `O2` nicht signifikant, testet man allerdings in `O0` ist $V0$ deutlich besser als $V1$. Aufgrund der Voraussetzungen konzentrieren wir uns in diesem Abschnitt jedoch auf die Ergebnisse in `O2`. Auf Grundlage der Ergebnisse der Laufzeittests und 4.1.2 kommen wir zu folgenden Entschlüssen:

(a) **Die Laufzeit ist proportional zur Dateigröße.**
Die verbrauchte Zeit steigt mit steigender Dateigröße linear an.

(b) **Mittels loop-unrolling kann die Laufzeit reduziert werden**[17]
Fügt man die Option `-funroll-loops` in die Makefile ein, kann man einen klaren Speedup erkennen.

(c) **Kein Zusammenhang zwischen SIMD und verbrauchter Zeit**
Im MD2-Hash-Algorithmus umfassen die Berechnung der Prüfsumme und die Generierung des Hash-Wertes mehrere Schritte. Einer dieser Schritte ist die Byte-Substitution, die mehrmals auf dem Eingabe-String durchgeführt wird. Dieser Prozess erfolgt sequenziell und behandelt jedes Byte einzeln. Die Substitution basiert sowohl auf dem aktuellen Byte-Wert, der verarbeitet wird, als auch auf dem Ergebnis, das aus dem vorherigen Substitutionsschritt erzielt wurde und ist somit iterativ in der Natur. Eine Vektorisierung mittels SIMD oder Multithreading erweist sich deswegen als kaum effektiv.

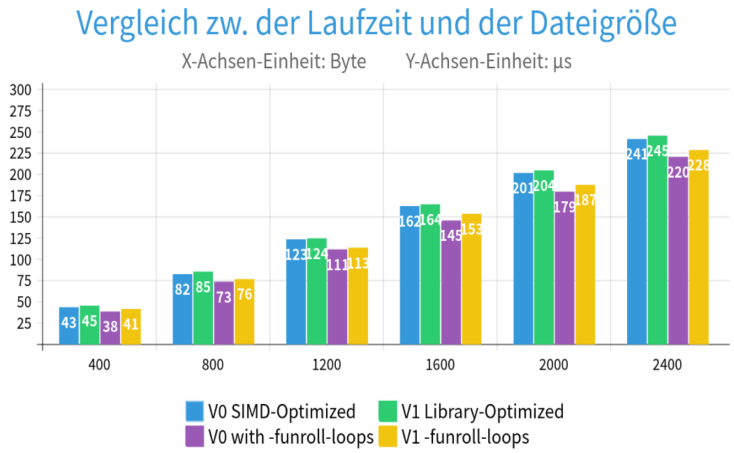

Abbildung 3: Vergleich zwischen Laufzeit und Dateigröße. Die Berechnungen wurden mit Eingabgrößen von 400 bis 2400 Bytes jeweils 250000 mal durchgeführt und das arithmetische Mittel für jede Eingabegröße auf der y Achse eingetragen.

4.2 Tests zur Programmgröße

Bei der Untersuchung Operationenanzahl (Menge des Codes) stellten wir fest, dass $V0$ deutlich weniger Operationen benötigt als $V1$.
Die .c-Dateien zeigen, dass die Implementierung mit Intrinsics die beste ist. Darüber hinaus wurde der Assembly-Code der verschiedenen Implementierungen mit dem Terminalbefehl `objdump -d -M intel main | less` überprüft. [18].

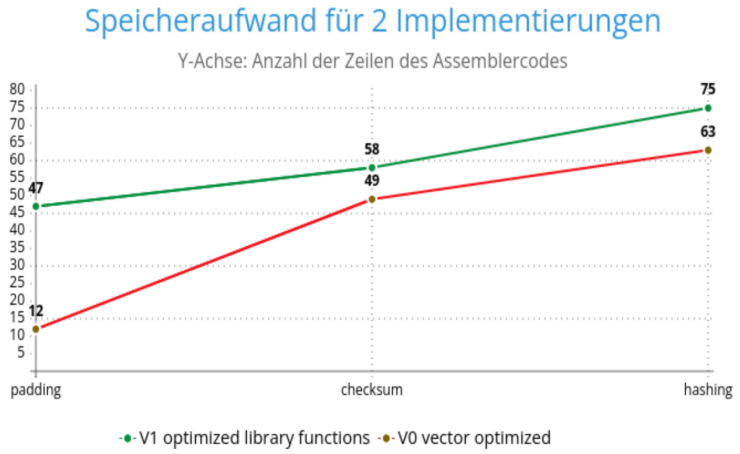

Abbildung 4: Speicheraufwand von $V0$ und $V1$

Der auf V0 basierende Assembly-Code hat viel weniger Vergleichs -und Sprungoperationen. Außerdem wurde in $V0$ die xmm-Erweiterung eingeführt, welche die Anzahl der Instruktionen weiter reduziert (vgl. 4).
Wird die Option `-funroll-loops` verwendet, wird der entstehende Code größer und weniger lesbar (vgl. 5).

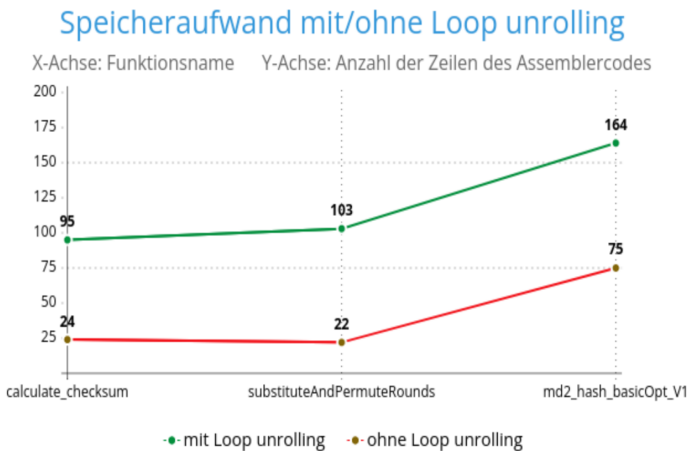

Abbildung 5: Speicheraufwand mit bzw. ohne Loop unrolling

Loop-Unrolling kann die Laufzeit des Programms reduzieren, kommt aber auf kosten von Lesbarkeit und die Speicher.

4.3 Zusammenfassung der Performanzanalyse

Die Implementierung mit Intrinsics ist am effizientesten. Die Verwendung von Loop Unrolling führt zu einem deutlichen Geschwindigkeitszuwachs, die Codegröße steigt damit aber auch an.
MD2 ist ein byteorientierter Hash-Algorithmus.[1] In der Tat verarbeiten alle Instruktionen 8-Bit Blöcke. Dies verbessert zwar die Kompatibilität mit älteren Architekturen und vereinfacht die Implementierung, führt aber auch zu der schlechten Performanz und Optimierbarkeit des MD2-Algorithmus.
Heutige Prozessoren können (mindestens) 32-Bit-Datenwörter verarbeiten, woran sich alle modernen Hashfunktionen orienteren, z.B. MD4, MD5, die Familien RIPEMD und SHA. Daher schränkt die Logik des Algorithmus allein die Optimierungsmöglichkeiten ein.

5 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde der MD2 Hashing-Algorithmus implementiert und auf seine Korrektheit und Performanz auf verschiedenen Limiationen eingegangen. Auf die Sicherheitsprobleme dieses mittlerweile obsoleten Algorithmus von Ronald Rivest wurde ebenfalls eingegangen.
Um dieses Projekt weiterzuführen könnte man weitere Hashfunktionen, die sich etabliert haben, implementieren und in das Rahmenprogramm einbauen. Diese erlauben bessere Optimierungsmöglichkeiten und sind schwerer zu brechen. Insbesondere die Untersuchung der Auswirkung von Quantum Computing auf jene Hashfunktion und Laufzeit könnte ein weiteres Projekt bilden.

Literatur

[1] B. Kaliski. *The MD2 Message-Digest Algorithm*, April 1992. `https://www.rfc-editor.org/info/rfc1319`, visited 2023-06-19.

[2] B. Kaliski. *PKCS7: Cryptographic Message Syntax Version 1.5*, March 1998. `https://www.rfc-editor.org/info/rfc2315`, visited 2023-06-19.

[3] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2018. ISBN 978-0-429-88132-9.

[4] Prof. Claudia Eckert. Vorlesung IT-Sicherheit, WS 22/23, C. Eckert, Kapitel 3: Hashfunktion und MAC. `https://www.moodle.tum.de/pluginfile.php/4271484/mod_resource/content/1/kap3_Hashfunktionen.pdf`, 2022. [Online; accessed 5-Jul-2023].

[5] N. Rogier and Pascal Chauvaud. Md2 is not secure without the checksum byte. *Designs, Codes and Cryptography*, 12(3):245–251, 1997. doi: 10.1023/A:1008220711840. URL https://doi.org/10.1023/A:1008220711840.

[6] Anonymous. Crypto stackexchange answer, 2014. URL https://crypto.stackexchange.com/a/18444. [Online; Accessed: 11-07-2023].

[7] Raj Jain. Md2 checksum - slide 7/18. https://www.cse.wustl.edu/~jain/cse571-09/ftp/l_07hsh.pdf, 2009. Washington University in St. Louis.

[8] Lars Knudsen and John Mathiassen. *Preimage and Collision Attacks on MD2*, 07 2005.

[9] S. Turner and L. Chen. Rfc 6149 - md2 to historic status, 2011. URL https://datatracker.ietf.org/doc/html/rfc6149. [Online; Accessed: 11-07-2023].

[10] Elaine Barker. Recommendation for key management, part 1: General. Technical Report 158, NIST, 2020. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.106.307.

[11] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 617–630, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45146-4.

[12] Wolfram Burgard and Cyrill Stachniss. Einführung in die Informatik, Hashtables. http://ais.informatik.uni-freiburg.de/teaching/ss08/info_MLehrstuhlfürSicherheitinderInformatik,I20DepartmentCE,SchoolCITST/material/mst_14_hashing.pdf, 2008. [Online; accessed 9-Jul-2023].

[13] Online MD2 Hash Calculator. https://www.conversion-tool.com/md2/#uploadProgress, 2023. [Online; accessed 5-Jul-2023].

[14] Prof. Dr. Martin Schulz and Vincent Bode. GRA, Arbeitsblatt 7, T7.1 File IO. https://gra.caps.in.tum.de/material/ERA_Blatt_07_Lsg.pdf, 2023. [Online; accessed 5-Jul-2023].

[15] Prof. Dr. Martin Schulz and Vincent Bode. GRA, Vorlesung 8, v8-0-benchmarking. https://gra.caps.in.tum.de/video/v8-0-benchmarking/23#video-navigation, 2023. [Online; accessed 5-Jul-2023].

[16] Sebastian Gallenmüller, Florian Wiedner, Johannes Naab, and Georg Carle. Ducked Tails: Trimming the Tail Latency of(f) Packet Processing Systems. *17th International Conference on Network and Service Management (CNSM), 2021*, pages 537–543, 2021.

[17] Prof. Dr. Martin Schulz and Vincent Bode. GRA, Vorlesung 6, Optimierungen in GNU C. https://gra.caps.in.tum.de/video/v6-2-opt2/20#video-navigation, 2023. [Online; accessed 5-Jul-2023].

[18] Prof. Dr. Martin Schulz and Vincent Bode. GRA, Arbeitsblatt 2, T2.3 Analyse des kompilierten Programms. https://gra.caps.in.tum.de/material/ERA_Blatt_02_Lsg.pdf, 2023. [Online; accessed 5-Jul-2023].