LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Gruppe 242 – Abgabe zu Aufgabe A505
Sommersemester 2022

Maha Marhag Magdalena Papagianni Alihan Sencan

1 Einleitung

Kryptographie, stammend aus dem altgriechischen *Geheim* und *schreiben*, ist die Wissenschaft der Verschlüsselung von Informationen. Es gibt viele Algorithmen, welche verschlüsselte Kommunikation ermöglichen, aber in dieser Ausarbeitung widmen wir uns dem Message-Digest 2 (MD2) Algorithmus und dessen Eigenschaften, Sicherheitsmerkmale, Angriffsmöglichkeiten und mögliche Alternativen. Im nächsten Kapitel werden wir unsere eigene C Implementierung genauer erläutern und in den weiteren Kapiteln dann auch auf die Korrektheit und die Performanz eingehen.

1.1 Überblick über den Message-Digest 2 Algorithmus

Der MD2 ist eine kryptografische Hashfunktion, die 1989 von Ronald Rivest entwickelt wurde [2]. Eine Hashfunktion ist eine Funktion, die verwendet werden kann, um Nachrichten, Dateien oder Daten (im weiteren Text wird einfachheitshalber nur von Nachrichten gesprochen) beliebiger Größe auf Werte fester Größe abzubilden [6].

Der MD2 Algorithmus wird zum Prüfen von Nachrichten genutzt. Vor allem ist der MD2 Algorithmus für digitalen Signaturen konzipiert worden, in der eine Nachricht auf eine sichere Weise „komprimiert“ werden kann, bevor sie mit einem privaten Schlüssel signiert wird [2]. Der Algorithmus akzeptiert eine Nachricht beliebiger Länge als Eingabe x und erzeugt als Ausgabe einen 16-Byte Message Digest $H(x)$, indem er eine gewisse Redundanz an die Nachricht anhängt und dann iterativ eine Komprimierungsfunktion von 48 Bytes auf 16 Bytes anwendet. [9]. MD2 ist ein Byte-orientierter Algorithmus und damit eine der ersten Hashfunktionen, er unterscheidet sich aber deutlich von moderneren Algorithmen wie MD4, MD5, SHA und SHA-1[5].

1.2 Sicherheitseingeschaften

ryptografische Hashfunktionen sollten folgende drei Sicherheitseigenschaften erfüllen: [12]

- **Urbildresistenz**: Hashfunktionen müssen nur in eine Richtung funktionieren. Bedeutet, dass für eine Hash-Ausgabe z es rechnerisch unmöglich sein muss, eine Eingabenachricht x zu finden, bei der $z = H(x)$ gilt.
- **Zweite Urbildresistenz:** Auch schwache Kollisionsresistenz genannt, besagt, dass zwei verschiedene Nachrichten $x_1 \neq x_2$ nicht den gleichen Hash-Wert haben

dürfen $H(x_1) \neq H(x_2)$. Dabei zeichnet sie sich dadurch aus, dass x_1 gegeben ist und versucht wird x_2 zu finden.

- **Kollisionsresistenz**: Wir nennen eine Hashfunktion kollisionssicher, wenn es rechnerisch nicht möglich ist, zwei verschiedene Eingaben $x_1 \neq x_2$ mit $H(x_1) = H(x_2)$ zu finden. Im Vergleich zur zweiten Urbildresistenz kann in diesem Fall x_1 und x_2 frei gewählt werden.

1.3 Angriffe auf der MD2

Es gibt verschiedene Angriffsarten, die bei der MD2 Hashfunktion erfolgreich durchgeführt wurden. In 2004 wurde ein Erstes-Urbild-Angriff durchgeführt [5], 2008 wurde dieser Angriff weiter verbessert [14] und 2009 folgte der erste Angriff auf Kollisionsresistenz [3]. Folgend werden die drei möglichen Angriffsarten auf Hashfunktionen näher beschrieben und erläutert:

- **Angriff auf Kollisionsresistenz**: Ziel ist es für unterschiedliche Nachrichten $x_1 \neq x_2$ den selben Hashwert zu errechnen $H(x_1) = H(x_2)$. Bei dieser Art von Angriff berechnet der Angreifer via brute-force-Suche den Hashwert $H(x_i)$ beliebiger Nachrichten x_i. Die generierten Hashwerte werden in einer geordneten Liste gespeichert. Der Angriff gilt als erfolgreich, sobald beim Speichern in der Liste ein gleicher Hashwert erkannt wird [12].

- **Urbild-Angriff**: Gegeben sei nur ein Hashwert $H(x_1)$, Ziel ist es eine Nachricht x_2 zu finden, so dass $H(x_2) = H(x_1)$ gilt. Dies kann durch eine brute-force-Suche durchgeführt werden[5].

- **Zweites-Urbild-Angriff**: Gegeben sei eine Nachricht x_1 und dessen Hashwert $H(m_1)$. Ziel dieses Angriffs ist es eine Nachricht $x_2 \neq x_1$ zu finden, so dass $H(x_2) = H(x_1)$ gilt. Der Erste-Urbild-Angriff ist auch gleichzeitig ein Zweiter-Urbild-Angriff, da es trivial ist sicherzustellen, dass eine erhaltene Nachricht von einer gegebenen Nachricht abweicht [3].

1.4 Alternativen

Wie gezeigt wurde ist MD2 keine ausreichend sichere Hashfunktion mehr und aus diesem Grund wurde diese 2011 von der IETF als *"historisch"*, also obsolet eingestuft [15]. Deshalb ist es ratsam andere Alternativen zu verwenden, wie zum Beispiel[7]:

- MD5 - ist eine weit verbreite Hashfunktion welche mittlerweile auch als unsicher gilt

- SHA-1 - eine weitere weit verbreitete Hashfunktion , die ebenfalls nicht mehr als sicher gilt

- SHA-3 - eine relativ neue Hashfunktion , die 2015 standardisiert wurde und im Rahmen eines öffentlichen Wettbewerbs entwickelt wurde

SHA-3 ist der aktuellste und sicherste Standard und es ist deshalb ratsam wenn möglich immer SHA-3 zu verwenden [7].

2 Lösungsansatz

In unserer Implementierung der MD2-Hashfunktion, beginnen wir unter der Annahme, dass wir eine $x-$Byte-Nachricht als Eingabe haben. Wir wollen zunächst ihren Message-Digest finden. Dabei ist x eine beliebige, nicht negative Zahl, also darf diese 0 oder beliebig groß sein. Um den Message-Digest der Nachricht zu berechnen, lesen wir zunächst eine Datei Zeile für Zeile in den Speicher ein. Einen Zeiger auf die Eingabe geben wir dann an eine weitere Funktion weiter, die den Hash berechnet. Dabei verwenden wir Padding, Checksum und Kompression, welche in den nächsten Unterkapiteln genauer erklärt werden.

2.1 Padding

Padding beschreibt das *„Auffüllen“* einer Nachricht mit Fülldaten. Die eingegebene Nachricht wird vor der Verschlüsselung in Blöcke geteilt, die eine gleiche und feste Länge von 16 Byte haben. Wenn die eingegebene Nachricht nicht ein Vielfaches der festen Blocklänge ist, wird diese mit entsprechend vielen Bytes, auch Pad-Bytes genannt, aufgefüllt[11]. Somit wird die Größe der Daten in das akzeptierte und feste Format des gegebenen Algorithmus gebracht [4]. In Abbildung 1 wird die Verteilung und das Padding visualisiert. Padding wird verwendet, nicht nur um die benötigen Bytes aufzufüllen, sondern auch um die Sicherheit der Algorithmen und Protokolle zu erhöhen[1][10].
Im speziellen verwenden wir in unserer Implementierung den `PKCS#7` Standard. Beim `PKCS#7` Padding kann die Blocklänge zwischen 1 und 2^{n-1} (n ist die Anzahl der Bytestellen) liegen. Wenn die Anzahl von n Bytes als Pad-Bytes addieren werden, wird das Byte “‘n”’ n-mal an den Block konkateniert, wie in Abbildung 2 zu sehen ist. n kann zwischen 1 und der Blocklänge liegen[10]. Um die Bytes aufzufüllen nutzen wir die Funktion `memset`. Als konkretes Beispiel in Abbildung 2 benötigen wir ein Padding von fünf Byte und füllen deshalb den Nachrichtenblock mit fünf mal der 5 auf. Würden wir nur ein Padding von vier Byte benötigen, würden wir dann analog mit 4 auffüllen.

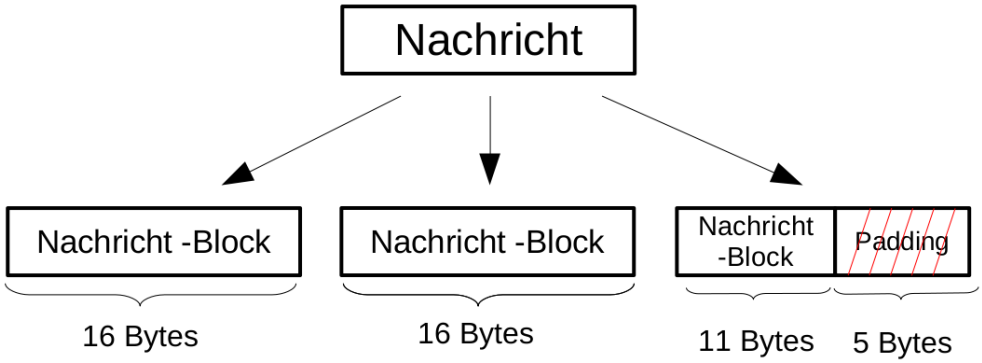

Abbildung 1: Block-Verteilung

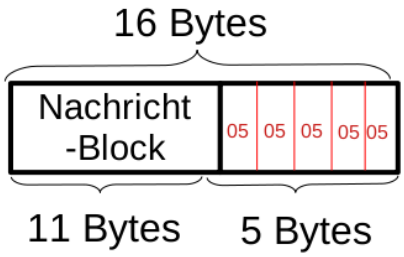

Abbildung 2: Pad-bytes

2.2 Checksum

Eine Prüfsumme (eng.: Checksum) ist definiert als: *„Eine Zahl, die sich aus der Addition aller Zahlen eines elektronischen Datensatzes ergibt, um die Richtigkeit der Daten zu überprüfen“*[8]. Im MD2-Hash-Algorithmus ist die Prüfsumme ein 16-Byte-Block, der an die durch Padding angereicherte Nachricht, angehängt wird. Für das Anhängen der Prüfsumme an das Ende der Nachricht wird eine S-Box verwendet [2]. Die S-Box ist eine 256-Byte-Zufalls-Permutationstabelle, welche ganze Zahlen von 0 bis 255 enthält. Unter Verwendung einer Variante des Durstenfeld-Algorithmus mit einem Pseudozufallszahlengenerator auf der Grundlage der Dezimalziffern von π werden diese gemischt und eine zufällige Permutation auf die Folge $a[i]$ mit $i = 1, 2, ..., 256$ angewendet. Der Inhalt der S-Box in hexadezimal wird in Abbildung 3 gezeigt [2].

```
{0x29, 0x2E, 0x43, 0xC9, 0xA2, 0xD8, 0x7C, 0x01, 0x3D, 0x36, 0x54, 0xA1, 0xEC, 0xF0, 0x06, 0x13,
  0x62, 0xA7, 0x05, 0xF3, 0xC0, 0xC7, 0x73, 0x8C, 0x98, 0x93, 0x2B, 0xD9, 0xBC, 0x4C, 0x82, 0xCA,
  0x1E, 0x9B, 0x57, 0x3C, 0xFD, 0xD4, 0xE0, 0x16, 0x67, 0x42, 0x6F, 0x18, 0x8A, 0x17, 0xE5, 0x12,
  0xBE, 0x4E, 0xC4, 0xD6, 0xDA, 0x9E, 0xDE, 0x49, 0xA0, 0xFB, 0xF5, 0x8E, 0xBB, 0x2F, 0xEE, 0x7A,
  0xA9, 0x68, 0x79, 0x91, 0x15, 0xB2, 0x07, 0x3F, 0x94, 0xC2, 0x10, 0x89, 0x0B, 0x22, 0x5F, 0x21,
  0x80, 0x7F, 0x5D, 0x9A, 0x5A, 0x90, 0x32, 0x27, 0x35, 0x3E, 0xCC, 0xE7, 0xBF, 0xF7, 0x97, 0x03,
  0xFF, 0x19, 0x30, 0xB3, 0x48, 0xA5, 0xB5, 0xD1, 0xD7, 0x5E, 0x92, 0x2A, 0xAC, 0x56, 0xAA, 0xC6,
  0x4F, 0xB8, 0x38, 0xD2, 0x96, 0xA4, 0x7D, 0xB6, 0x76, 0xFC, 0x6B, 0xE2, 0x9C, 0x74, 0x04, 0xF1,
  0x45, 0x9D, 0x70, 0x59, 0x64, 0x71, 0x87, 0x20, 0x86, 0x5B, 0xCF, 0x65, 0xE6, 0x2D, 0xA8, 0x02,
  0x1B, 0x60, 0x25, 0xAD, 0xAE, 0xB0, 0xB9, 0xF6, 0x1C, 0x46, 0x61, 0x69, 0x34, 0x40, 0x7E, 0x0F,
  0x55, 0x47, 0xA3, 0x23, 0xDD, 0x51, 0xAF, 0x3A, 0xC3, 0x5C, 0xF9, 0xCE, 0xBA, 0xC5, 0xEA, 0x26,
  0x2C, 0x53, 0x0D, 0x6E, 0x85, 0x28, 0x84, 0x09, 0xD3, 0xDF, 0xCD, 0xF4, 0x41, 0x81, 0x4D, 0x52,
  0x6A, 0xDC, 0x37, 0xC8, 0x6C, 0xC1, 0xAB, 0xFA, 0x24, 0xE1, 0x7B, 0x08, 0x0C, 0xBD, 0xB1, 0x4A,
  0x78, 0x88, 0x95, 0x8B, 0xE3, 0x63, 0xE8, 0x6D, 0xE9, 0xCB, 0xD5, 0xFE, 0x3B, 0x00, 0x1D, 0x39,
  0xF2, 0xEF, 0xB7, 0x0E, 0x66, 0x58, 0xD0, 0xE4, 0xA6, 0x77, 0x72, 0xF8, 0xEB, 0x75, 0x4B, 0x0A,
  0x31, 0x44, 0x50, 0xB4, 0x8F, 0xED, 0x1F, 0x1A, 0xDB, 0x99, 0x8D, 0x33, 0x9F, 0x11, 0x83, 0x14}
```

Abbildung 3: S-Box

Zur Berechnung der Prüfsumme, wie in Abbildung 4 zu sehen ist, verwenden wir ein mit Null initialisiertes 16-Byte-Array (Buffer) und mehrfache `XOR`-Operationen mit der Prüfsumme (checksum) und der S-Box.

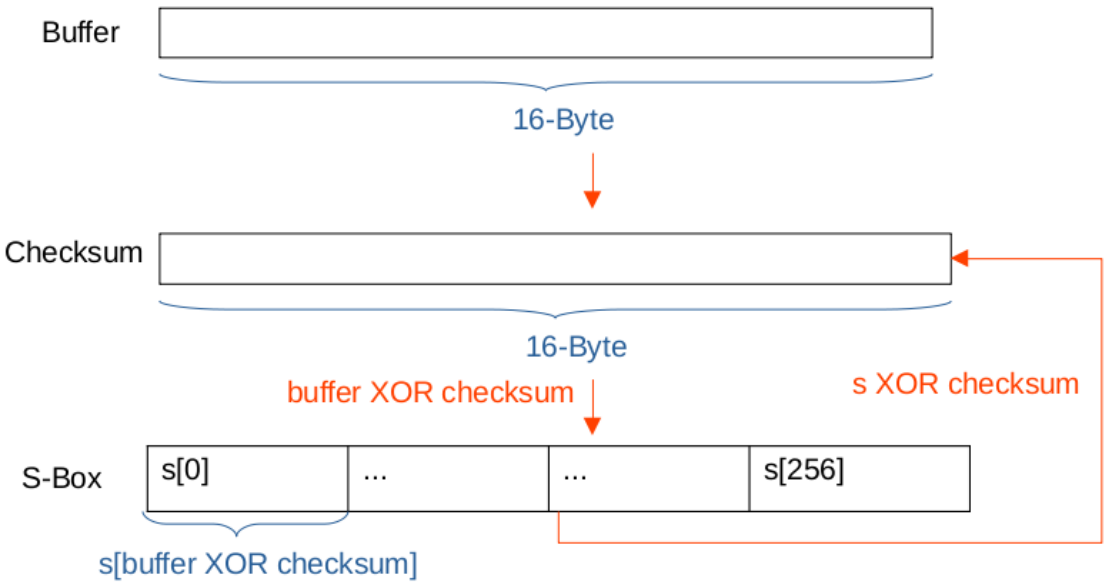

Abbildung 4: Berechnung der Prüfsumme

2.3 Kompressionsfunktion

Um die Hash-Operationen durchzuführen verwenden wir ein anfangs leeres Output-Array mit einer Größe von 48-Byte, also insgesamt drei mal 16-Byte Blöcke. Der 16-Byte große Input-Buffer wird als Block in den zweiten Block des Output-Arrays kopiert. Anschließend wird mit Hilfe der 256-Byte großen S-Box (siehe Abbildung 3) das Ergebnis

mittels `XOR`-Operationen berechnet. Eine Kompression kommt zustande, weil sich das Ergebnis nur auf der ersten Block des Output Arrays beschränkt. Die Kompressionsfunktion wird auch in Abbildung 5 dargestellt.

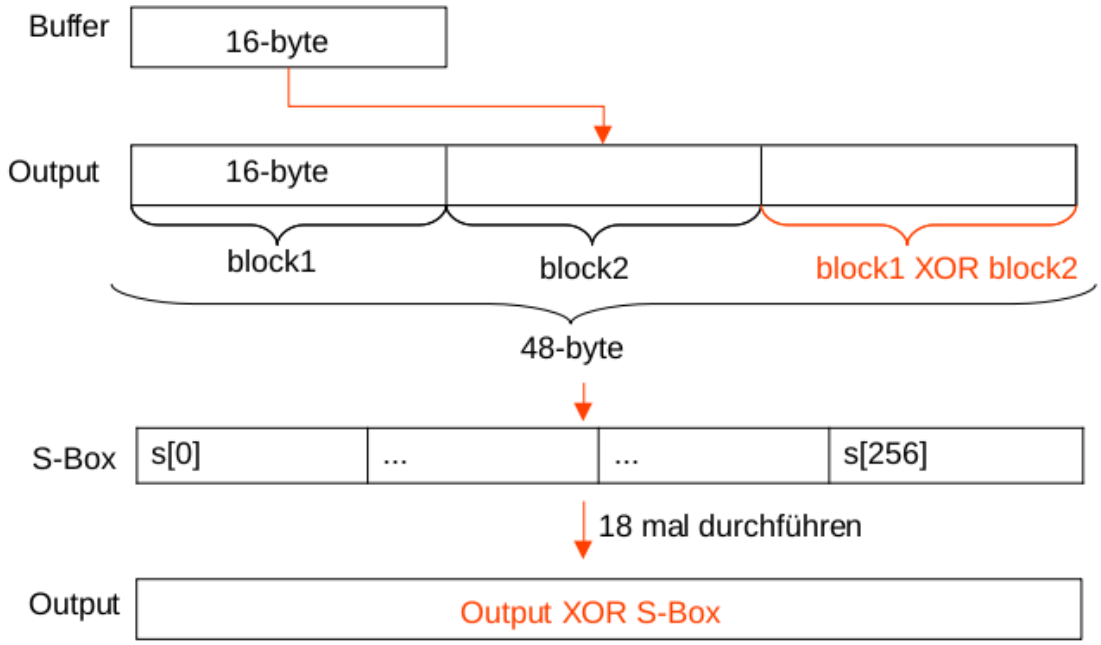

Abbildung 5: Kompressionsfunktion

2.4 Lösungsalternative

Als alternative Implementierung (`V1`) haben wir eine Lösung mit `Structs` gewählt. Dies erlaubt Daten unterschiedlicher Typen zu kombinieren, vereinfacht den Aufbau des Quelltexts und erhöht somit die Lesbarkeit und Wartbarkeit des Programms. Diesen Ansatz haben wir nicht als primäre Implementierung gewählt, da wir die vorgegebene Signatur ändern mussten.

2.5 Optimierungen

Für die Berechnung eines Hashwertes sind viele Speicherzugriffe erforderlich, da nicht nur die Nachricht beziehungsweise Eingabedatei verarbeitet werden muss, sondern auch fast jeder Zwischenschritt Daten im Speicher verarbeitet. Diese sind die laufzeitintensivsten Operationen in unserer Implementierung und somit der ideale Kandidat, um diese genauer zu Untersuchen und mögliche Optimierungsansätze zu evaluieren und umzusetzen. Folgende konkrete Optimierungen haben wir schließlich implementiert:

- Pad-Bytes werden nicht mehr in den Buffer kopiert und iterativ in einer While-Schleife angehangen, sondern mit der Bibliotheksfunktion `MEMSET` effizient in

einem Durchlauf geschrieben

- Die iterative Initialisierung der Arrays mit Null, welche für die Berechnung des Hash verwendet wird, wurde auch durch die `MEMSET`-Funktion ersetzt
- Effizientere vektorisierte `SIMD-XOR`-Operationen werden eingesetzt, anstatt sequenziell und wiederholt `XOR`-Operationen aufzurufen

Andere Optimierungen haben wir nicht vorgenommen, da zum Beispiel in `IO.c` keine Iterationen und Speicherallokationen erfolgen und wir `Loop-Carried-Dependencies` in den Funktionen `Checksum` und `Transform` haben. Weitere Optimierungen wären jedoch möglich gewesen, diese behandeln wir aber näher in Abschnitt 5.

3 Korrektheit

Zur Überprüfung der Korrektheit unserer Ergebnisse haben wir eine Reihe von Tests erstellt. Diese Tests behandelt Eingabedateien verschiedenster Größen und auch einige Randfälle - wie in der untenstehenden Tabelle kurz aufgeführt. Natürlich hat die Größe der Eingabe und der Typ der Zeichen keinen Einfluss auf das Ergebnis, da 16-Byte ausgerichtete Speicherzugriffe (input, output und checksum Arrays) eine 16-Byte Ausgabe garantieren. Neben den Beispielen in der Test Suite des RFCs [2] haben wir auch mehrere Tests mit Hilfe eines Online MD2-Hashgenerators [13] erstellt und verifiziert, um eine breiterer Reichweite von Testfällen abzudecken.

Eingabe	**Hash**
leere Datei	`8350e5a3e24c153df2275c9f80692773`
`.... . .-.. .-.. -- / .- -- .-. .-.. -.. / -.-.-`	`c6dc54d633087cbe5046d25129e1467c`
`abc`	`da853b0d3f88d99b30283a69e6ded6bb`
`ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789`	`da33def2a42df13975352846c30338cd`

4 Performanzanalyse

In diesem Kapitel möchte wir die Perfomanzunterschiede unsere Implementierung anhand zweier Umsetzungsversionen erläutern. Dabei handelt es sich um die *optimierte* und die *nicht-optimierte* Version. Die vorgenommenen Optimierungen haben wir bereits in Unterabschnitt 2.5 näher erläutert. Die Laufzeitvergleiche für die Leistungsanalyse wurden gemäß Abschnitt „Benchmarking“ des Praktikums durchgeführt. Wie in den Folien angegeben, wurde `MONOTONIC Clock` verwendet, um die Zeit für die zwei Versionen zu messen. Die optimierte Version hatte nur eine 6-mal schnellere Durchlaufzeit für Eingaben mit weniger als 500 Zeichen. Dies ist dem Umstand geschuldet, dass zusätzliche `SIMD`-Befehle, die zum Laden und Speichern von Daten verwendet werden, die Laufzeit beeinflussen. Die zusätzlichen Befehle, die für die Speicherregulation nötig sind, verlangsamt die Geschwindigkeit des Programms für kleine Eingaben, da der Overhead in Relation zur Laufzeut mehr ins Gewicht fällt, aber sind sehr vorteilhaft bei größeren Dateneingaben. Mit wachsender Eingabegröße ist die optimierte Version

aber deutlich performanter. Die Zeiten der optimierten Version sind bei Eingaben im fünfstelligen Bereich 20- bis 30-mal schneller als die nicht-optimierte Version und sogar über 70-mal so schnell bei 10.000 Zeichen. In Abbildung 6 sieht man den Vergleich der beiden Implementierungen. Die Zeiten basieren auf mehreren Durchläufen und sind dann anschließen arithmetisch gemittelt worden.

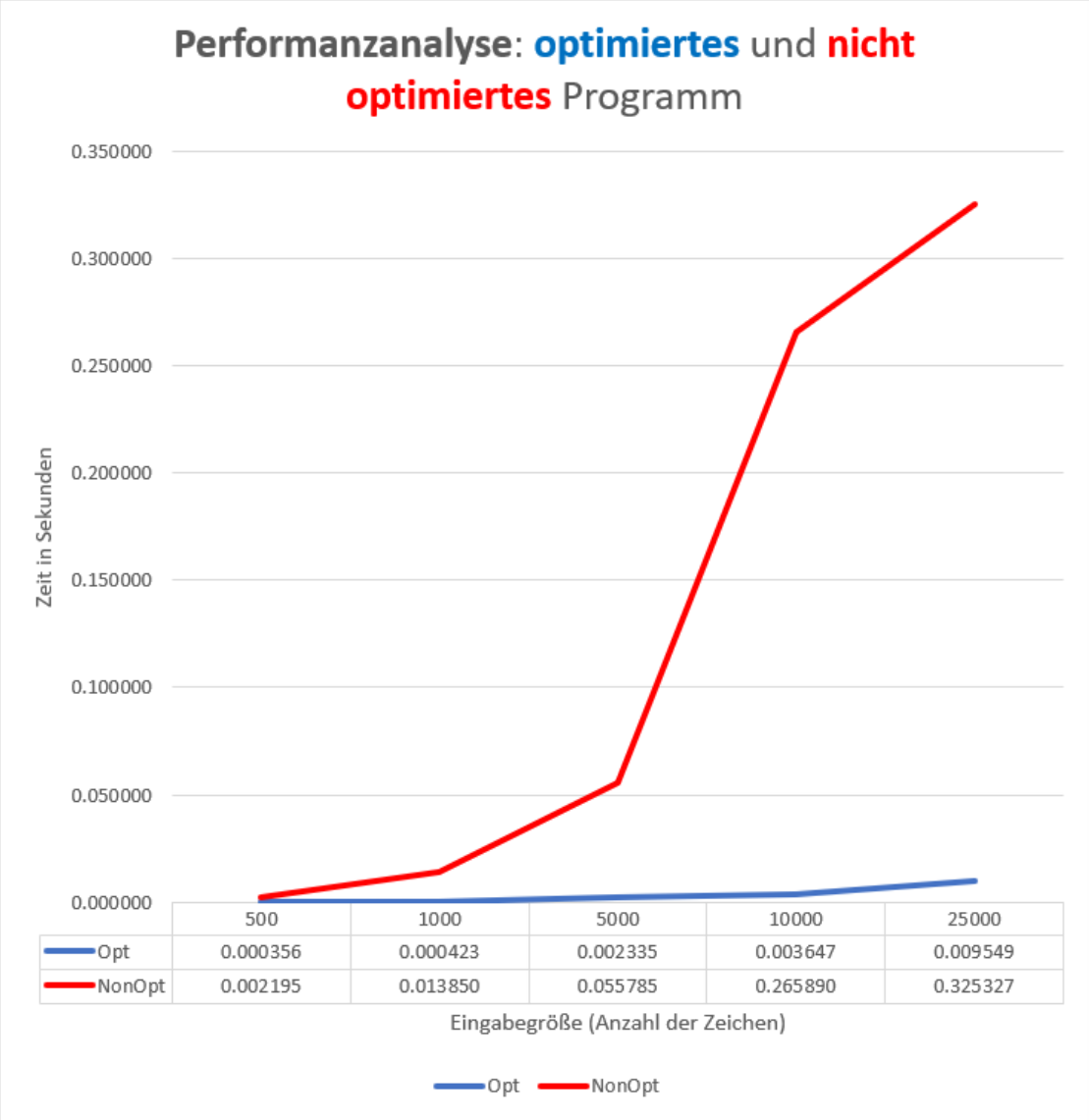

Abbildung 6: Performanzanalyse

5 Zusammenfassung und Ausblick

Unser Ziel war es selbständig die MD2-Hashfunktion in C zu Implementieren. Hashfunktionen werden heutzutage in vielen Bereichen verwenden und stellen einen essentiellen Bestandteil dar [5]. Diese verwenden Padding, Checksums und S-Boxen, um aus Daten beliebiger Größe einen Hashwert fester Länger zu generieren. Obwohl der MD2-Algorithmus heute nicht mehr als sicher gilt und seine Verwendung abnimmt, war es dennoch eine interessante Erfahrung diesen selbst zu implementieren. Wir haben einen tiefen Einblick in den Aufbau und die Einzelteile der Hashfunktion erhalten können und konnten auch selbst testen wie stark Optimierungen sich auf die Laufzeit auswirken. Hätten wir mehr Zeit gehabt würden wir vermehrt `SIMD`-Operationen verwenden - nicht nur für die `XOR`-Operationen. Dies hätte den Vorteil, dass mehr Vorgehen parallel ausgeführt werden können und beim Padding wir auch 16-Byte gerichtete Eingaben verwenden könnten. Hätte aber den Nachteil gehabt, dass wir in der Hashfunktion einen noch höheren Speicheroverhead hätten.

Literatur

[1] Bundesamt für Sicherheit in der Informationstechnik. Kryptographische verfahren: Empfehlungen und schlüsselängen. *BSI TR-02102*, 2022.

[2] Burt Kaliski. The md2 message-digest algorithm. RFC 1319, April 1992.

[3] Lars R. Knudsen, John Erik Mathiassen, Frédéric Muller, and Søren S. Thomsen. Cryptanalysis of md2. *Journal of Cryptology*, 23(1):72–90, Jan 2010.

[4] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.

[5] Frederic Muller. The md2 hash function is not one-way. In Pil Joong Lee, editor, *Advances in Cryptology - ASIACRYPT 2004*, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[6] Schueffel Patrick, Groeneweg Nikolaj, and Baldegger Rico. *The Crypto Encyclopdia - Coins, tokens and digital assets from A to Z*. Growth Publisher, Bern, 2019.

[7] Jan Pelzl and Christof Paar. *Kryptografie verständlich: Ein Lehrbuch für Studierende und Anwender*, chapter 11, pages 293–307. Springer Berlin Heidelberg, 2016.

[8] Rationality. *The Cambridge Dictionary of Philosophy*. Cambridge University Press, 1999.

[9] N. Rogier and Pascal Chauvaud. Md2 is not secure without the checksum byte. *Designs, Codes and Cryptography*, 12, Nov 1997.

[10] K. Schmeh. *Kryptografie – Verfahren, Protokolle, Infrastrukturen*, pages 199–234. dPunkt Verlag, Heidelberg, 2007.

[11] Jörg Schwenk. *Sicherheit und Kryptographie im Internet. Von sicherer E-Mail bis zu IP-Verschlüsselung*. Vieweg, 2005.

[12] Stephan Spitz, Michael Pramateftakis, and Joachim Swoboda. *Kryptographie und IT-Sicherheit: Grundlagen und Anwendungen*, chapter 3, pages 95–100. Vieweg+Teubner, 2011.

[13] Björn Staven. Online generator zum md2 hash-code erstellen. `https://hash-generator.io/online-md2-hash-generator.php#hash_footer_link`.

[14] Søren S. Thomsen. An improved preimage attack on md2, 2008. crypto@znoren.dk 13937 received 28 Feb 2008.

[15] S. Turner. Md2 to historic status. RFC 6149, March 2011.