Entwicklung einer Benchmark Software für kryptographische Hashfunktionen

Marvin Schlüchter

FH-Dortmund
marvin.schluechter@rub.de

Zusammenfassung

Kryptographische Hashfunktionen sind ein integraler Bestandteil der modernen IT-Sicherheit. Hashfunktionen ermöglichen es u.a. die Integrität von Dateien und Nachrichten sicherzustellen und sind des Weiteren direkt an der Umsetzung von Authentizität durch die sog. Digitale Signatur beteiligt. Wesentlich bei der Frage, welcher der verschiedenen Algorithmen einzusetzen ist, ist neben der Frage der Anforderungen an das Soft- bzw. Hardwaresystem, sowie die Herkunft des Algorithmus, die Laufzeit. Oftmals stehen Studierende vor dem Problem, dass eine akkurate Einschätzung zeitlicher Größenordnungen für kryptographische Hashfunktionen fehlt. Dieses Paper zeigt die Entwicklung einer Benchmark Software für kryptographische Hashfunktionen. Dabei werden die wichtigsten Entwicklungsschritte aufgezeigt und dargestellt, welche grundsätzlichen Probleme sich bei dem Entwurf und der Umsetzung einer solchen Software ergeben. Darüber hinaus wird die auf diese Weise entstandene Software „CryptBench" vorgestellt. CryptBench operiert auf der Kommandozeile und ist in der Lage verschiedene grundlegende Funktionen im Umgang mit kryptographischen Hashfunktionen umzusetzen. Dazu gehört das „einfache" hashen von Nachrichten oder Dateien und der Ausgabe des Hashwertes auf der Kommandozeile. Aber auch umfangreiche Benchmarks können erstellt werden und die zugehörigen Ergebnisse können sowohl auf der Kommandozeile ausgegeben, als auch in eine Datei gespeichert werden. Die von dieser Software unterstützten Hashfunktionen sind: MD5, SHA1, SHA2, SHA3, Whirlpool und RIPEMD-160. CryptBench richtet sich sowohl an private Nutzer als auch an Dozenten, die im Bereich der Informationssicherheit lehren und forschen. Besonderes Potential entfaltet CryptBench dann, wenn es um die Lehre der softwaretechnischen Herausforderungen mit einer möglichst fehlertoleranten Implementierung von kryptographischen Primitiven (Hashfunktionen) geht.

1 Einleitung

Grundproblem bei der Entwicklung von kryptographischen Hashfunktionen ist die Abbildung von Eingaben beliebiger Länge auf eine Ausgabe mit fixierter Länge. Hierzu werden gegenwärtig hauptsächlich zwei Verfahren eingesetzt:

- Merkle-Damgård-Konstruktion
- Sponge-Konstruktion

Beiden Verfahren liegt die gleiche Idee zugrunde: Es werden Funktionen entwickelt, die Eingaben fester Länge auf Eingaben fester Länge abbilden. Im zweiten Schritt werden diese Funktionen dann verkettet (Details siehe [Nist15] und [Merk79]). Darüber hinaus werden Hashfunktionen in zwei Kategorien eingeteilt. „keyed hash function" und „unkeyed hash functions"

P. Schartner · A. Baumann (Hrsg.) · D•A•CH Security 2017 · syssec (2017) 149-158.

[MeOV96], wobei in dieser Ausarbeitung nur die unkeyed Hashfunktionen behandelt werden. Im Folgenden werden die innerhalb dieser Ausarbeitung Anwendung findenden kryptographischen Hashfunktionen kurz vorgestellt. Um die verschiedenen Hashfunktionen qualitativ vergleichen zu können werden folgende Kriterien herangezogen: *Abstammung, Blockgröße, Hashwertgröße, Anzahl der Runden sowie die verwendeten Operationen*.

1.1 MD5

MD5 [Rive92a] stellt eine Weiterentwicklung der kryptographischen Hashfunktion MD4 [Rive92b] dar. Beide Verfahren wurden von Ronald Rivest entwickelt und vorgestellt. Dieser Algorithmus spielt eine zentrale Rolle für die Entwicklung von Hashfunktionen, weil er als Grundlage für zahlreiche andere Verfahren dient. Von der Benutzung wird weitgehend abgeraten, da zahlreiche Angriffe auf dieses Verfahren bekannt sind. MD4 und MD5 bauen direkt auf der Merkle-Damgård-Konstruktion auf. MD5 ist eine kryptographische Hashfunktion, die Nachrichten beliebiger Länge auf einen 128 Bit langen Hashwert abbildet. Der Algorithmus arbeitet in vier Runden mit jeweils 16 Teilrunden. Die vorgeschlagene Zielarchitektur für dieses Verfahren ist eine 32-Bit CPU.

1.2 SHA-1

SHA-1 entstand, während die amerikanische Regierung, vertreten durch das NIST, den Standard DSS (Digital Signature Standard) entwickelt hat. DSS sollte den neuen Signatur Algorithmus DSA (Digital Signature Algorithm) spezifizieren und benötigte dafür eine möglichst sichere kryptographische Hashfunktion. Herausgekommen ist dabei der Algorithmus SHA-1 [EsJo01], welcher auf MD4 bzw. MD5 aufbaut, aber u. a. eine größere Sicherheit bieten sollte. SHA-1 verarbeitet Nachrichten beliebiger Länge und bildet diese auf einen 160 Bit langen Hashwert ab. Dabei arbeitet der Algorithmus in vier Runden mit jeweils 20 Teilrunden.

1.3 SHA-2

Wie auch SHA-1, wurde SHA-2 [Nist12] von der NIST im Auftrag der US-Regierung entwickelt. Der Algorithmus sollte die entdeckten Angriffsmöglichkeiten auf SHA-1 beseitigen und ist auch heute noch die allgemeine Empfehlung des NIST für eine sichere kryptographische Hashfunktion (vgl. [Dang12], S. 9). Anders jedoch als sein direkter Vorgänger, bezeichnet SHA-2 keine direkt ausführbare Hashfunktion, sondern eine Klasse von Hashfunktionen. SHA-2 setzt sich aus den vier Algorithmen SHA-224, SHA-256, SHA-384 und SHA-512 zusammen, die sich in ihrer Hashwertgröße und auch in ihrem Ablauf unterscheiden. Dabei entspricht die Zahl am Ende des Namens der Größe des erzeugten Hashwertes. Die vier Hashfunktionen lassen sich in zwei Gruppen einordnen: Einmal die Hashfunktionen, die mit einer Blockgröße von 512 Bit arbeiten und auf der anderen Seite die Hashfunktionen, die mit einer Blockgröße von 1024 Bit arbeiten (siehe Abbildung 1).

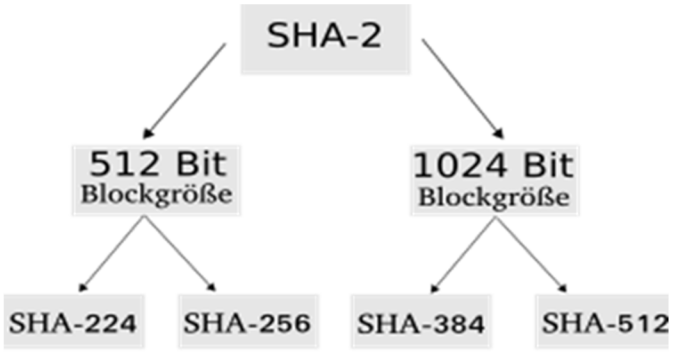

Abb. 1: Algorithmen der Klasse SHA-2

1.4 RIPEMD-160

Das in Europa entwickelte Verfahren RIPEMD-160 [DoBP96] (RACE Integrity Primitives Evaluation Message Digest) basiert auf dem weniger bekannten Verfahren RIPEMD, welches durch die Entwickler Hans Dobbertin, Antoon Bosselaers und Bart Preneel entschieden verändert und verbessert wurde. Es wurde 1996 veröffentlicht und steht in direkter Konkurrenz zu dem von der NIST entwickelten Algorithmus SHA-1. RIPEMD-160 nimmt eine Eingabe beliebiger Länge entgegen und bildet diese auf einen 160 Bit langen Hashwert ab. Das Verfahren basiert auf MD4 und arbeitet in 5 Runden mit jeweils 16 inneren Runden.

1.5 SHA-3

Durch die Angriffe auf SHA-1 und MD5 suchte das NIST nach einer kryptographischen Hashfunktion, welche ein gänzlich neues Verfahren etabliert. Denn viele der einschlägigen Experten waren besorgt, dass in Zukunft auch die bis dato als sicher geltenden Hashfunktionen der Klasse SHA-2 ähnlichen Angriffen unterliegen würden. Begründet sahen die Experten ihre Annahme darin, dass SHA-2, genau wie SHA-1 und MD5, auf der Merkle-Damgård-Konstruktion aufbauen, welche als potenzieller Angriffsvektor gesehen wurde. Das NIST entschied sich daraufhin einen Wettbewerb abzuhalten, dessen Gewinner zum neuen SHA-3 Standard gekürt werden sollte (vgl. [Chen06]). Am 02.10.2012 wurde daraufhin der Algorithmus Keccak zum Gewinner erklärt. Dieser Algorithmus setzt auf die sog. „Sponge-Konstruktion“, welche eine gänzlich andere Vorgehensweise zur Erstellung von Hashfunktionen vorsieht, als die Merkle-Damgård-Konstruktion. Keccak ist somit die zweite „sichere“ kryptographische Hashfunktion, die vom NIST, neben SHA-2 empfohlen wird. Auf Basis von KECCAK hat das NIST vier kryptographische Hashfunktionen als SHA-3 Familie standardisiert. Die folgenden Definitionen der Funktionen wurden direkt dem FIPS 202 [Nist15] entnommen:

$$\text{SHA3-224}(M) = \text{KECCAK}[448](M \,||\, 01, 224)$$

$$\text{SHA3-256}(M) = \text{KECCAK}[512](M \,||\, 01, 256)$$

$$\text{SHA3-384}(M) = \text{KECCAK}[768](M \,||\, 01, 384)$$

$$\text{SHA3-512}(M) = \text{KECCAK}[1024](M \,||\, 01, 512)$$

1.6 Whirlpool

Dieses Verfahren wurde im Jahre 2000 von den Entwicklern Vincent Rijmen und Paulo S. l. M. Barreto vorgestellt [BaRi03]. Von Whirlpool gibt es drei Versionen:

1. WHIRLPOOL-0
2. WHIRLPOOL-T
3. WHIRLPOOL

Whirlpool-0 ist die erste vorgestellte Variante des Verfahrens, welche später durch die verbesserte Version Whirlpool-T abgelöst wurde. Nachdem jedoch eine Schwäche, in der bis dato aktuellen Version festgestellt wurde, wurde eine finale Version von Whirlpool veröffentlicht, welche die Schwäche behoben hat. Whirlpool bildet eine Nachricht beliebiger Länge, die echt kleiner als 2^{256} Bit ist, innerhalb von zehn Runden auf einen 512 Bit langen Hashwert ab. Im Mittelpunkt dieser Hashfunktion steht die Merkle-Damgård-Konstruktion, die als Kompressionsfunktion die Blockchiffre W nutzt.

2 Softwareanforderungen

Es soll eine Software zum Benchmarking von kryptographischen Hashfunktionen entstehen. Benchmarking beschreibt innerhalb dieser Ausarbeitung, das Messen und Vergleichen der Ausführungsgeschwindigkeiten von Algorithmen auf einem vorher festgelegten Soft- und Hardwaresystem. Der Benutzer soll das Programm starten und festlegen, welche Hashfunktionen er mit welchen Eingaben ausführen und vergleichen möchte. Hat der Benutzer sich für eine Menge an Hashfunktionen entschieden, soll das Programm eine Liste an Ergebnissen liefern. Darüber hinaus werden für das Projekt die folgenden Anforderungen festgelegt:

- Die CPU-Architektur (Central Processing Unit) auf der die Software laufen muss, ist x86 bzw. x86_64. Weitere Architekturen müssen nicht unterstützt werden.
- Alle kryptographischen Hashfunktionen müssen exakt nach dem entsprechenden Standard entwickelt werden (RFC (Request for Comments) oder FIPS (Federal Information Processing Standards)).
- Die Software muss auf einem Linux-Betriebssystem laufen. Die Software muss nicht plattformunabhängig sein. Die Software soll auf der aktuellen Mac OS X Version laufen.
- Die kryptographischen Hashfunktionen sollen möglichst performant bzgl. der Laufzeitgeschwindigkeit und des Speicherverbrauchs implementiert werden.

Um die Anforderungen erfolgreich umsetzten zu können, müssen u. a. diese grundlegenden Probleme gelöst werden.

- Ermöglichen einer möglichst genauen Zeitmessung
- Ermöglichen einer erweiterbaren Softwarearchitektur bei hoher Laufzeitgeschwindigkeit und Hardwarenähe

Darüber hinaus müssen die folgenden Probleme im Zusammenhang mit kryptographischen Hashfunktionen gelöst werden.

- Praktische Umsetzung kryptographischer Hashfunktionen auf modernen Computersystemen
- Verständlicher und leicht zugänglicher Quelltext bei hoher Laufzeitgeschwindigkeit und Hardwarenähe

3 Bestehende Softwarelösungen

Gegenwärtig befinden sich einige Softwarelösungen und Produkte auf dem Markt, die eine Implementierung der hier betrachteten Hashfunktionen umsetzten. Hierzu gehören beispielsweise OpenSSL[1], GnuTLS[2] und Crypto++[3]. OpenSSL ist eine freie Implementierung des SSL- bzw. TLS-Protokolls und stellt darüber hinaus auch eine Softwarebibliothek für kryptographische Verfahren bereit. GnuTLS ist ebenfalls eine freie Implementierung des SSL- und TLS-Protokolls. Auch diese Software stellt eine Softwarebibliothek für kryptographische Verfahren zur Verfügung. Crypto++ ist eine Softwarebibliothek, die eine Vielzahl an kryptographischen Primitiven anbietet. Diese Softwarelösungen eignen sich jedoch aufgrund des hohen Komplexitätsgrades der Implementierungen nicht für den vorliegenden Anwendungsfall. Da bei allen vorgestellten Softwareumsetzungen stets mehrere Entwickler an den Umsetzungen der Hashfunktionen beteiligt waren, werden Zusammenhänge und Abstammungen der einzelnen Hashfunktionen nicht direkt erkenntlich. Darüber hinaus setzten die aufgezeigten Implementierungen an vielen Stellen auf sogenannten „integrierten Assemblercode“ (vgl. z.B. [Dai09]), welcher den Einarbeitungsaufwand seitens der Studierenden stark erhöht. Aus diesem Grund ist eine reine C/C++ Implementierung vorzuziehen, auch wenn dies zu mitunter starken Performanceeinbußen führen kann. Aufgrund dieser Umstände wurde also eine Eigenimplementierung angestrebt, welche ganzheitlich in einer höheren Programmiersprache, hier C++, entwickelt werden sollte.

4 Implementierung der Software

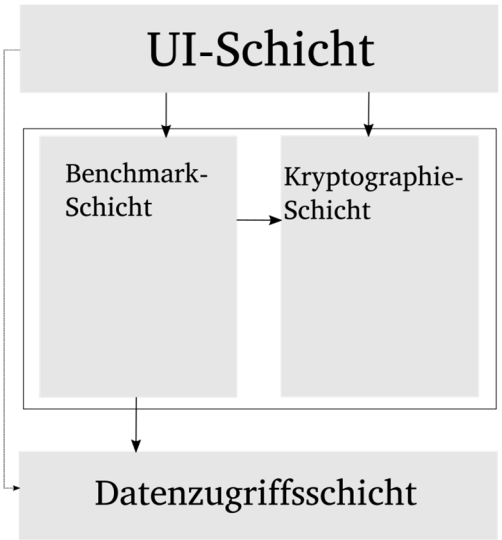

Abb. 2: Softwarearchitektur von CryptBench

Der grundlegende Aufbau der Softwarearchitektur ist an der Drei-Schichten-Architektur angelehnt, allerdings mit einigen Änderungen im Sinne einer bestmöglichen Anpassung an dieses Softwareprojekt. Die klassische Drei-Schichten-Architektur besteht aus der GUI-, Fachlogik-

[1] https://www.openssl.org/

[2] http://www.gnutls.org

[3] https://www.cryptopp.com/

sowie der Datenzugriffsschicht und ist eine Erweiterung bzw. Spezifizierung der allgemeinen Schichtenarchitektur (vgl. [VAC+09], S. 217f). Diese Aufteilung erlaubt eine lose Kopplung einzelner Komponenten und Subsysteme der Software und ermöglicht dadurch eine einfache Wart- und Erweiterbarkeit. Die Fachlogikschicht wurde angepasst und wurde in diesem Fall in zwei weitere Schichten unterteilt: Benchmark-Schicht und Kryptographie-Schicht. Der Benchmark-Schicht werden alle Funktionen zugeordnet, die sich mit der Zeitmessung der kryptographischen Hashfunktionen beschäftigen, während die Kryptographie-Schicht die kryptographischen Hashfunktionen selbst umfasst. Ein Vorteil dieser Aufteilung ist, dass fehleranfällige Low-level-Operationen, die zur Umsetzung von kryptographischen Funktionen benötigt werden, in nur eine logische Komponente gekapselt werden können. Diese Kapselung erleichtert die Implementierung der Software und erhöht damit die Fehlerrobustheit der resultierenden Anwendung.

CryptBench ist ausschließlich für die x86- bzw. x86_64-Architektur konzipiert. Dieser Umstand ermöglicht eine wesentliche Vereinfachung der Implementierung der kryptographischen Komponenten dieser Software. Das Betriebssystem, auf dem die Anwendung läuft, ist ein beliebiges Linux-Betriebssystem. Um die Anwendung so sicher, stabil und fehlertolerant wie möglich zu gestalten, wurden aktuelle Verfahren und Konzepte genutzt, die die Programmiersprache C++14 zu Verfügung stellt. Auf der anderen Seite wurde auf bestimmte Sprachmittel, die sich als besonders fehleranfällig und aufwändig erwiesen haben, verzichtet.

Im Folgenden ist eine Liste dargestellt, die die wichtigsten Punkte aufzeigt:

- Auf Raw Pointer wurde weitestgehend verzichtet, um möglichen Fehlern, die durch falsche Nutzung entstehen, vorzubeugen. Einzige Ausnahme ist die Implementierung von kryptographischen Funktionen innerhalb der Kryptographie-Schicht, da hier einerseits ein großer Geschwindigkeitsbedarf besteht und andererseits direkter Speicherzugriff teilweise notwendig ist.
- Dynamische Speicherobjekte wurden niemals direkt mittels Raw Pointer reserviert und bearbeitet, sondern stets über Smart Pointer. Dieses Vorgehen soll u.a. Speicherlecks vorbeugen und dem Softwareentwickler die Aufgabe der manuellen Speicherverwaltung abnehmen.
- Durch den ausschließlichen Einsatz von Smart Pointern wurde weiterhin die sog. „Rule of Zero“[4] angewandt, die vorschreibt, dass auf selbst implementierte Copy- und Move-Konstruktoren sowie Destruktoren verzichtet werden soll.
- Auf C-Konstrukte, wie z.B. C-Funktionen, wurde verzichtet. Stattdessen wurden die objektorientierten Gegenstücke aus der C++-Standardbibliothek eingesetzt. Der Grund ist, dass eine durchgängige Nutzung von objektorientierten Sprachmitteln die Lesbarkeit und die Kohärenz des Quelltextes erhöht. Weiterhin lassen sich auf diese Weise Funktionen und Prinzipien der objektorientierten Programmierung durchgehend anwenden. Ausnahme bildet auch hier wieder die Implementierung der kryptographischen Funktionen mit einer identischen Begründung wie unter Punkt 1.
- Bot es sich bei bestimmten Funktionen oder Methoden an (beispielsweise wenn die Funktion besonders häufig aufgerufen wird und nur sehr wenige Zeilen Quellcode umfasst), dann wurde die Funktion als Inline-Funktionen angelegt.

[4] https://rmf.io/cxx11/rule-of-zero/

5 Teststrategie

Kryptographische Primitive bilden die Grundlage für viele andere Verfahren und Protokolle. Die Anforderungen an diese Algorithmen und Funktionen hinsichtlich Fehlertoleranz und Fehlerrobustheit sind also besonders hoch. Es bestand demnach die Notwendigkeit einer möglichst umfangreichen und genauen Teststrategie. Wichtigste Hilfe um dieser Anforderung gerecht zu werden, sind die sog. Testvektoren, die im Zusammenhang mit den jeweiligen Standards veröffentlicht werden. Die für dieses Projekt eingesetzte Teststrategie sah vor, die mit den Standards veröffentlichten Testvektoren mittels Modultests zu implementieren. Dabei wurde für jede implementiere Hashfunktion ein Modultest angelegt, der jeweils 7 Datensätze enthält. Diese Anzahl an Datensätzen wurde gewählt, da genau 7 Testdaten für die Hashfunktion MD5 von Ronald Rivest im ursprünglichen RFC (vgl. [Rive92a], S. 19ff) veröffentlicht wurden. Die einzelnen Testdatensätze sind dabei, wie folgt aufgebaut:

Datensatz := (Nachricht, Hashwert)

6 Einsatz der Benchmark Software

Es wurde ein Benchmark der durchschnittlichen Laufzeitgeschwindigkeit der implementierten Hashfunktionen auf zwei verschiedenen Soft- und Hardwaresystemen durchgeführt. Dabei wurde ein beispielhaftes UDP-Paket als Eingabe herangezogen, um die in IP-Netzen vorkommenden Datengrößen zu verdeutlichen. Nachfolgend ist das verwendete UDP-Paket dargestellt:

- Größe des Pakets: 520 Bytes mit 508 Byte Nutzdaten (Payload)
- Nutzdaten: 508-mal der Buchstabe ’a’ als ASCII-Zeichen codiert (0x61)
- Quell-IP: 127.0.0.1, Ziel-IP: 127.0.0.1
- Quell-Port: 55197, Ziel-Port: 8000

Das UDP-Paket wurde dabei als Byte-String in eine Datei geschrieben und der Benchmark Software auf diese Weise übergeben.

Benchmark 1

Der erste durchgeführte Benchmark wurde auf folgender Hard- und Software durchgeführt:

Hardwareeigenschaften:

- Modellname: Lenovo Thinkpad T400
- Prozessor (CPU): Intel Core 2 Duo (P8600) mit 2,4 GHz (Gigahertz) Taktrate
- Hauptspeicher (RAM): 3GB (Gigabyte), davon 2,9 nutzbar

Softwareeigenschaften:

- Betriebssystem: Fedora 23 (Linux)
- Betriebssystemtyp: 32 Bit

Eigenschaften des Compilers:

- Name: GCC bzw. G++ (Cross GCC, Linux)
- Version: 5.3.1-2
- Einstellungen: Höchste Optimierungsstufe (-O3)

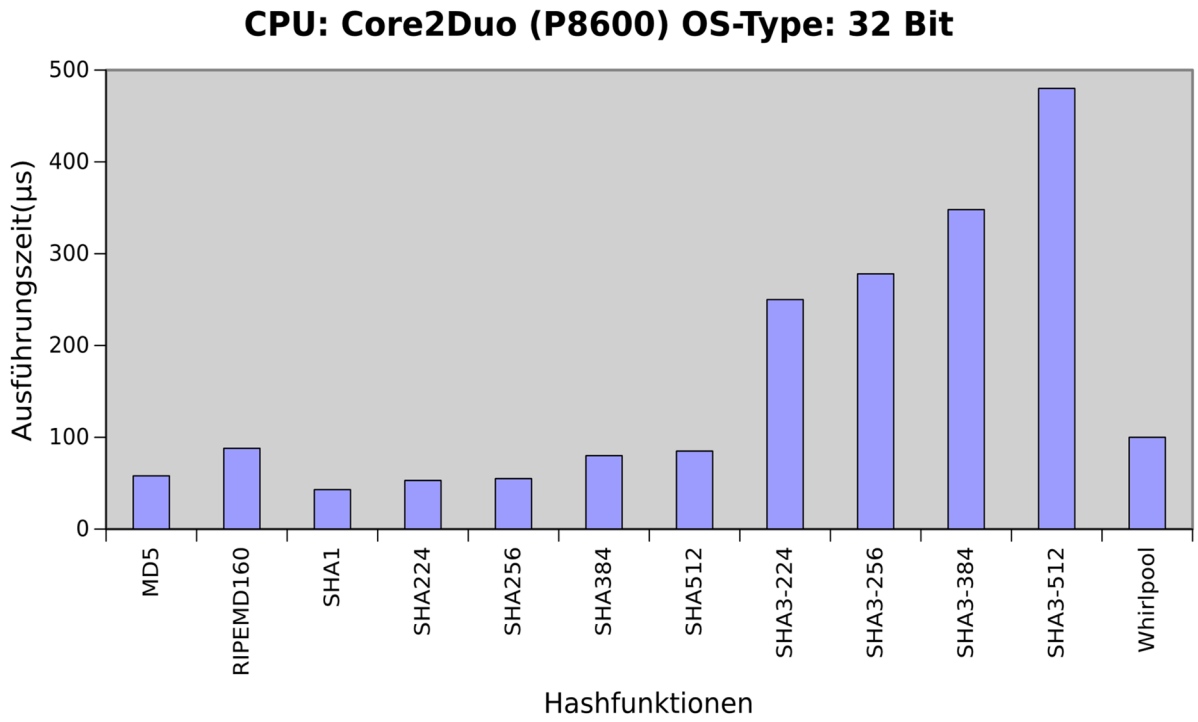

Abb. 3: Ergebnis Benchmark 1

Benchmark 2

Der zweite durchgeführte Benchmark erfolgte auf folgender Hard- und Software:

Hardwareeigenschaften:

- Modellname: MacBook Pro (2007)
- Prozessor (CPU): Intel Core 2 Duo (P8600) mit 2,4 GHz (Gigahertz) Taktrate
- Hauptspeicher (RAM): 4GB (Gigabyte)

Softwareeigenschaften:

- Betriebssystem: Mac OS X El Capitan
- Betriebssystemtyp: 64 Bit

Eigenschaften des Compilers:

- Name: GCC bzw. G++ (Cross GCC, Mac OS X)
- Version: 6.2.0
- Einstellungen: Höchste Optimierungsstufe (-O3)

Die Ergebnisse verdeutlichen, dass auf einem 64-Bit-Prozessor und einem 64-Bit-Betriebssystem, die Ausführungsgeschwindigkeit insgesamt höher sind als auf einem 32-Bit-Betriebssystem. Darüber hinaus besitzen SHA-384 und SHA-512 auf einem 64-Bit-System eine höhere relative Ausführungsgeschwindigkeit als die Funktionen SHA-224 und SHA-256.

Auf einem 32 Bit System dagegen ergibt sich ein gegenteiliger Effekt. Dieser Umstand wird damit begründet, dass SHA-384/512 intern mit 64-Bit-Wörtern arbeiten, was besonders auf einem 64-Bit-Prozessor (mit 64-Bit-OS) zu einer effizienteren Nutzung der CPU-Register führt.

SHA-3 besitzt insgesamt eine niedrigere Ausführungsgeschwindigkeit als die anderen verglichenen Hashfunktionen. Besonders jedoch auf 32-Bit-Systemen ist die Laufzeit sehr groß. Dieser Umstand ist ebenfalls darauf zurückzuführen, dass SHA-3 mit 64-Bit-Wörtern arbeitet. Die Ausführungsgeschwindigkeit kann jedoch stark optimiert werden, auch auf 32-Bit-CPUs und -Betriebssystemen. Dies war jedoch nicht mehr Untersuchungsgegenstand des Projektes. Abschließend wird noch ein weiteres Mal darauf hingewiesen, dass die fehlenden Optimierungen bei der Analyse der Ergebnisse berücksichtigt werden müssen. Dieser Umstand bekräftigt noch einmal, dass dieses Programm ausschließlich für den Einsatz in der Lehre konzipiert wurde und auch nur in einem solchen Kontext eingesetzt werden sollte.

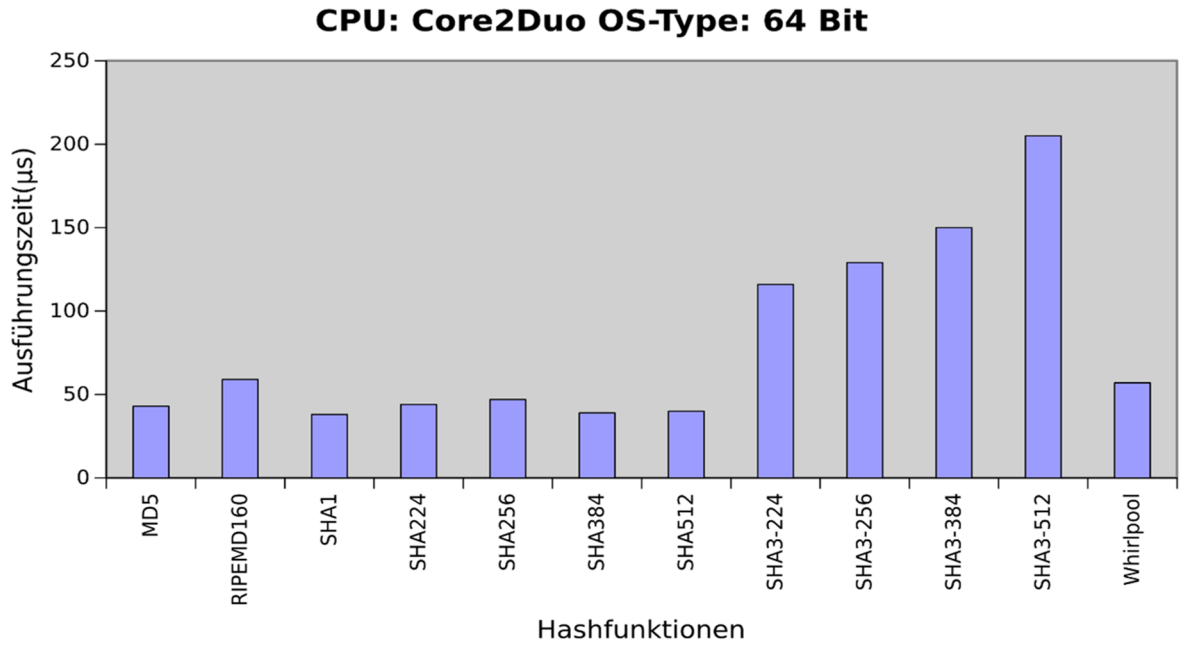

Abb. 4: Ergebnis Benchmark 2

7 Fazit

Es konnten alle in Kapitel 2 aufgeführten Anforderungen abgedeckt werden und da die Software sich durch einige Eigenschaften, wie z.B. die reine C++ Implementierung, von bereits vorhandenen Umsetzungen abgrenzt, wertet der Autor dieses Projekt als Erfolg. Es ist eine stabile, einfach zu wartende und weitestgehend modulare Software entstanden. Probleme, die im Zusammenhang mit den Hashfunktionen auftreten, wurden gelöst und in eine eigene Komponente gekapselt. Auf diese Weise konnten die stark fehleranfälligen Low-Level-Operationen in eine separate logische Einheit isoliert werden, was zu einer einfacheren Wartung und Testbarkeit der Software führt. Allerdings bleiben noch einige offene Punkte, die weitere Bearbeitung benötigen:

- Die Laufzeit der implementierten Hashfunktion sollte optimiert werden.
- Aufgrund der großen Komplexität von SHA-3 konnte dieses Verfahren nur oberflächlich behandelt werden. Es bietet sich eine umfangreiche Überarbeitung der vorliegenden Implementierung an.
- Die hier eingesetzte Teststrategie muss noch erweitert werden, um eine möglichst fehlerfreie Software zu garantieren.

Literatur

[BaRi03] P. Barreto and V. Rijmen: The whirlpool hashing function, Scopus Tecnologia S.A. and Cryptomathic NV (2003) URL http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html. Stand 2016-06-06.

[Chen06] L. Chen: Nist comments on cryptanalytic attacks on sha-1, NIST (2006) URL: http://csrc.nist.gov/groups/ST/hash/statement.html. Stand 2016-07-24.

[Dai09] Wei Dai: Crypto++ 5.6.0 benchmarks (2009)
URL: https://www.cryptopp.com/benchmarks.html. Stand 2016-11-01.

[Dang12] Q. Dang: Recommendation for applications using approved hash algorithms, National Institute Of Standards And Technology (2012) URL: http://csrc.nist.gov/publications/nistpubs/800-107-rev1/sp800-107-rev1.pdf Stand 2016-05-23.

[DoBP96] H. Dobberting, A. Bosselaers, und B. Preneel: Ripemd-160 : A strengthened version of ripemd, German Information Security Agency and Katholieke Universiteit Leuven and ESAT-COSIC (1996) URL: http://www.esat.kuleuven.be/~bosselae/ripemd160/pdf/AB-9601/AB-9601.pdf Stand 2016-05-28.

[EsJo01] D. Eastlake and P. Jones: Us secure hash algorithm 1 (sha1), Motorola and Cisco Systems (2001) URL: https://tools.ietf.org/pdf/rfc6234 Stand 2016-05-22.

[MeOV96] A. Menezes, P. van Oorschot, und S.Vanstone: Handbook of Applied Cryptography, CRC Press (1996).

[Merk79] R. Merkle: Secrecy, Authentication, and Public Key Systems, Stanford University (1979). URL: http://www.merkle.com/papers/Thesis1979.pdf. Stand 2016-06-02.

[Nist12] NIST. Fips pub 180-4 : Secure hash standard(shs), National Institute of Standards and Technology (2012) URL: http://www.nist.gov/customcf/get_pdf.cfm?pub_id=910977, Stand 2016-05-26.

[Nist15] NIST. Sha-3 standard: Permutation-based hash and extendable-output functions, National Institute of Standards and Technology (2015) URL: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf. Stand 2016-06-24.

[Rive92a] R. Rivest: The md5 message-digest algorithm (Rfc), MIT Laboratory for Computer Science and RSA Data Security, Inc. (1992)
URL: https://www.ietf.org/rfc/rfc1321.txt Stand 2016-05-07.

[Rive92b] R. Rivest: The md4 message-digest algorithm (Rfc), MIT Laboratory for Computer Science (1992)
URL: https://tools.ietf.org/html/rfc1320 Stand 2016-05-08.

[VAC+09] O. Vogel, I. Arnold, A. Chugtai, E. Ihler, T. Kehrer, U. Mehlig, und U. Zdun: Software-Architektur Grundlagen-Konzepte-Praxis, Spektrum Akademischer Verlag GmbH (2012).