Implementierung der SHA-1 Hashfunktion auf einem ATmega328p Mikrocontroller

Grundlagen, Firmwaredesign und Implementierung

Christoph Beickler Matr.-Nr: 2252027
B.Eng. Elektro und Informationstechnik
Fakultät Ingenieurwissenschaften und Informatik TH Aschaffenburg
Aschaffenburg, Deutschland
s221003@th-ab.de

***Abstract*—Im Rahmen dieses Projektes wurde von mir der Secure Hash Algorithmus (SHA1) auf einem ATmega328p (Arduino Uno) in ANSI C implementiert. Die zu hashenden Daten werden hierbei über die UART-Schnittstelle empfangen. Ein Steuerkommando startet die Hash-Kalkulation und nach Empfang des Ausgabekommandos (#$) wird der ermittelte 160 Bit breite Hashwert über die UART-Schnittstelle zurückgesendet. Diese Arbeit diskutiert die hier verwendeten kryptografischen Grundlagen und beschreibt den Aufbau und das Design der implementierten Firmware.**

Keywords — SHA-1, ATmega328p,

I. Einleitung (*Heading 1*)

Mikrocontroller sind heutzutage ein nicht wegzudenkender Bestandteil der meisten elektronischen Schaltungen. Sie verarbeiten und speichern Daten, welche ihnen über verschiedenste Wege zugespielt werden, wie Sensoren, Speichereinheiten oder durch Benutzereingaben. Neben allgemeinen Anwendungen in sicherheitstechnischen Bereichen verwendet man kryptografische Hashfunktionen, um Firmwaren zu verifizieren, um z. B. Manipulation oder Fehler bei der Datenübertragung auszuschließen, sowie zur Prüfung von gesendeten und empfangenen Datenpaketen zwischen zwei Kommunikationsendpunkten. Diese Hashfunktionen können aus einer Datenabfolge beliebiger Länge nun mithilfe von mathematischen Einmalfunktionen einen nicht reversiblen Hash erstellen. Im Falle von SHA-1 ist dies eine 160 Bit breite Datenfolge. Jede Änderung an den Eingabedaten führt hierbei zu einer kompletten Änderung des Ausgabewertes.

II. Theoretische Grundlagen

A. Mathematische Einwegfunktion

Eine mathematische Einwegfunktion ist eine effizient berechenbare, aber nicht effizient umkehrbare Rechenfunktion. D. h., zu einem gegebenen Funktionswert ist es innerhalb einer realistischen Zeit nicht möglich, die passende Eingabe zu finden. Kryptografische Hashfunktionen machen sich diesen Umstand zunutze, um aus einer beliebig langen Datenfolge einen definiert langen Hash zu erstellen. Eine Rekonstruktion der eingegebenen Daten ist mit aktuellen technischen Mitteln praktisch unmöglich. Absolute Sicherheit bieten diese nicht, da die Hashs über Lookup Tables oder Rainbow Tables. Dies gilt nicht als Rekonstruieren der Eingabe, da so lange Werte gehasht werden, bis man per Zufall den richtigen Eingabewert gefunden hat.

B. Typische Anwendungen von Einwegfunktionen

Ein typisches Anwendungsfeld ist die Integritätsprüfung: Dateien, Nachrichten oder Firmware werden gehasht und später erneut geprüft, um Manipulationen oder Übertragungsfehler zu erkennen.

Ein weiterer Einsatzzweck sind Authentifizierungen, z.B. bei der Speicherung von Passwörtern. Hier werden Hashwerte statt der Passwörter im Klartext gespeichert. Da die Kennworteingabe immer erst durch einen Hash-Algorithmus laufen muss, bevor die Eingabe und der gespeicherte Wert verglichen werden, ist das Bekanntwerden des gespeicherten Hashs kaum ein Problem, da dieser bei der Eingabe einen anderen Hash-Wert erzeugen würde.

C. Preimage Resistance und Second Preimage Resistance Charakteristik einer Einwegfunktion

Die Preimage Resistance (Urbildresistenz) beschreibt die Resistenz einer Einwegfunktion, dass zu einem gegebenen Hash-Wert, der aus einer unbekannten Nachricht stammt, ein Eingabewert gefunden werden kann, welcher den gleichen Hash-Wert zurückgibt. Dies ist dann mit sehr hoher Wahrscheinlichkeit der ursprüngliche Eingabewert.

Die Second Preimage Resistance beschreibt die Resistenz einer Einwegfunktion, dass zu einem bekannten Eingabewert ein zweiter Eingabewert gefunden werden kann, welcher den gleichen Hash-Wert produziert.

D. Was sind Kollisionen und welche Auswirkungen haben sie auf die Sicherheit von Hashfunktionen?

Unter einer Kollision versteht man den Fall, dass zwei verschiedene Eingabedaten a und b, mit $a \neq b$, den gleichen Hash-Wert erzeugen. Für eine kryptografische Hashfunktion ist dies besonders kritisch, da der Hash-Wert häufig als eindeutige Identifikation zur Integritätsprüfung verwendet wird. So könnte z. B. zu einer Datei mit schadhaftem Code eine harmlose gefunden werden, welche den gleichen Hash produziert. Diese harmlose Datei kann dann als Vorwand genutzt werden, um eine Signatur zu erzeugen, welche auch zum schadhaften Code passt. Bei einem rein Hash-basierten Prüfverfahren kann so der schadhafte Code unbemerkt eingeschleust werden.

Ingenieuren von Google und dem CWI Amsterdam ist eine solche Kollision bereits gelungen. Sie konnten zwei

unterschiedliche PDF-Dateien erzeugen, welche beide denselben Hash-Wert produzieren [1]. Der SHA-1 Hash-Algorithmus gilt seitdem nicht mehr als kollisionssicher und ist für sicherheitskritische Anwendungen nicht zu empfehlen. Dass es solche Kollisionen geben muss, ist aus mathematischen Gründen unvermeidbar. Wenn aus einer unendlich großen Eingabemenge nur eine endliche Menge an Ausgaben produziert wird, müssen zwangsläufig verschiedene Eingaben das gleiche Ergebnis produzieren. Ziel der Hashfunktion kann es nicht sein, kollisionssicher zu sein, sondern solche Kollisionen so schwer wie möglich auffindbar zu machen.

E. *Ist die boolesche XOR-Funktion eine geeignete Methode, die Integrität einer Nachricht zu prüfen?*

Nein, denn die XOR-Prüfsumme ist linear und dadurch leicht zu umgehen. Änderungen an verschiedenen Stellen können einander aufheben. Sodass der gleiche XOR-Wert entsteht. Dies macht Kollisionen und gezielte Manipulationen sehr einfach möglich.

F. *Das Geburtstagproblem und seine Relation zu Hash-Kollisionen*

Das Geburtstagsproblem beschreibt das Phänomen, dass in einer relativ kleinen Gruppe bereits mit überraschend hoher Wahrscheinlichkeit zwei Personen am gleichen Tag Geburtstag haben. Der Grund dafür ist, dass nicht ein bestimmter Geburtstag gesucht wird, sondern irgendein passendes Paar. Mit jeder weiteren Person wächst die Anzahl möglicher Paare stark an. Der Zusammenhang zu Hash-Kollisionen ist identisch: Werden viele verschiedene Eingaben gehasht, steigt die Wahrscheinlichkeit, dass zwei unterschiedliche Eingaben den gleichen Hash-Wert produzieren. Beim SHA-1 Algorithmus benötigt man rund 2^(n/2) Versuche, um mit einer Wahrscheinlichkeit von p > 0,5 eine Kollision zu finden [2]

G. *Mathematische Einwegfunktionen im Kontext der Kryptowährung Bitcoin*

Die Einwegfunktionen spielen bei der Kryptowährung Bitcoin eine zentrale Rolle, da der gesamte Aufbau auf Hashfunktionen basiert. Im Falle von Bitcoin wird das Proof-of-Work-Mining mithilfe des SHA-256-Algorithmus verwendet.

Beim Proof-of-Work müssen Miner einen Blockheader (Nonce) so verändern, dass der resultierende Hash-Wert bestimmte Bedingungen erfüllt. Dies kann nur durch das Ausprobieren vieler verschiedener Nonces gelingen. Ist dieser Wert gefunden, so wird der neue Block mit den Informationen der getätigten Transaktionen sowie dem Hash des vorherigen Blocks an die Kette (Blockchain) angehängt. Das Verarbeiten des vorherigen Hashs macht eine Manipulation der vorherigen Transaktionen faktisch unmöglich.

III. Implementierung

Nachdem nun die theoretischen Grundlagen diskutiert wurden, soll folgend auf den Aufbau und die Implementierung der Firmware eingegangen werden. Zentrale Rolle spielen hierbei die Umsetzung der USART-Schnittstelle sowie die Implementierung des SHA-1 Algorithmus.

A. *Firmware-Architektur und Programmablauf*

Die Firmware besteht aus drei Bestandteilen: main.c als Steuerprogramm, usart.c/usart.h zur Steuerung der seriellen Kommunikation sowie sha1.c/sha1.h zur Implementierung der Hashfunktion. Der Ablauf ist so gestaltet, dass eingehende Daten über die UART-Schnittstelle asynchron per Interrupt empfangen und in den Eingangs Buffer „rxBuffer“ gespeichert werden. Die eigentliche Hashberechnung wird anschließend im Hauptprogramm ausgelöst. Die Empfangs-ISR ist nur zum Speichern der Daten, um keine unnötig langen „Totzeiten“ in der seriellen Kommunikation zu haben und ggf. eingehende Daten zu verpassen.

Beim Programmstart wird nach der Initialisierung der USART-Schnittstelle und dem Einschalten einer LED auf dem Arduino eine Endlosschleife ausgeführt, welche bei jeder Iteration überprüft, ob das *startFlag,* zum Starten der Berechnung, oder das *answearFlag*, zur Ausgabe des Ergebnisses, gesetzt ist. Diese Flags werden von der ISR nach empfang der jeweiligen Steuerkommandos gesetzt.

B. *USART-Schnittstelle*

Die USART-Schnittstelle wird im Hauptprogramm mit *usart_init()* initialisiert. Hierbei wird aus der gewünschten Baudrate von 9600 der Prescaler berechnet und in die *UBRROH/UBRROL*-Register geschrieben. Die Baudrate ist mit 9600 niedrig angesetzt, da eine stabile Kommunikation hier Vorrang vor Geschwindigkeit hat. Für den Betrieb werden sowohl die Empfangs- als auch die Sendefunktion benötigt. Hierfür werden die Register *RXEN0* und *TXEN0* gesetzt. Zur Nutzung der Interrupt-Funktion wird *RXCIE0* gesetzt.

Für die Ausgabe werden mehrere Hilfsfunktionen genutzt: *uart_sendByte()* sendet ein einzelnes Byte blockierend, *uart_sendString()* sendet nullterminierte Strings und *uart_transmit_hex()* gibt ein Byte als zwei Hex-Zeichen aus. Letztere Funktion wird genutzt, um den 160 Bit Hash als 40 Hex-Zeichen darzustellen, was für Debugging und Vergleich mit Testvektoren besonders hilfreich ist

Der Datenempfang erfolgt über die *ISR(USART_RX_vect)*. Bei jedem empfangenen Zeichen wird *UDR0* gelesen und das Byte lokal verarbeitet. Um die Bedienung im Terminal zu erleichtern, wurde ein Echo implementiert: Jedes empfangene Zeichen wird direkt wieder zurückgesendet, sodass die Eingabe sichtbar ist. Zeilenumbrüche (CR/LF) werden dabei dargestellt, aber nicht in den Hash-Puffer übernommen. Hierdurch ist die Eingabe über das Terminal strukturierter und das typische Enterdrücken berücksichtigt, ohne dass diese Steuerzeichen den Hashwert beeinflussen. In der aktuellen Implementierung beeinflusst Backspace nicht den Inhalt des Puffers, sondern nur die Darstellung.

Die eigentlich zu verarbeitenden Daten werden im *rxBuffer* gespeichert, wobei *rxBufPos* die aktuelle Position im Buffer markiert. Da der ATmega328P nur begrenzten SRAM besitzt, ist die Buffergröße eine relevante Größe. In dieser Implementierung ist *RX_BUFFER_SIZE* auf 1024 Byte gesetzt, um auch längere Nachrichten aufnehmen zu können. Um Fehlerfälle abzufangen, wird bei einem Überlauf der Puffer zurückgesetzt und beide Flags werden gelöscht und eine Nachricht über den Terminal ausgegben.

Zur Steuerung der Anwendung werden zwei Kommandos verwendet, die am Ende der aktuellen Eingabe gesendet werden:

- **„#!“** startet die Hash-Kalkulation (*startFlag = 1*)
- **„#$“** triggert die Ausgabe des zuletzt berechneten Hashwertes (*answearFlag = 1*)

Die Erkennung dieser Kommandos geschieht direkt in der ISR. Wird ein Steuerkommando erkannt, werden die beiden Kommandozeichen aus dem Puffer entfernt (*rxBufPos –= 2*), sodass ausschließlich die eigentlichen Nutzdaten gehasht werden. Hierdurch ist ein zeitunabhängiges Übertragen der Daten möglich, welches nur durch tippen der Steuerkommandos beendet wird.

C. *Synchronisation zwischen ISR und Hauptprogramm.*

Um Verfälschung der zu verarbeitenden Daten vor Start der Hash-Berechnung zu verhindern, müssen diese Daten atomar übergeben werden, also ohne Störung durch weitere einkommende Daten. Hierfür wird kurzzeitig global der Interrupt gesperrt (*cli()*), um eine korrekte *rxBufPos* zu übernehmen und anschließend das *startFlag* zurückzusetzen. Danach wird der Interrupt wieder freigegeben (*sei()*). Zusätzlich wird während der Hashberechnung der RX-Interrupt deaktiviert (*UCSR0B &= ~(1<<RXCIE0)*). Nach Abschluss der Hashberechnung wird der Interrupt wieder aktiviert. Damit wird sichergestellt, dass die Nachricht, die gehasht wird, exakt dem Pufferinhalt zum Zeitpunkt des Startkommandos entspricht.

D. *Implementierung des SHA-1-Algorithmus*

Der SHA-1-Algorithmus ist in sha1.c umgesetzt und nutzt einen Kontext (*SHA1_CTX*), der den aktuellen Zustand (*state[5]*), den Bitzähler (*count[2]*) sowie einen 64-Byte Blockpuffer (*buffer[64]*) enthält. Der Ablauf folgt der klassischen API-Struktur:

- *sha1_init()* setzt den Kontext zurück und initialisiert die fünf 32-Bit Startwerte.
- *sha1_update()* verarbeitet eingehende Daten beliebiger Länge, füllt den 64-Byte Blockpuffer und ruft bei vollem Block die Kompressionsfunktion auf.
- *sha1_final()* führt Padding und Längenanhang durch und schreibt den finalen 160 Bit Hashwert in das Digest-Array

In sha1_transform() wird jeweils genau ein 512-Bit Block (64 Byte) verarbeitet und in den internen Hash-Zustand (*state[5]*) „eingemischt“. Der Ablauf ist dabei in drei klar getrennte Schritte unterteilt:

Zunächst werden die 64 Byte Blockdaten in 16 Wörter à 32 Bit umgewandelt. Da SHA-1 intern big-endian arbeitet, werden jeweils vier aufeinanderfolgende Bytes zu einem 32-Bit Wort zusammengesetzt (Byte 0 ist dabei das höchstwertige Byte). Diese 16 Wörter bilden den Start der Message Schedule.

Im zweiten Schritt wird diese Message Schedule von 16 auf 80 Wörter erweitert. Dazu werden ab Runde 16 neue Wörter über eine XOR-Verknüpfung älterer Werte berechnet und anschließend mittels der Rotationsfunktion um 1 Bit nach links rotiert. : w[t]=ROTL1(w[t−3]⊕w[t−8]⊕w[t−14]⊕w[t−16])

Danach folgt die eigentliche Kompressionsrunde über 80 Iterationen. Hierzu werden die fünf Arbeitsregister *a, b, c, d, e* mit dem aktuellen Zustand *state[0..4]* initialisiert. In jeder Runde wird abhängig vom Rundenzähler eine andere boolesche Funktion *f* sowie eine Rundkonstante *K* verwendet (0–19, 20–39, 40–59, 60–79). Pro Runde wird ein temporärer Wert berechnet, der aus einer *ROTL5*-Rotation von *a*, der Funktion *f(b,c,d)*, dem Register *e*, der Konstante *K* und dem jeweiligen Schedule-Wort *w[t]* besteht. Anschließend werden die Register verschoben, wobei *b* um 30 Bit nach links (entspricht 2 Bit nach rechts) rotiert wird.

Am Ende der 80 Runden werden die Arbeitsregister *a* bis *e* wieder Modulo 2^32 auf den ursprünglichen Zustand *state[0..4]* addiert. Damit ist der Block vollständig verarbeitet und der Kontext enthält den neuen Zwischenstand für den nächsten 64-Byte Block.

Padding in *sha1_final()* erfolgt über das Anhängen von 0x80, anschließend von 0x00 Bytes bis die Nachrichtenlänge 448 Bit Modulo 512 erreicht, und zuletzt durch Anhängen der ursprünglichen Nachrichtenlänge als 64-Bit Wert. Der finale Hash wird dann aus dem Kontextzustand wieder big-endian in das Digest-Array geschrieben. Für die Ausgabe über UART wird dieser Digest anschließend byteweise in Hex konvertiert.

E. *Ausgabe des Hashwertes und senden per USART*

Nach erfolgreichem Hashen kann das Ergebnis über das Ausgabekommando „#$“ abgerufen werden. Im Hauptprogramm wird dann „SHA1: “ gefolgt von den 20 Ergebnis-Bytes als Hexwert übertragen. Zusätzlich wird eine LED getoggelt, um auch ohne Terminal sichtbar zu machen, dass eine Ausgabe erfolgt ist.

Durch diese Implementierung wird die Aufgabenstellung erfüllt, Eingabedaten über UART zu empfangen, per Steuerkommando die Hashberechnung zu starten und nach Empfang eines Ausgabekommandos den 160 Bit breiten SHA-1 Hashwert über die UART-Schnittstelle zurückzusenden.

IV. Zusammenfassung

Im Rahmen dieses Projekts wurde von mir der Secure Hash Algorithmus SHA-1 auf einem ATmega328p-Mikrocontroller in ANSI C implementiert. Die Firmware ermöglicht das Empfangen und Senden von Daten über eine serielle USART-Schnittstelle. Diese empfangenen Daten werden anschließend zu einem 160 Bit breitem Hashwert verarbeitet.

Augenmerk lag hierbei auf einem sicheren, störungsfreien Kommunikationsablauf. Diese Sicherheit wird über das Verwenden der Interrupt Service Routine erreicht, welche nur die minimal nötigen Arbeitsschritte ausführt, ohne die Kommunikation zu behindern. Des Weiteren werden gezielt die Interrupt Routinen ausgeschaltet, wenn der Algorithmus die

empfangenen Daten bearbeitet, um so die Integrität des Buffers zu bewahren.

Die Steuerung der Anwendung erfolgt über zwei Kommandos, die am Ende der Eingabe gesendet werden: „#!“ zum Start der Hashberechnung und „#$“ zur Ausgabe des Ergebnisses. Bei Erkennung dieser Kommandos entfernt die ISR die Kommandozeichen aus dem Puffer, sodass ausschließlich die Nutzdaten in die Hashberechnung eingehen.

Aktuell wird die Datengröße auf die Größe des Empfangs-Buffers limitiert, welcher 1024 Byte groß ist, was der Hälfte des dem ATmega328p verfügbaren Arbeitsspeicher entspricht. Das statische Festlegen dieser Buffergöße verhindert, dass im laufenden Betrieb ein Speicheroverflow geschehen kann, limitiert aber gleichzeitig auch die maximal übertragbare Länge einer Datenkette.

Diese Ausarbeitung diskutiert die theoretischen Grundlagen, über Einwegfunktionen, Sicherheitseigenschaften dieser Einwegfunktionen über die Verbindung mit dem Geburtstagsproblem bis hin zur Relevanz mit Bezug auf Bitcoin. Behandelt wurden diese Themen anhand der Leitfragen aus der Aufgabenstellung. Anhand dieser Grundlagen wurde in Kapitel III die konkrete Umsetzung dieser Firmware beschrieben: die serielle Kommunikation mit Kommandoprotokoll, das Interrupt-basierte Puffern der Daten und die schrittweise Verarbeitung im SHA-1 Algorithmus einschließlich Padding, Message Schedule und der 80-rundigen Kompressionsfunktion.

Insgesamt entsteht so eine ausführliche und nachvollziehbare Beschreibung dieser Projektarbeit, mit einer stabil laufenden Firmware, welche die Aufgabenstellung erfüllt und einen Hashwert angemessen schnell und reproduzierbar berechnet und ausgibt.

V. Schlussfolgerung

Die Implementierung der erwarteten Funktionen wurde erfolgreich abgeschlossen. Sowohl die Kommunikation über die serielle USART-Schnittstelle als auch die Umsetzung des SHA-1-Algorithmus sind gelungen und erfolgreich getestet. Der geforderte Funktionsumfang wurde vollständig implementiert.

Diese Projektarbeit zeigt die praktikable Umsetzung solcher Funktionen auf kleinen Embedded-Systemen. Durch die klare Trennung zwischen der Kommunikations- und Hash-Logik lässt sich der Code einfach handhaben und leicht verändern bzw. erweitern.

Die Nutzung der Interrupt-basierten Kommunikation vermeidet Laufzeitfehler, bei denen der Mikrocontroller ggf. nicht bereit wäre, Daten zu empfangen, und erlaubt eine schnellere Reaktionsgeschwindigkeit.

Für nicht sicherheitskritische Anwendungen ist eine solche Implementierung durchaus geeignet, jedoch müssen hier die geringe Speichergröße sowie die aktuell geringe Baudrate betrachtet werden. Längere Datenketten lassen sich so nicht hashen. Um dieses Problem zu umgehen, könnte über einen externen Speicher nachgedacht werden, oder aber ein Preprocessing, welches die Daten in geeignet große Blöcke aufteilt. Hierfür müsste die Firmware umprogrammiert werden, um nach der Kalkulation des Hashwertes weitere Daten zu empfangen und mit diesen die Berechnung weiterzuführen. Gerade bei größeren Datenmengen sollte auf die Güte der Datenverbindung und eine erhöhte Datenrate Wert gelegt werden. Für diesen Fall könnte es sich auch eignen, die Größe des Empfangs-Buffers auf einen maximal großen Wert zu erhöhen. Durch die statische Natur der Firmware sollte dies jedoch ohne größere Probleme möglich sein.

Insgesamt empfiehlt es sich allerdings, über die Implementierung einer moderneren und sicheren Alternative nachzudenken. SHA-256, SHA-512 sowie SHA-3 bieten hier Alternativen mit wesentlich höherer Sicherheit, welche dadurch auch geeigneter für zukünftige Anwendungen sind. Durch die bereits besprochenen Kollisionsrisiken lässt sich SHA-1 nicht mehr empfehlen und sollte nur als nicht sicherheitsrelevante Bestandspflege in Erwägung gezogen werden.

Eine Rückgabe von Fehlercodes, um im Betrieb die Fehlersuche eingrenzen zu können, würde das Einsetzen und Anwenden der Hardware noch weiter erleichtern.

Referenzen

[1] M. Jäger, “Erfolgreiche Kollisionsattacke auf SHA-1 Hashfunktion,” *Security-Insider*, Mar. 08, 2017. [Online]. Available: https://www.security-insider.de/erfolgreiche-kollisionsattacke-auf-sha-1-hashfunktion-a-587296/

[2] “Hashfunctions talk,” *Uni Freiburg*. http://www2.informatik.uni-freiburg.de/~stachnis/pdf/stachniss-habil-talk-hashfunctions.pdf Folie 21

[3] “FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION: Secure Hash Standard (SHS),” National Institute of Standards and Technology, 180–4, Aug. 2015. doi: 10.6028/NIST.FIPS.180-4.