INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Implementierung und Evaluation von WalnutDSA als Modul für FreeRTOS, Riot und Linux Kernel

Konrad Haslberger, Do Hwan Kim und Stefan Seil

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 18.12.2017

. . . . . .

(Unterschrift des Kandidaten)

Abstract

WalnutDSA ist ein Algorithmus digitaler Signaturen für ressourcenbeschränkte Geräte, der mit sogenannten *Braids* arbeitet und in linearer Laufzeit verifiziert. Das Ziel dieser Arbeit war es WalnutDSA für FreeRTOS zu implementieren und auf einem ESP8266 zu evaluieren. Die Arbeit beginnt mit Ausführungen zu den Grundlagen, woraufhin die Braid-Theorie und die E-Multiplikation näher beleuchtet werden. Daraufhin wird in einem ausführlichen Kapitel zu WalnutDSA auf die einzelnen Bestandteile des Algorithmus eingegangen. Es folgen Details zur Implementierung und die Evaluierung mit Tests zur Korrektheit und Laufzeit. In einem plattformübergreifenden Versuch wird die Kompatibilität der eigenen Implementierung mit zwei anderen getestet und ihre Laufzeiten untereinander und mit Implementierungen anderer Algorithmen verglichen. Aus den Testwerten geht eine Diskussion über die Umsetzbarkeit und Sicherheit des Verfahrens hervor, woraufhin in abschließenden Gedanken ein Fazit über den Nutzen von WalnutDSA in der Kryptographie im Bereich Internet of Things gezogen wird.

Abstract

Die Notwendigkeit von Sicherheit in IoT wird mit Verbreitung von IoT Geräte zunehmend höcher. Jedoch manche asymmetrische Verschlüsselung benötigt große Rechenleistung, der für solche Geräte mit kleine Ressourcen nicht vorhanden ist. WalnutDSA ist ein digitale Signaturalgorithmus, der Integrität, Authentizität und Nichtabstreitbarkeit gewährlesten kann. Der größte Vorteil von WalnutDSA ist, dass die Verifizierung von Signatur sehr schnell ist, sodass es auch bei IoT Geräte in passabler Zeit möglich ist. Noch ein Vorteil ist, dass die Laufzeit von Verifizierung linear mit der Sicherheitslevel wächst. In dieser Arbeit wurde WalnutDSA für den Betriebsystem RIOT OS als ein Modul mit Arduino M0 Pro als Testumgebung implementiert. Dabei werden die Datenstruktur für Bestandteile und konkrete Algorithmen untersucht.

Abstract

Der Trend des *Internet of Things* stellt durch die komplexe Vernetzung vieler leistungsschwacher Geräte neue Herausforderungen an kryptographische Systeme. Zudem werden durch die laufende Forschung an Quantencomputern neue Verfahren benötigt, die vor quantenbasierte Angriffe geschützt sind. WalnutDSA ist ein digitales Signaturverfahren, welches zur Authentisierung von Geräten im Netz eingesetzt werden kann. Der Algorithmus basiert auf der Zopftheorie und verspricht damit geringen Ressourcenverbrauch sowie eine Resistenz gegen Angriffe eines Quantenrechners. Im Rahmen dieser Arbeit wurde WalnutDSA auf einem Rapberry Pi als Modul für den Linux Kernel implementiert. Die Umsetzung wurde gegen zwei andere Implementierungen getestet und bezüglich Rechenzeit und benötigem Speicherplatzverbrauch evauliert. Während die Leistung der Implementierung der Autoren von WalnutDSA nicht erreicht wurde, so kann das Kernelmodul als mögliche Referenzimplementierung des Verfahrens für Linux gesehen werden. Die theoretischen Untersuchungen zeigen, dass WalnutDSA viel Potential für ein zukunftträchtiges digitales Signaturverfahren innehält.

Inhaltsverzeichnis

1 Einleitung

Durch die Fortschritte der Rechnerarchitektur und zugehöriger Herstellungsprozesse können Computer heute in sehr kleinen Formfaktoren hergestellt werden. Die dadurch entstandene Branche des *Internet of Things (IoT)* hat einen rasanten Trend für die Verwendung von stark vernetzten eingebetteten Geräten in bisher analogen Umfeldern gestartet. Aktuelle Schätzungen prognostizieren einen wirtschaftlichen Einfluss der Branche von bis zu 6,2 Billionen US-Dollarn bis 2025 [1]. Für Endkonsumenten sind z.B. bereits rechnergesteuerte Produkte für Türschlösser, Thermostate oder Glühbirnen verfügbar. Aber auch im professionellen Kontext werden immer mehr Bereiche zur Automatisierung von Arbeitsschritten mit Computern ausgestattet, wie z.B. industrielle Kontrollanlagen oder intelligente Stromnetze. Aufgrund der komplexen Vernetzung vieler unterschiedlicher Geräte ergeben sich neue Herausforderungen zur Gewährleistung der Sicherheit und Privatsphäre von Benutzern [2]. Leider verfügen viele IoT-Geräte nicht über ausreichende Sicherheitsmaßnahmen; selbst bekannte Produkte großer Hersteller weisen Sicherheitslücken auf [3].

Um die Kommunikation dieser vernetzten Geräte abzusichern, können die üblichen Sicherheitsprotokolle der IT-Sicherheit verwendet werden. Allerding sind viele Verfahren in der Kryptographie aufgrund zu hohen Rechenaufwands nicht für kleine eingebettete Systeme geeignet. Daher ergibt sich die Suche nach sog. *Lightweight Cryptography* [4], also Verfahren welche auch im Umfeld von leistungsschwachen Computern eingesetzt werden können. Ein Teil der Sicherung von IoT-Geräten besteht in der Verwendung von Authentisierungs- und Authentifizierungsmaßnahmen. Bei der Kommunikation mehrerer Geräte möchte der Empfänger die Identität des Senders bestätigen und sicherstellen, dass sie nicht von einer dritten Partei gefälscht wurde. Es handelt sich also um eine digitale Unterschrift, mit der sich die Rechner im Netz untereinander ausweisen können. Gerade im IoT-Szenario findet kontinuierlich ein Datenaustausch zwischen den zahlreichen Geräten statt, was hohe Anforderungen an die verwendeten Verfahren stellt. Üblicherweise wird zur Authentisierung asymmetrische Kryptographie eingesetzt. Zu den traditionellen Public-Key-Verfahren zählen RSA und DSA, welche häufig in normalen Desktop-Computern oder Serversystemen verwendet werden. Diese Algorithmen basieren jedoch auf sehr rechenintensiven mathematischen Problemen, die sie aus mangelnder Leistung vieler eingebetteter Systeme für den IoT-Sektor weitestgehend ungeeignet machen. So steigt z.B. der Speicherplatzverbrauch der Verfahren zu schnell an, um den laufend wachsenden Erfordernissen an Sicherheitsmaßnahmen gerecht zu werden. Zudem ist seit geraumer Zeit ein Angriff auf Basis eines Quantencomputers bekannt, der die beiden Kryptosysteme bei Verfügbarkeit eines solchen Rechners auf einen Schlag unsicher machen würde.

Eine potentielle Lösung bietet das US-amerikanische Unternehmen SecureRF mit dem digitalen Signaturverfahren *WalnutDSA* [5]. Der Algorithmus verwendet im Gegensatz zu traditionellen asymmetrischen Kryptosystemen Berechnungsprobleme aus drei unterschiedlichen Teilbereiche der Mathematik, allen voran der Zopftheorie. Die Autoren versprechen dadurch ein Verfahren mit kleinem Speicherplatzverbrauch und geringen Leistungsanfor-

derungen für effiziente Signaturen in eingebetteten Systemen, welches zudem auch gegen Angriffe eines Quantencomputers resistent sein soll [6].

Im Rahmen dieser Arbeit soll WalnutDSA als Modul für den Linux Kernel auf einem Raspberry Pi implementiert werden und diese Implementierung folglich in Bezug auf Rechendauer und Speicherplatzverbrauch gegenüber anderen vorhandenen Verfahren untersucht werden. Während das gewählte Gerät wohl leistungsstärker als viele andere häufig verwendete eingebettete Systeme ist, so dient die Implementierung auch zur Anwendung in einem performanteren Kontrollsystem im IoT-Szenario. Zudem kann die Software durch die Verwendung der Programmiersprache C und die weitestgehend plattformunabhängige Entwicklung als Kernelmodul als quelloffene Referenzimplementierung verwendet werden.

Die vorliegende Arbeit ist wiefolgt strukturiert: Zu Beginn wird der Hintergrund zu digitalen Signaturen, der Gefahr von Quantencomputern in der Kryptographie und dem Trend des Internet of Things dargestellt. Anschließend folgt eine Übersicht über den derzeitigen Forschungsstand kryptographischer Verfahren für leistungsschwache eingebettete Geräte bzw. Verfahren mit Resistenz gegen quantenbasierte Angriffe. Es wird eine kurze Einführung in die Zopftheorie und die modulare Arithmetik in endlichen Körpern gegeben sowie die Einwegfunktion des Signaturverfahrens erläutert, um die nötigen mathematischen Grundlagen zu etablieren. Es folgt eine Übersicht über die Bestandteile von WalnutDSA, wobei sowohl auf die grundlegende Funktionsweise als auch auf interessante Aspekte bei der generischen Implementierung eingegangen wird. Im Anschluss soll die Implementierung des Verfahrens unter Linux auf dem Raspberry Pi dargestellt und evaluiert werden, wobei auch auf Probleme bei der Umsetzung sowie auf das Laufzeitverhalten und den Speicherplatzverbrauch des Programms eingegangen wird. In einer Diskussion wird schließlich die Sicherheit des Verfahrens sowie die Umsetzbarkeit und das Potenzial von WalnutDSA untersucht. Zudem werden Vergleiche sowohl zur bisherigen Implementierung von SecureRF als auch zu anderen potentiellen digitalen Signaturverfahren gezogen.

2 Einleitung

Schon bevor Kevin Aushton 1999 den Begriff *„Internet of Things“* ins Leben rief, sprach Mark Weiser 1991 in seinem Aufsatz *„The Computer for the 21st Century“* von der Vorstellung, nicht nur Menschen wären in der Zukunft als Teilnehmer Internet-ähnlicher Netzwerke zu sehen, sondern auch Geräte.[7] In den seitdem 36 vergangenen Jahren hatte die Rechnerarchitektur große Fortschritte zu verzeichnen. Dem Gesetz von Moore folgend wuchs die Anzahl der Transistoren pro Flächeneinheit exponentiell, sodass Technologie, die früher noch als Supercomputer betitelt wurde, heute in jedem Taschenrechner vorzufinden ist und Finger-große Mikrocontroller über genug Rechenleistung verfügen, um Daten verschiedener Sensoren auszuwerten oder komplexe Berechnungen durchzuführen. Ebenso rasant wuchs auch die Bereitschaft der Menschen, die Technologie in ihr alltägliches Leben aufzunehmen. In einer berühmten Prognose aus dem Jahre 1943 prophezeite Thomas Watson, ehemaliger Vorstandsvorsitzender von IBM, einen weltweiten Markt für fünf Computer. Stark im Kontrast steht dazu der heutige Alltag, in dem Mini-Computer in Form von Smartphones für viele Menschen einen festen Platz eingenommen haben. Smart Home Geräte führen den Gedanken, dem Menschen das tägliche Leben zu erleichtern, noch weiter. Intelligente Türsteuerungen, Kühlschränke oder Heizungen seien hier nur wenige Beispiele. Die Bereitschaft entwickelt sich zum Vertrauen und es werden bereits intelligente Systeme in der Fortbewegung oder in der Wasserversorgung zur Kontrolle der Trinkwasserqualität eingesetzt.

2.1 Neue Gefahren des Internets der Dinge

Ein Beispiel aus dem Sommer 2015, in dem ein dokumentierter Hack auf einen fahrenden Jeep Cherokee durchgeführt wurde, wodurch der Hacker unter anderem Zugriff auf das Getriebe erhielt[8], zeigt jedoch deutlich, dass sich hinter all den Möglichkeiten und dem Komfort, die die neue Technologie bietet, auch neue Gefahren verbergen. Ein moderner Personenkraftwagen beinhaltet viele, zunächst als voneinander unabhängig erscheinende Komponenten, wie Sensoren oder Steuereinheiten. Durch die Kommunikation der einzelnen Entitäten miteinander entsteht ein komplexes Netz der Informationsverarbeitung, das Funktionen wie Erhöhung der Radiolautstärke bei schnellerem Fahren oder das Auslösen eines Alarms bei einem versuchten Einbruch erst möglich macht. Spätestens wenn das autonome Fahren zum neuen Standard der Fortbewegung wird, handelt es sich bei den Funktionen nicht länger nur um technische Spielereien. Die Garantie der Korrektheit der unter den Geräten ausgetauschten Informationen wird für die Sicherheit der Straßenverkehrsteilnehmer absolut notwendig sein. Erhält ein intelligentes Bremssystem ein elektrisches Aktivierungssignal, so muss fehlerfrei überprüft werden können, ob das Signal von einer vertrauenswürdigen Quelle mit den nötigen Berechtigungen, wie z.B. von einer zentrale Steuereinheit, stammt. Außerdem darf es einer dritten, unautorisierten Partei nicht möglich sein, die Nachricht unbemerkt zu manipulieren. Dieses Problem ist natürlich nicht nur bei diesem speziellen Beispiel der Fahrzeugtechnik,

sondern in allen möglichen Bereichen der IoT vorzufinden.

2.2 Grenzen aktueller kryptographischer Verfahren

In diesem Kontext besonders schützenswerte Ziele der Informationssicherheit sind also die Authentizität und die Integrität, die Unversehrtheit der Daten. Diese Ziele stehen jedoch im Konflikt mit der grundsätzlichen Tendenz der IoT, einzelne, größere und leistungsstärkere Einheiten durch mehrere, kleinere, schwächere und energieeffizientere Geräte zu ersetzen. Denn die Rechenleistung nimmt im Allgemeinen weiter zu, wodurch auch Angriffe auf bekannte Verschlüsselungs- und Signatur-Algorithmen immer wirksamer werden. Die Folge ist ein erhöhter Ressourcenaufwand, der benötigt wird, um den erwünschten Grad an Sicherheit aufrecht zu erhalten. Die schwachen Mikrocontroller wiederum können dabei nicht mithalten. Asymmetrische Verfahren der Kryptographie, die für Integrität und Authentizität grundsätzlich aufgrund des einfacheren Schlüsselaustauschs gegenüber ihren symmetrischen Gegenstücken vorzuziehen sind, sind von den neuen Anforderungen besonders betroffen. RSA, ein traditioneller, auf dem Faktorisierungsproblem basierender Algorithmus, braucht z.B. mit doppelter Schlüssellänge für die Generierung digitaler Signaturen sechs bis sieben mal länger.[9] Für die Zukunft wird also ein Verfahren der asymmetrischen Kryptographie benötigt, deren Ressourcenaufwand mit höheren Sicherheitsanforderungen weniger schnell steigt.

2.3 WalnutDSA als zukunftssicheres Verfahren digitaler Signaturen

Als eine mögliche Lösung dieses Problems veröffentlichte *SercureRF*, ein amerikanisches Unternehmen, das sich auf Kryptographie der IoT spezialisiert hat, 2016 mit *WalnutDSA* einen Algorithmus, der mithilfe von Braids, einer nicht abelschen mathematischen Gruppe, digitale Signaturen erstellt, die sich mit linearem Zeitaufwand verifizieren lassen. Darüber hinaus sei WDSA weder anfällig für Quantencomputer noch für effiziente Lösungen des *Conjugacy Search Problems*, der Schwachstelle älterer auf Braids basierende Algorithmen. Um den praktischen Wert von WDSA zu prüfen, wurde im Verlauf dieser Arbeit eine Implementierung für FreeRTOS-basierte Systeme entwickelt. Bevor auf die Details der Implementierung eingegangen wird, soll dem Leser mithilfe einer Einführung in die Braid-Theorie und einer anschließenden Erklärung der Teiloperationen des Verfahrens die Funktionsweise des Algorithmus veranschaulicht werden. In einem plattformübergreifenden Test mit zwei weiteren Systemen wurde anschließend versucht herauszufinden, ob sich WalnutDSA auf eingebetteten Systemen gut umsetzen lässt und eine Antwort auf die Frage zu finden, ob es sich bei der Braid-basierten Kryptographie um den Schlüssel für ein sicheres IoT der Zukunft handelt.

3 Einleitung

In den letzten Jahren, der Einsatz von Internet of Thing Geräten ist stark gewachsen. Wie das Wort Things beinhält dieser Begriff sehr breite Feld von Objekte, wie z.B. ein Smartarmbanduhren, eine netzfähige Kaffemashine oder auch verschiedene Messgeräte, die unterschiedleiche Gesundheitswerte überwachen. Es ist großer Vorteil, dass sie mit dem Netz verbunden sind und man z.B. mit gesammelten Daten arbeiten kann, aber gleichzeitig kann es ein Problem werden, da es um sensitiven Daten, wie z.B. aktueller Ort, handeln kann, die nicht von berechtigten Person gesehen werden dürfen und an den Nutzer von den Daten unverändert gesendet werden sollen. Auch bei Steuerung soll berechtigte Personen erfolgt werden. Dabei braucht man den Einsatz von Kryptographie auch bei IoT wie jede andere System.
Die IoT Geräten haben stark Beschränkung in ihren Rechenleistungen, Speicher im Vergleich zu den normalen Rechnern. Ein kryptographische Verfahren für ein IoT Gerät soll also das berücksichtigen. Zudem gibt es ein anderer wichtiger Aspekt. Der Verfahren muss sicher gegen Quantencomputern sein. Für die berühmte asymmetrische Verschlüsselungsalgorithmus RSA und ECDSA gibt es schon bekannte effiziente Angriffsalgorithmus, sodass sie nicht mehr sicher sind, wenn es ein Quantencomputer mit genügend Leistung entwicklt wird.
Im Paper [5] wird eine digitale Signatur Algorithmus Walnut-DSA vorgestellt, der die Aspekte berücksichtigen. Ein digitaler Signatur ist ein Verfahren, mit der der Empfänger von einer signierte Nachricht sicherstellen kann, dass die Nachricht von dem Sender unverändert zu ihm angekommen ist. Als dieser Verfahren wurde oft RSA, ECDSA und DSA, die sich auf Zahlentheorie basieren. Die Algorithmenen sind zwar gegen den klassischen Computern sicher, aber sie sind für die System mit sehr eingeschränkte Ressourcen nicht effizient und es gibt bereits bekannter Angriff, der in Zunkunft mit der Entwicklung von Quantencomputer ausführbar wird. Demgegenüber gibt es gegen Walnut-DSA noch kein bekannter effizienter Angriffsalgorithmus. Der größte Vorteil von Walnut-DSA ist, dass die Verifikation von Signatur auch mit sehr wenig Speicher und Rechenkraft möglich ist. In dem Paper wurde gezeigt, dass dieser Algorithmus um 40 Mal schnellere Laufzeit als ECDSA bei Verifizierung hat[5].
In dieser Arbeit geht es um Implementierung und Test von Walnut-DSA als ein Modul für den Betriebsystem RIOT nach dem Paper[5]. Der nächste Kapitel werden über die digitale Signatur, die Post-Quantum-Kryptographie und IoT, um den genannte Aspekte plausibler zu machen. Demnächst werden die Zopftheorie und E-Multiplikation beschrieben, auf der der Walnut-DSA basiert. Bevor man den Bestandteilen von Walnut-DSA erklären, wird noch die vergleichbare Algorithmen zu Walnut-DSA gezeigt. Im Kapitel Implementierung geht es um die konkretere Umsetzungsmöglichkeiten von Walnut-DSA. Hier wird beschrieben, für welche Umgebung der Code implementiert und getestet wurde und die umgesetzte Optimierungen betrachtet. In Diskussion wird zu erst die Testergebnisse von FreeRTOS, Linux und RIOT verglichen. Zudem wird die Sicherheit und unterschiedliche Umsetzbarkeit von Walnut-DSA diskutiert.

4 Background

Hier geht es um die grundlegende Probleme, an der der Walnut-DSA arbeiten.

4.1 Ziel von Cryptography

Neben der Vertrauchlichkeit, dass nur erlaubte Nutzer Daten lesen können, kann die Kryptographie für andere Ziele verwendet werden.
Integrität: Der Empfänger von Nachricht muss in der Lage sein, die empfange Daten zu überprüfen, ob es während Übertragung modifiziert wurde.
Authenzität: Der Empfänger kann verifizieren, dass Nachricht vom Sender kommt.
Nichtabstreitbarkeit: Der Sender von einer Nachricht kann nicht später verneinen, dass die Nachricht von ihm stammen.[10, S. 2-3]

4.2 Arten von Verschlüsselungsverfahren

Es gibt zwei Arten von Verschlüsselungsverfahren. Die beide Arten unterscheiden sich wegen ihre Schlüsseln. *Symmetrische Verschlüsselungsverfahren*: In dieser Art von Verfahren wird eine Nachricht mit Schlüssl k verschlüsselt und auch entschlüsselt. Dabei erfordet es, dass die beide Partei in der Kommunikation den Schlüssel bereits über einen sicheren Weg austauscht haben[11]. Die Nichtabstreitbarkeit kann es nur mithilfe ein dritte vertrauchte Partei geben[12]. *Asymmetrische Verschlüsselungsverfahren*: Bei dem asymmetrischen Verschlüsselungsverfahren werden unterschiedliche Schlüsseln, wobei ein Teil geheim bleiben und ein anderer Teil öffentlich wird. Bei Verschlüsselung von einer Nachricht, die nur vom Empfänger gelesen werden darf, wird der öffentlichen Schlüssel vom Empfänger verwendet. Da diese Nachricht nur mit dem privaten Schlüssel vom Empfänger entschlüsselbar ist, kann inklusiv der Sender niemand die Nachricht lesen. Dabei wird kein geheimer Schlüsselaustausch nötig.[11]

4.3 Digitale Signatur

Eine digitale Signatur ist eine kryptographische Transformation von Daten über asymmetrische Verschlüsselung. Im Gegensatz zum Fall, wo der Sender seine Nachricht mit dem öffentlichen Schlüssel vom Empfänger verschlüsselt, nutzt der Sender seine eigene private Schlüssel zu Verschlüsselung. Dabei kann man versichern, dass der Sender von der Signatur wirklich die Information anerkannt hat. Als Zusatzeffekt könnte man die digitale Signatur, dafür nutzen, um zu überprüfen, ob gesendete Daten nach Signierung modifiziert wurde. Mit der digitalen Signatur kann man also den orginalen Sender authentifizieren, die Integrität von Daten überprüfen und die Nichtabstreitbarkeit gewinnen.[13, S. 2, 9]
Die Abbildung 4.1 zeigt die Prozesse von digitalen Signaturen. Ein digitale Signatur Algorithmus enthält ein Signaturgenerierungsverfahren und ein Signaturverifikationsverfahren.

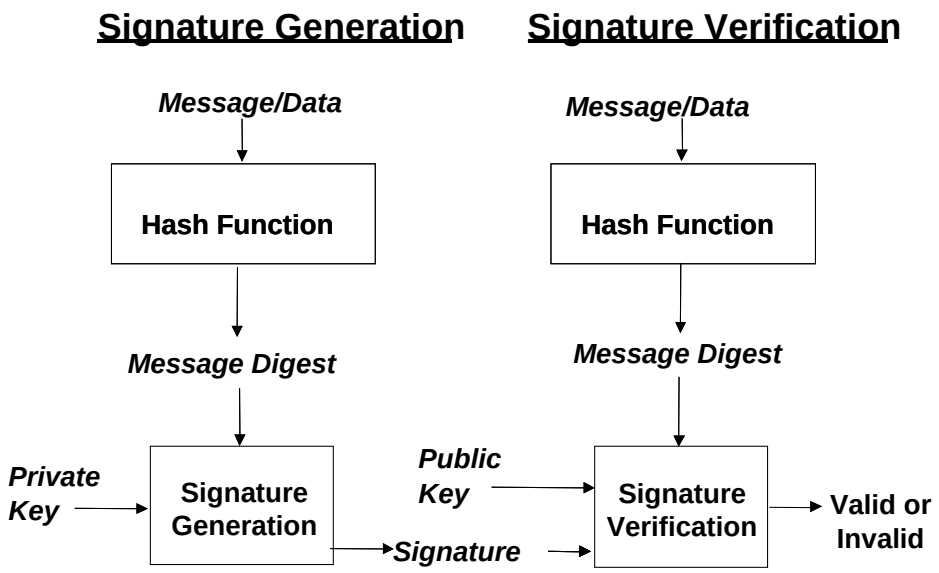


Abbildung 4.1: Digitalen Signatur Prozesse[13, S. 9]

Bei Generierung von Signatur wird ein privaten Schlüssel vom Sender benötigt. Bei Verifizierung von Signatur wird ein öffentlichen Schlüssel verwendet. Der öffentlichen Schlüssel müssen nicht geheim bleiben, aber die Integrität soll beibehalten werden. Der Empfänger verifiziert die orginale Nachricht und die zugehörige Signatur mithilfe der Signatur. Zudem braucht der Empfänger eine Garantie z.B. über ein Certificate Authority, dass der Sender seine vermeintliche Rechte tatsächlich besitzt. Sonst kann jeder, der irgendein mathematisch korrekte Schlüssel besitzt, kann sich für beliebige Identität signieren, aber der Empfänger hat keine Möglichkeit, diese zu überprüfen. [13, S. 9-10]

4.4 Post-Quanten-Kryptographie

Die klassische öffentlichen Schlüsselverfahren, wie z.B. RSA, basiert sich auf Schwierigkeiten von bestimmten zahlentheoretischen Problemen. Nun wurde es herausgefunden, dass die Quantencomputer mehere zahlentheoretische Probleme exponentiell schneller als den klassischen Computern berechnen können. Dazu fallen Faktorisierung, diskreter Logarithmus, pellsche Gleichung und Berechnung von Einheitengruppe und Idealklassengruppe eines algebraischen Zahlenkörpers[14, S. 17].
Die Abbildung 4.2 zeigt welche Kryptosysteme über einem Quantumalgorithmus gebrochen sind. Die Probleme, die mit Quantencomputer effizient lösbar sind, lassen sich mit dem HSP(Hidden Subgroup Problem) erklären, die eine Generalisierung von Shors Faktoriserung und diskrete Logarithmenalgorithmus ist. Als das HSP muss man für eine gegebene Gruppe und eine Funktion, die konstant und distinkt zu Nebenklassen von einer unbekannten Subgruppe ist, eine Menge von Generatoren für die Subgruppe finden. Es ist dabei entscheidend, um welche Gruppe ein Problem handelt. Ein wichtiger Faktor ist, ob die gegebene Gruppe abelsch ist.
In der Abbildung 4.3 und 4.4 sieht man verschiedene Gruppen und deren Bezug zu Quantenalgorithmen. Bis auf R^n Gruppe gibt es bereits Quantenalgorithmus, wobei der Algorith-

Cryptosystem	Broken by Quantum Algorithms?
RSA public key encryption	Broken
Diffie-Hellman key-exchange	Broken
Elliptic curve cryptography	Broken
Buchmann-Williams key-exchange	Broken
Algebraically Homomorphic	Broken
McEliece public key encryption	Not broken yet
NTRU public key encryption	Not broken yet
Lattice-based public key encryption	Not broken yet

Abbildung 4.2: Aktuelle Sicherheit von klassische Kryptosystem im Bezug zu Quantencomputern[14, S. 16]

Abelian Group G	Associated Problem	Quantum Algorithm?
$\mathbb{Z}_2^n$		Yes
The integers $\mathbb{Z}$	Factoring	Yes
Finite groups	Discrete Log	Yes
The reals $\mathbb{R}$	Pell's equation	Yes
The reals $\mathbb{R}^c$, c a constant	Unit group of number field	Yes
The reals $\mathbb{R}^n$, n arbitrary	Unit group, general case	Open

Abbildung 4.3: Abelsche Gruppen und HSP[14, S. 25]

mus ab R sehr komplex wird, da die Gruppen nicht mehr abzählbar sind. Für die nichtabelschen Gruppe stehen noch viele offene Fragen, ob sie mithilfe eines Quantenalgorithmus lösbar ist[14, S. 25-29].

4.5 Internet der Dinge

Der Term Internet der Dinge(Kurzgeschrieben *IoT* aus englischen *Internet of Things*) ist die Generalisierung von Geräte, die mit dem Internet verbindbar sind. Typischerweise bezieht man mit IoT Sensoren in Automobile, im Bereich Medizin oder auch z.B. alle Geräte, die mit dem Begriff Smarthome einbezogen werden, die klein und wenig Rechleistung und Speicher zur Verfügung haben[15, S. 113]. In der Regel sind die IoT Geräte nicht nur in der Lage, sich mit dem Netzwerk zu verbinden, sonderen auch sensorische Daten zu sammeln und/oder ihr Umgebung zu modifizieren. Im Gegensatz zu der Kommunikation zwischen RFID Leser und Objekt können IoT Geräte sowohl aktive als auch passive Rolle in der Kommunikation übernehmen[15].
Die grundlegende Sicherheitsprobleme bei den normalen System gelten auch bei IoT. In IoT kommt dazu noch Problem mit beschränkte Rechenleistung und Speicher. Und falls ein Gerät mit einer Batterie getrieben wird, muss man auch auf die Energieeffizienz achten. Außerdem muss man sicherstellen können, wie zuverlässig Geräte, die auch oft verteilt sind, physisch verbunden sind. Im Extremfall kann es passieren, dass ein Komponent plötzlich ausgeht, da der Nutzer woanders den Netzkabel braucht hat und den Kabel deshalb ohne große Achtung rauszieht[16]. Neben den physische Probleme, gibt es Probleme, wie man ein zuverlässig ein IoT Gerät identifiziert und wie man eine sichere Verbindung baut. Einer von Lösungsvorschläge ist, dass man die MAC Addresse von jeweilige Geräte nutzt. Diese kann

Nonabelian Group G	**Associated Problem**	**Quantum Algorithm?**
Heisenberg group		Yes
$\mathbb{Z}_p^r \rtimes \mathbb{Z}_p$, r constant		Yes
$\mathbb{Z}_p^n \rtimes \mathbb{Z}_2$, p a fixed prime		Yes
Extraspecial groups		Yes
$\downarrow$?		
Dihedral group $D_n = \mathbb{Z}_n \rtimes \mathbb{Z}_2$	Unique shortest lattice vector	Subexponential-time
Symmetric group S_n	Graph isomorphism	Evidence of hardness

Abbildung 4.4: nichtabelsche Gruppen und HSP[14, S. 26]

aber von einem bösartigen Sender verfälscht werdem und IoT System hat keinen Weg, dabei zu überprüfen, ob die Nachricht von den legitime Nutzer(in diesem Fall auch ein Gerät) kommt. Eine andere Variante ist Einsatz von asymmetrische Verschlüsselungsverfahren. Das Problem dabei ist, dass derartige Verfahren oft große Menge von Ressourcen verbraucht, die bei den IoT Geräten eher wenig vorhanden ist. [15].

5 Braids

Die Zopftheorie geht auf den österreichischen Mathematiker Emil Artin zurück, welcher sie mit seiner 1925 erschienenen Arbeit *Theorie der Zöpfe* [17] ins Leben rief. Die Theorie lässt sich in den mathematischen Teilbereich der Topologie einordnen und beschäftigt sich sowohl mit dem geometrischen Aufbau von Zöpfen, als auch mit ihren zugrundeliegenden algebraischen Eigenschaften.

In der Kryptographie hat die Zopftheorie erst recht spät Anklang gefunden; die ersten Arbeiten erschienen 1999 [18] und 2000 [19]. Daher handelt es sich bei der Braid-basierten Kryptographie um ein noch recht neues Konzept, welches jedoch durch die Abweichung von den traditionellen mathematischen Problemen kryptographischer Verfahren als auch durch die intuitive Verständlichkeit der Zopftheorie einen interessanten Kandidaten für moderne und zukunftsträchtige Kryptosysteme darstellt.

Im Folgenden soll eine kleine Einführung in die Zopftheorie gegeben werden, um die mathematischen Grundlagen für WalnutDSA zu etablieren. Für eine detailliertere Ausarbeitung siehe bspw. *An Introduction to Braid Theory* [20], woran sich folgendes Kapitel anlehnt. Aufgrund der überwiegend englischsprachigen Literatur in der Braid-basierten Kryptographie wird im Rahmen dieser Arbeit neben dem deutschen Begriff ,,Zopf" auch der englische Begriff ,,Braid" verwendet werden. Beide Terminologien haben jedoch dieselbe Bedeutung.

5.1 Grundlagen der Braid-Theorie

Ein Zopf bzw. Braid ist eine Verflechtung aus mehreren Strähnen im dreidimensionalen Raum. Sei der Raum ein Würfel und nehme man n Strähnen Schnur, so müssen vier Eigenschaften erfüllt sein, sodass es sich dabei um einen sog. *n-Braid* mit n Strähnen handelt [20, 5]:

1. Keine Strähne liegt ganz oder teilweise außerhalb des Würfels.
2. Jede Strähne beginnt an der Oberseite des Würfels und endet an dessen Unterseite.
3. Zwei verschiedene Strähnen können sich nicht überschneiden, sonder nur nebeneinander bzw. vor- oder hintereinander vorbei laufen.
4. Die Strähnen bewegen sich kontinuierlich nach unten, sodass kein Teil horizontal oder nach oben verläuft.

Abbildung 5.1 zeigt links einen 3-Braid; die rechte Konstruktion ist kein Braid, da die vierte Eigenschaft nicht erfüllt wird.

Es existiert eine mögliche Äquivalenz zwischen mehreren solchen n-Braids mit n Strähnen. Ein n-strähniger Braid b ist äquivalent zu einem weiteren b', wenn die einzelnen Strähnen

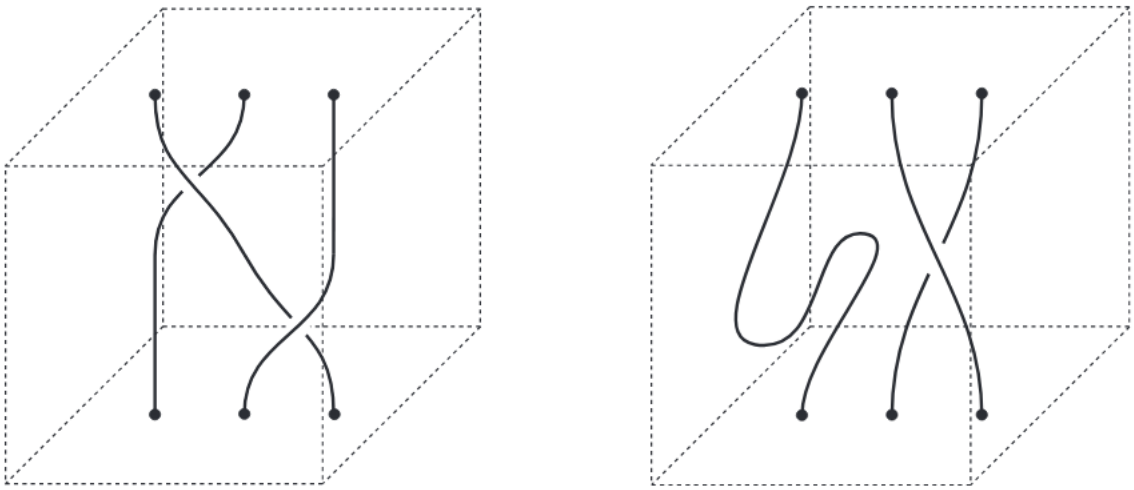


Abbildung 5.1: Links ein 3-Braid, rechts kein Braid. Quelle: [20].

von b so ,,verschoben" werden können, dass der Braid b' entsteht. Dabei müssen jedoch stets die vier besagten Eigenschaften erfüllt sein, und es darf keine Strähne durchgeschnitten und wieder zusammengeklebt oder ihr Anfangs- oder Endpunkt im Würfel verschoben werden. Abbildung 5.2 zeigt zwei äquivalente 2-Braids.

Zur einfacheren Veranschaulichung kann die dreidimensionale Projektion sowie der Würfel selbst weggelassen werden und ein Braid auf einer Ebene dargestellt werden. Hierbei sollen vermeintliche Überschneidungen so gesehen werden, dass eine durchlaufende Strähne über einer durchtrennten verläuft. Abbildung 5.3 zeigt die zweidimensionale Darstellung des 3-Braids aus Abbildung 5.1.

Die essentielle Operation in den Zopfgruppen ist die Multiplikation. Zwei n-Braids b und b' können multipliziert werden, indem man b' an das untere Ende von b anhängt und dabei die Enden der Strähnen verschmelzt. Abbildung 5.4 zeigt das Produkt aus zwei 3-Braids.

Die Äquivalenzklassen aller Braids mit n Strähnen, die obige Eigenschaften erfüllen, bilden die Zopfgruppe B_n. Die Verknüpfung ist hierbei die Multiplikation. Das Einselement ist der n-Braid mit ausschließlich geraden Strähnen ohne Überkreuzungen. Die Inverse eines n-Braids ist dessen Spiegelung am horizontalen unteren Ende. Abbildung 5.5 zeigt wieder den bekannten 3-Braid mit seiner Inversen.

5.2 Artingeneratoren und Braid-Permutation

Eine mögliche Repräsentierung von Braids besteht in Form von *Artingeneratoren*. Dies sind die einfachsten n-Braids in der Zopfgruppe B_n, abgesehen vom trivialen Braid. Geometrisch ausgedrückt ist ein Artingenerator b_i mit $1 < i < n$ für die Zopfgruppe B_n der Braid, bei dem sich nur die i-te und $i+1$-te Strähne überkreuzen, wobei die i-te Strähne unter der $i+1$-ten verläuft. Bei der Inversen dieses Generators, b_i^{-1}, verläuft die i-te Strähne über der $i+1$-ten Strähne. Abbildung 5.6 zeigt die Darstellung der Artingeneratoren einer Zopfgruppe

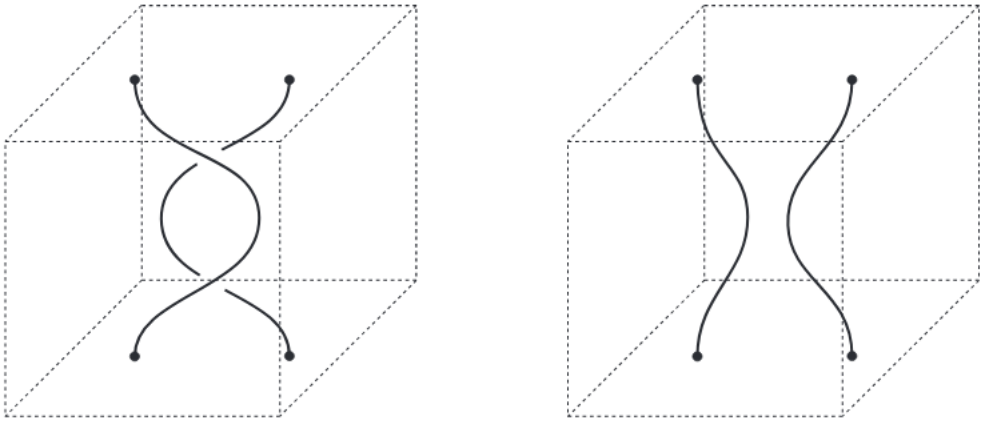


Abbildung 5.2: Zwei äquivalente 2-Braids. Quelle: [20].

Abbildung 5.3: Zweidimensionale Darstellung eines 3-Braids.

Abbildung 5.4: Rechts das Produkt aus dem linken und mittleren 3-Braid. Quelle: [20].

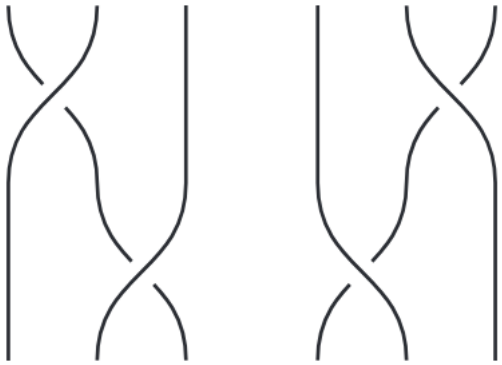


Abbildung 5.5: Rechts die Inverse des linken 3-Braids.

Abbildung 5.6: Verallgemeinerte Darstellung eines Artingenerators (links) und seiner Inversen (rechts). Quelle: [20].

B_n. Jeder Braid kann in diese Artingeneratoren der entsprechenden Zopfgruppe aufgeteilt und somit auch aus ihr Erschaffen werden. Abbildung 5.4 zeigt die Zusammensetzung des 3-Braids $b_1 b_2$ aus den zwei Artingeneratoren b_1 und b_2.

Es existieren zwei fundamentale Relationen in dieser Repräsentierung von Braids. Für die Menge an Artingeneratoren $b_1, b_2, ..., b_{n-1}$ gilt:

$$b_i b_{i+1} b_i = b_{i+1} b_i b_{i+1} \quad \text{für } i = 1, ..., n-2, \tag{5.1}$$

$$b_i b_j = b_j b_i \quad \text{für } |i-j| >= 2. \tag{5.2}$$

Mithilfe dieser lassen sich jegliche zwei äquivalente Braids einer Zopfgruppe ineinander ,,umwandeln" (s. wiederum Abbildung 5.2). Abbildung 5.7 zeigt die erste, Abbildung 5.8 die zweite der beiden Operationen.

Artingeneratoren sind die entscheidende Repräsentierung von Braids bei WalnutDSA. Für die Implementierung des Verfahrens lassen sie sich offensichtlich sehr einfach als ganze Zahlen speichern, sodass sich ein Braid als Liste aus Artingeneratoren darstellen lässt, aus denen er aufgebaut wird. Ein 4-Braid bestehend aus den Artingeneratoren $b_1 b_3 b_2^{-1} b_3 b_1^{-1}$ ließe sich bspw. einfach als Zahlenfolge 1 3 −2 3 −1 kodieren.

Eine weitere für WalnutDSA wichtige Abbildung in der Zopfgruppe ist die der Braid-Permutation σ. Je nach Überkreuzungen enden verschiedenen Strähnen an unterschiedlichen Enden eines Braids. Dies kann als Permutation dargestellt werden, welche die An-

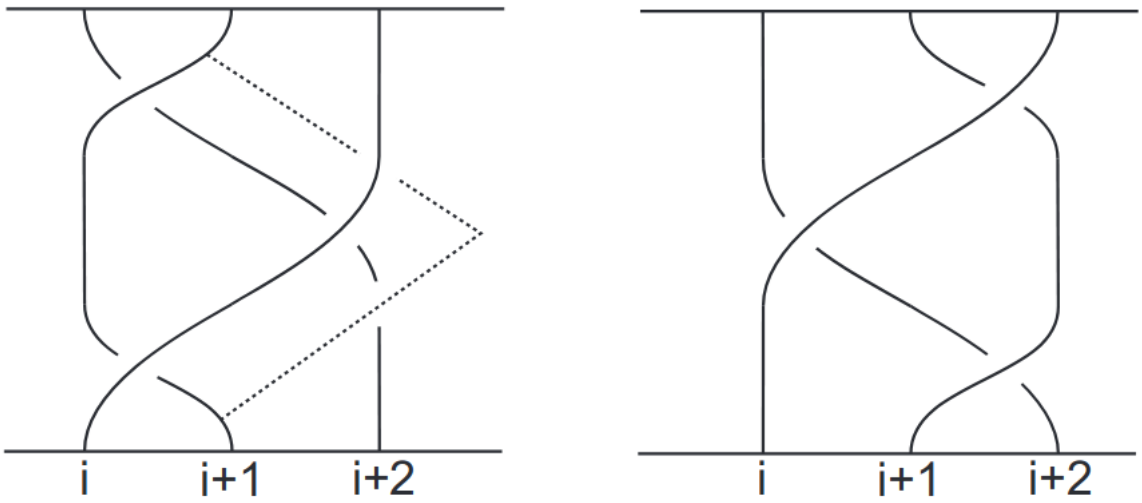


Abbildung 5.7: Darstellung der Relation 5.1. Quelle: [20].

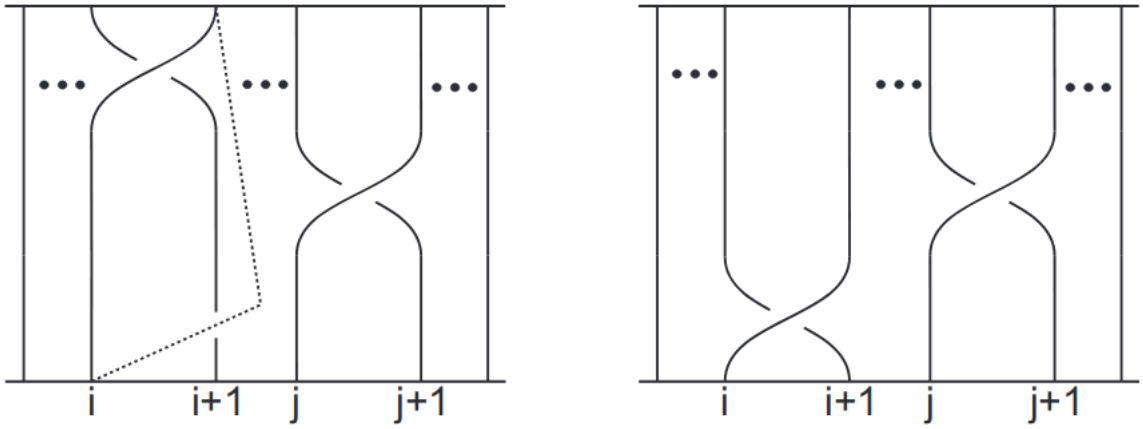


Abbildung 5.8: Darstellung der Relation 5.2. Quelle: [20].

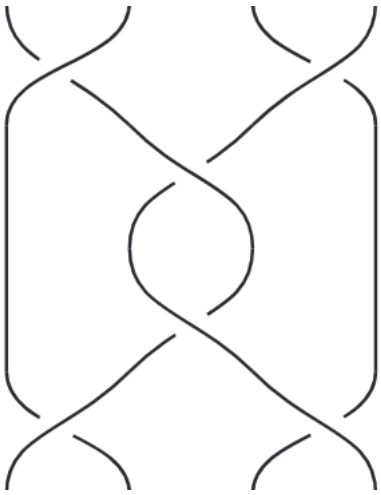


Abbildung 5.9: Darstellung eines reinen 4-Braids. Quelle: [20].

fangsposition einer Strähne auf ihre Endposition im Braid, jeweils gegeben als Index i mit $1 \leq i \leq n$, abbildet. Somit existiert für jeden Braid eine zugrundeliegende, jedoch nicht eindeutige Braid-Permutation.

Es besteht ein einfacher Zusammenhang zwischen der Permutation eines Braids und den Artingeneratoren, aus denen er zusammengesetzt ist. Sowohl für einen Artingenerator b_i als auch dessen Inverse b_i^{-1} werden die Strähnen mit den Indizes i und $i+1$ vertauscht; somit werden auch bei der Braid-Permutation diese beiden Elemente ausgetauscht. Beginnend bei der trivialen Permutation kann so die Braid-Permutation berechnet werden, indem man fortlaufend die Vertauschungen der entsprechenden Artingeneratoren auf die Permutation anwendet. Die Permutation des 3-Braids aus Abbildung 5.3 bzw. $b_1^{-1}b_2$ lautet bspw. wiefolgt:

$$\sigma(b_1^{-1}b_2) = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}$$

Auch die Braid-Permutation lässt sich einfach durch ein normales Array aus natürlichen Zahlen repräsentieren, wobei der Wert an der Stelle i mit $0 \leq i < n$ mit der Permutation der $i+1$-ten Strähne korrespondiert.

5.3 Untergruppe der reinen Braids

Ein Braid ist ein *reiner* Braid, wenn seine zugrundeliegende Permutation trivial ist, wenn also die Anfangs- und Endposition jeder Strähne gleich ist. Diese Braids bilden die Untergruppe der reinen Braids P_n, welche offensichtlich in der normalen Zopfgruppe enthalten ist. Abbildung 5.9 zeigt einen reinen 4-Braid.

Für diese Untergruppe existieren eigene Generatoren, welche selbst wiederum aus Artingeneratoren bestehen. Sie werden in mehreren Teilprozeduren von WalnutDSA benutzt und können zur einfachen Generierung von reinen Braids verwendet werden. Ein Generator $p_{i,j}$ ist nach folgendem Schema aufgebaut:

$$p_{i,j} = b_{j-1}b_{j-2}\dots b_{i+1}b_i^2b_{i+1}^{-1}\dots b_{j-2}^{-1}b_{j-1}^{-1} \quad \text{für } 1 \leq i < j \leq n \tag{5.3}$$

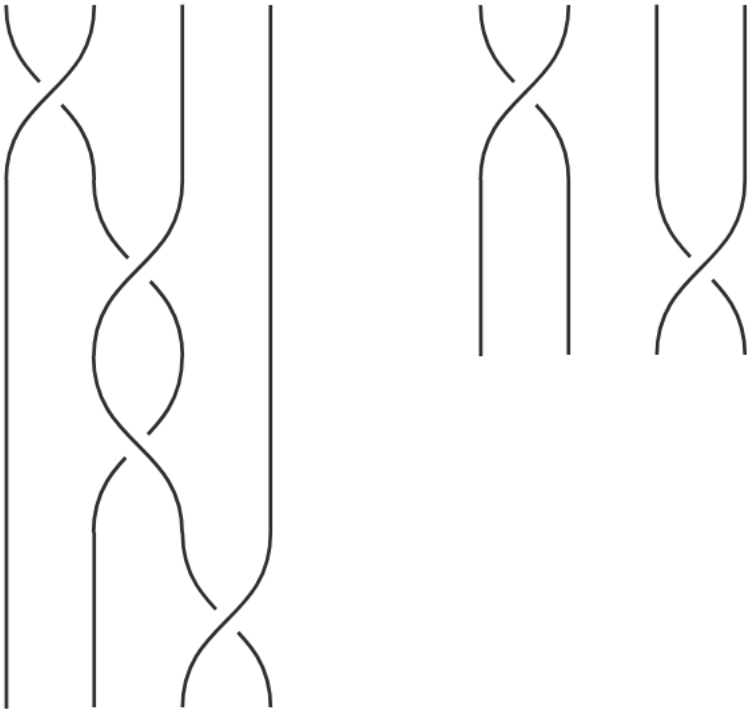


Abbildung 5.10: Rechts die frei-reduzierte Version des linken 4-Braids.

5.4 Freie Reduktion

Das Produkt eines Artingenerators und seiner Inversen ist das Einselement, sprich der triviale Braid, welcher bei der Repräsentierung durch Artingeneratoren der Nicht-Existenz eines Generators gleich kommt. Daher werden bei einer solchen Operation die beiden entsprechenden Generatoren ,,ausgelöscht". Man beachte, dass dies auch nur dem Anwenden der beiden fundamentalen Operationen 5.1 und 5.2 gleichkommt. Besteht ein Braid aus noch weiteren Artingeneratoren als den beiden Operanden der Multiplikation, so wird er dadurch verkürzt. Da der resultierende Braid immer noch äquivalent zum alten ist und die zugrundeliegende Permutation gleich bleibt, handelt es sich hierbei gewissermaßen um eine Reduktion. Diese Form der Reduktion wird *freie Reduktion* genannt. Ein Braid gilt als *frei-reduziert*, wenn in seiner Repräsentierung durch Artingeneratoren keine Vorkommnisse der Form $b_i b_i^{-1}$ bzw. $b_i^{-1} b_i$ enthalten sind. Abbildung 5.10 zeigt die freie Reduktion eines 4-Braids.

6 E-Multiplikation

Die E-Multiplikation ist eine der Kernoperationen des WalnutDSAs, mit deren Hilfe der öffentliche Schlüssel berechnet und die Verifizierung der Signatur durchgeführt wird. Bevor ihre Funktionsweise näher behandelt werden kann, wird neben der Braidtheorie auch Wissen aus anderen mathematischen Bereichen benötigt.

6.1 Braids als Colored-Burau-Matrizen

Bei einer *Colored-Burau-Matrix* handelt es sich um die beste bekannte Möglichkeit, Braids linear darzustellen.[21, p. 49] Sei $GL_n\left(\mathbb{Z}\left[t^{\pm t}\right]\right)$ die Gruppe invertierbarer $n \times n$ Matrizen über Polynome eines Laurent-Rings $\mathbb{Z}\left[t^{\pm t}\right]$ $\{f(t_1), ..., f(t_N)\}$, dann beschreibt die *reduzierte Colored-Burau-Repräsentation* einen Homomorphismus $B_N \to GL_N\left(\mathbb{Z}\left[t^{\pm t}\right]\right)$, der Braids der Ordnung N auf Matrizen abbildet. Ein Artin-Generator b_i lässt sich wie folgt darstellen [21, p. 50]:

$$CB(b_i^{+1}) = \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & t_i & -t_i & 1 & & \\ & & & \ddots & & \\ & & & & & 1 \end{pmatrix} \qquad CB(b_i^{-1}) = \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & 1 & -t_{i+1}^{-1} & t_{i+1}^{-1} & & \\ & & & \ddots & & \\ & & & & & 1 \end{pmatrix} \tag{6.1}$$

Die t-Einträge befinden sich in der i-ten Zeile. Bei $i = 1$ fällt dabei der linke Eintrag weg. Der Artin-Generator b_i hat die Eigenschaft, den i-ten Strang des Braids mit den $i+1$-ten zu vertauschen. Assoziiert man mit jedem Generator eine Permutation aus der symmetrischen Gruppe mit N Elementen $\sigma_i \in S_N$ auf die N Variablen $\{t_1, ..., t_N\}$, lassen sich Braids mit mehreren Generatoren durch die Multiplikation der Tupel $(CB\left(b_i\right), \sigma_i)$ berechnen.[22, p.4]

$$\left(\begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & t_i & -t_i & 1 & & \\ & & & \ddots & & \\ & & & & & 1 \end{pmatrix}, \sigma_i\right) \quad \circ \quad \left(\begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & t_j & -t_j & 1 & & \\ & & & \ddots & & \\ & & & & & 1 \end{pmatrix}, \sigma_j\right)$$

$$= \left(\begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & t_i & -t_i & 1 & \\ & & & \ddots & \\ & & & & 1 \end{pmatrix} * \begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & t_{\sigma_i(j)} & -t_{\sigma_i(j)} & 1 & \\ & & & \ddots & \\ & & & & 1 \end{pmatrix}, \sigma_i \sigma_j \right) \tag{6.2}$$

6.2 Arithmetik in Binärkörpern

Mit wachsender Länge des Braids wachsen auch die Laurent-Polynome der CB-Matrizen. Die Matrixeinträge und die Rechenoperationen werden schnell sehr komplex. Für den Einsatz auf ressourcenarmen Systemen, wird daher eine algebraische Struktur benötigt, die die CB-Einträge für effiziente Rechenoperationen möglichst klein hält.

Endliche Körper, oder auch *Galois-Körper* genannt, beinhalten endlich viele, positive ganze Zahlen und erfüllen die Anforderungen an mathematische Körper bzgl. Addition und Multiplikation. Die Anzahl der Elemente q (Ordnung) ist hierbei eine ganzzahlige Potenz einer Primzahl p. In der Informatik sind vor allem *Galois-Körper* der Charakteristik $p = 2$ interessant, da ein *Galois-Körper* $GF(2^m)$ alle Binärzahlen mit m Bits beinhaltet. Diese werden auch Binärkörper genannt. Die Elemente eines Binärkörpers lassen sich als Polynome mit den Koeffizienten 0 und 1 und einem Grad $g < m$ darstellen.[23, pp. 25-26]

6.2.1 Addition im Binärkörper

Die Addition im Binärkörper ist die Addition der Polynome mit anschließenden *modulo* 2 auf die Koeffizienten und gleicht damit einer bitweisen Addition ohne Übertrag bzw. einer XOR-Operation.

Beispiel im $GF(2^4) : 7 + 9 = 0111 \oplus 1001 = 1110 = 14$

Jedes $a \in GF(2^m)$ besitzt ein Inverses bzgl. der Addition $(-a)$, sodass $a + (-a) = 0$. Da es sich bei der Addition um die XOR-Operation handelt, ist das Inverse von a gleich a selbst. [23, 26-27]

6.2.2 Multiplikation im Binärkörper

Die Multiplikation im Binärkörper ist definiert durch $a * b \, modulo \, x$ mit dem irreduziblen Polynom x, das sich nicht aus einem Produkt aus Polynomen eines geringeren Grades als m faktorisieren lässt. [23, p. 28]

Beispiel im $GF(2^8) : 50{*}40 = (z^5{+}z^4{+}z){*}(z^5{+}z^3) \, mod \, (z^8{+}z^4{+}z^3{+}z{+}1) = z^7{+}z^4{+}1 = 145$

Analog zur Addition besitzt auch jedes Element $a \in GF(2^m)$ ein Inverses bzgl. der Multiplikation a^{-1}, sodass $a * a^{-1} = 1$.

Die Subtraktion erfolgt über eine Addition mit dem Inversen bzgl. der Addition und die Division über die Multiplikation mit dem Inversen bzgl. der Multiplikation:[23, pp. 25-26]

$$a/b = a * b^{-1} \qquad a - b = a + (-b)$$

6.3 E-Multiplikationsschritt

Die E-Multiplikation nimmt als Eingabe ein Tupel (M, σ) mit $M = CB(\beta)$ und $\beta \in B_N$. Die Operation erfolgt schrittweise für jeden Generator in β. Das Ergebnis ist wieder ein Tupel (M, σ)[22, p. 6].

$$(M, \sigma_0) \star (\beta, \sigma_\beta) = \left(\left((M, \sigma_0) \star \left(b_{i_1}^{\in_1}, \sigma_{\beta_1}\right)\right) \star \left(b_{i_2}^{\in_2}, \sigma_{\beta_2}\right)\right) \star \cdots \star \left(b_{i_k}^{\in_k}, \sigma_{\beta_k}\right) \tag{6.3}$$

Um das Bilden komplexer Einträge zu vermeiden werden die Variablen $t_1 ... t_N$ durch positive Elemente eines Binärfelds der Ordnung q ersetzt, die sogenannten „$T - Werte$“ . Da die Arithmetik nun im Binärkörper stattfindet, ist das negative Vorzeichen der Colored-Burau-Darstellung der Artin-Generatoren vernachlässigbar. Bei negativen Generatoren ist es notwendig, das Inverse bezüglich der Multiplikation des dazugehörigen T-Value zu bestimmen.

$$CB(b_i^{+1}) = \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & t_i & t_i & 1 & & \\ & & & \ddots & & \\ & & & & & 1 \end{pmatrix} \qquad CB(b_i^{-1}) = \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & 1 & t_{i+1}^{-1} & t_{i+1}^{-1} & & \\ & & & \ddots & & \\ & & & & & 1 \end{pmatrix} \tag{6.4}$$

Die iterative Multiplikation entspricht der CB-Multiplikation 6.2, bis auf den Unterschied dass die Addition und die Multiplikation durch ihre Pendants aus der Galois-Arithmetik ersetzt werden. D.h. der Eintrag z in Zeile i, Spalte j wird mit dem irreduzibles Polynom zu $GF(2^m)$ x berechnet durch:

```
for i ← 1 to N do
    for j ← 0 to N do
        result ← 0; for i ← 0 to N do
            result ← result ⊕ (a[i][k] * b[k][j] modulo x);
        end
    end
end
```

Der Ressourcenaufwand der E-Multiplikation wächst aufgrund der iterativen Abarbeitung aller Artin-Generatoren linear mit der Braidlänge. Der benötigte Speicherplatz der Matrix ist lediglich abhängig von N und q. Als zentraler Bestandteil der Verifizierung ist

die E-Multiplikation daher ein geeignetes Mittel für die Kryptographie auf ressourcenarmen Systemen.

6.4 Beispiel einer E-Multiplikation

gegeben: $N = 4$; $\beta_1 = \epsilon$; $\beta_2 = \{3, -1\}$; T-Werte = $\{14,\ 1,\ 8,\ 1\}$

$$\left(\left(\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}, (1,2,3,4)\right)\right) \star \left(\{3,-1\}, \sigma_{\{3,-1\}}\right)$$

$$= \left(\left(\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}, (1,2,3,4)\right)\right) \star (\{3\}, \sigma_3) \star (\{-1\}, \sigma_{-1})$$

$$= \left(\left(\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix} *^{(1,2,3,4)} \begin{pmatrix}1&0&0&0\\0&1&0&0\\0&t_3&t_3&1\\0&0&0&1\end{pmatrix}, \sigma_3\right)\right) \star (\{-1\}, \sigma_{-1})$$

$$= \left(\left(\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix} * \begin{pmatrix}1&0&0&0\\0&1&0&0\\0&t_3&-t_3&1\\0&0&0&1\end{pmatrix}, \sigma_3\right)\right) \star (\{-1\}, \sigma_{-1})$$

$$= \left(\left(\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&t_3&t_3&1\\0&0&0&1\end{pmatrix}, (1,2,4,3)\right)\right) \star (\{-1\}, \sigma_{-1})$$

$$= \left(\left(\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&t_3&t_3&1\\0&0&0&1\end{pmatrix} *^{(1,2,4,3)} \begin{pmatrix}t_2^{-1}&t_2^{-1}&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}, \sigma_3 * \sigma_{-1}\right)\right)$$

$$= \left(\left(\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&t_3&t_3&1\\0&0&0&1\end{pmatrix} * \begin{pmatrix}t_2^{-1}&t_2^{-1}&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}, (2,1,4,3)\right)\right)$$

$$= \left(\left(\begin{pmatrix}t_2^{-1}&t_2^{-1}&0&0\\0&1&0&0\\0&t_3&t_3&0\\0&0&0&1\end{pmatrix}, (2,1,4,3)\right)\right) = \left(\left(\begin{pmatrix}1&1&0&0\\0&1&0&0\\0&8&8&0\\0&0&0&1\end{pmatrix}, (2,1,4,3)\right)\right)$$

7 Stand der Forschung

Die beschränkte Rechenleistung und Speicherkapazität eingebetteter Systeme stellt neue Herausforderungen an digitale Signaturverfahren. Zudem ist durch die fortschreitende Forschung an Quantencomputern die Resistenz gegen quantenbasierte Angriffe ein neues Ziel in der Kryptographie. Es existieren neben WalnutDSA weitere Verfahren, die diese Aspekte abdecken wollen. Auf einige dieser Kryptosysteme soll nun eingegangen werden.

7.1 Elliptische-Kurven-Kryptographie

Auf elliptischen Kurven beruhende Kryptosysteme wie ECDSA übertragen das schwere Problem des diskreten Logarithmus auf die Gruppe der Punkte einer elliptischen Kurve. Dadurch kommen sie mit weitaus kleineren Schlüsseln zurecht, benötigen jedoch bei der Verifizierung von Signaturen mehr Rechenzeit. So korrespondiert z.B. ein 224-Bit ECDSA-Schlüssel bzgl. der Sicherheit mit einem 2048-Bit RSA-Schlüssel [24, 25]. Die tatsächliche Leistung des Algorithmus hängt von der gewählten elliptischen Kurve ab. Bei der Signaturgenerierung ist das Verfahren mit traditionellem RSA weitestgehend gleichauf, wobei die Verifizierung von RSA unabhängig von der Schlüssellänge um mehrere Größenordnungen schneller ist [25].

Verfahren auf elliptischen Kurven sind genau wie RSA auch anfällig für Angriffe durch Quantencomputer. Der *Shor-Algorithmus* [26] bietet eine Möglichkeit, die üblicherweise schweren Probleme der Primfaktorzerlegung und des diskreten Logarithmus effizient mit einem Quantencomputer zu lösen. Daher sind besagte kryptographische Systeme nicht quantenresistent.

7.2 Gitter-basierte und RLWE-Kryptographie

Gitter-basierte Kryptosysteme ziehen ihre Sicherheit aus schweren Berechnungsproblemen in mathematischen Gittern. Bei Gittern handelt es sich um regelmäßigen Mengen aus Vektoren des euklidischen Vektorraums. Eine typische Einwegfunktion dieser Kryptosysteme ist das *Shortest Vector Problem*, welches den kürzesten Vektor in einem Gitter sucht. Diese Verfahren sind vielversprechende Lösungen zur Post-Quanten-Kryptographie, da sie nicht auf den Problemen der Primfaktorzerlegung und des diskreten Logarithmus beruhen, sondern einen davon unabhängigen, neuen Ansatz verfolgen. Effiziente Lösungen, die die Sicherheit dieser Verfahren schwächen würden, wurden bisher noch nicht gefunden. Zudem bietet die Familie der Gitter-basierten Verschlüsselungssysteme theoretisch sowohl kleine Schlüssellängen als auch eine effiziente Verarbeitung [27].

Die *Ring learning with errors signature (RLWE-SIG)* ist ein Gitter-basiertes digitales Signaturverfahren. Es existiert eine Implementierung für FPGAs, die effizienter als RSA arbeitet, jedoch weitaus größere Schlüssellängen benötigt. In der Arbeit der Autoren werden ca. 12000 Bits für den öffentlichen und 2000 Bits für den privaten Schlüssel verwendet, um vergleichbare Sicherheit zu einem 1024-Bit RSA-Schlüssel zu erzeugen. Wie stark die

Schlüssellängen mit steigendem Sicherheitslevel wachsen, wurde nicht untersucht. Das Verfahren arbeitet jedoch um den Faktor $1,5$ schneller als eine vergleichbare Implementierung von RSA [28].

7.3 Weitere Braid-basierte Kryptosysteme

Neben WalnutDSA gibt es weitere digitale Signaturverfahren, die auf Zopfgruppen basieren. Die gängige Einwegfunktion dieser Kryptosysteme ist das Konjugationsproblem in einer Zopfgruppe. Hierfür ist bislang keine effiziente Lösung – weder für Quantencomputer noch für das klassische Rechenmodell – gefunden worden. Zudem werden durch die Repräsentierung der Signaturen durch Braids gängige mathematische Angriffsvektoren ausgeschalten [29].

Das *Braid Signature Scheme (BSS)* ist ein anderes digitales Signaturverfahren auf Basis der Zopfgruppen. Es existiert eine Implementierung der Autoren, bei der Schlüssellängen von 370–591 Bit verwedet wurden, um angemessene Sicherheit zu erreichen. Die Längenangaben sind jedoch grobe Schätzungen und es sind keine ausreichenden Daten vorhanden, um ein klares Bild über die Leistungsanforderungen des Algorithmus zu erhalten [30].

8 WalnutDSA - Bestandteile

Es sollen nun die einzelnen Teilprozeduren von WalnutDSA dargestellt und auf relevante Aspekte bei der generischen Implementierung eingegangen werden. Zum Zeitpunkt der Veröffentlichung der vorliegenden Arbeit sind aufbauend auf die ursprüngliche Fassung zwei aktualisierte Versionen der wissenschaftlichen Arbeit von WalnutDSA erschienen. Die im Folgenden behandelten Definitionen des Verfahren richten sich nach der ersten Aktualisierung vom 18. September 2017. Es soll hierbei zunächst auf die öffentlichen Informationen und Sicherheitsparameter sowie die Schlüsselgenerierung eingegangen werden. Anschließend wird die Kodierungsfunktion für den Nachrichtenhashwert erläutert und die Generierung der sog. Cloaking-Elemente dargestellt. Abschließend soll die Bildung und Verifizierung von Signaturen erklärt und auf zwei unterschiedliche Umformungsfunktionen eingegangen werden.

8.1 Öffentliche Informationen und Sicherheitslevel

Bei der Verwendung eines digitalen Signaturverfahrens besitzt jeder Kommunikationspartner ein Schlüsselpaar aus einem privatem und einem öffentlichen Schlüssel. Der private Schlüssel eines Teilnehmers ist nur ihm selbst bekannt und dient zur Generierung von Signaturen. Der öffentliche Schlüssel wird an andere Teilnehmer weitergegeben und kann folglich dazu verwendet werden, die erstellten Signaturen zu verifizieren.

Die Autoren von WalnutDSA definieren eine Reihe an ,,öffentlichen systemweiten Parametern" [5, 6]. Es ist nicht ganz klar, welchen Zweck die Gruppierung dieser Parameter genau erfüllen sollen. An selber Stelle wird definiert, dass diese von einer ,,zentralen Autorität" generiert werden [5, 6]. Dadurch kann angenommen werden, dass es sich hierbei um diejenigen Informationen handelt, die zwischen den Kommunikationspartnern ausgetauscht werden müssen, damit die Implementierungen miteinander kompatibel sind und folglich die Verifizierung von fremden Signaturen möglich ist. Es bleibt jedoch fraglich, inwiefern diese Parameter ,,systemweit" definiert sein sollen. Man kann sich vorstellen, verschiedene Parameter für unterschiedliche Schlüsselpaare auf einem System zu verwenden (s. Implementierung auf Raspberry Pi). Möglicherweise ist als ,,System" das Kryptosystem selbst gemeint. Im Rahmen dieser Arbeit wurde angenommen, dass die Parameter dem gerade beschriebenen Zweck dienen, zwei kommunizierende Implementierungen miteinander zu synchronisieren. Auf die Parameter soll nun eingegangen werden.

Die öffentlichen Informationen bestehen aus folgenden Bestandteilen:

n Die Anzahl an Strähnen in der verwendeten Zopfgruppe B_n.

q Eine Zweierpotenz für die Ordnung des bei der E-Multiplikation verwendeten endlichen Körpers $GF(q)$.

T-Werte n umkehrbare, geordnete Elemente aus dem endlichen Körper $GF(q)$, wobei zumindest zwei verschiedene Elemente a und b mit $1 < a < b < n$ den Wert 1 haben

müssen.

Kodierungsgeneratoren Vier verschiedene Generatoren $p_{i,j}$ der Gruppe der reinen Braids (als Untergruppe zur Zopfgruppe B_n).

Die Autoren von WalnutDSA erwähnen die zur Nachrichtenkodierung (s. Kapitel 8.3) verwendeten vier reinen Braidgeneratoren nicht. Sie müssen jedoch unbedingt bei den Kommunikationspartnern übereinstimmen, um die Signaturverifizierung möglich zu machen.

Zudem enthalten die öffentlichen Informationen nach der ursprünglichen Definition noch zwei weitere Parameter. Zum einen sind dies die Werte a und b, die die Indizes der beiden T-Werte mit dem Wert 1 darstellen sollen. In dieser Arbeit wurde auf diese beiden Werte verzichtet, da man sie für die Signaturverifizierung nicht benötigt und für die Signaturgenerierung einfach aus den T-Werten selbst auslesen kann (s. Kapitel 8.4). Zum anderen ist es eine Umschreibungsfunktion, die am Ende der Generierung auf die Signatur angewendet wird. Dies scheint auch hinfällig zu sein, da diese Funktion die Signatur zwar in ihrer Gestalt abwandelt, jedoch den zugrundeliegenden Braid nicht verändert und folglich bei der Verifizierung nicht benötigt wird. Somit wurde der Parameter in dieser Arbeit auch aus den öffentlichen Informationen ausgelassen.

Um die ausreichende Sicherheit von WalnutDSA gegen Bruteforce-Angriffe sicherzustellen, benötigen sowohl der private Schlüssel als auch die sog. Cloaking-Elemente (welche auch als Braids repräsentiert werden, s. Kapitel 8.4) eine Mindestlänge in Artingeneratoren. Wie bei gängigen Verschlüsselungsverfahren kann die Sicherheit eines Schlüsselpaars oder einer Signatur durch ein *Sicherheitslevel* SL in Bits angegeben werden. Für einen endlichen Körper GF ist dieses über die kleinste Anzahl an Operationen definiert, die zum Finden eines Geheimnisses im Körper benötigt werden [5, 13]. Ein Sicherheitslevel von 128 Bit korrespondiert also z.B. mit der Sicherheit eines symmetrischen AES-128 Verfahrens. Konkret dient es zur Bestimmung von besagten Mindestlängen des privaten Schlüssels und der Cloaking-Elemente.

Für die Cloaking-Elemente wird mithilfe des Sicherheitslevels speziell die Mindestlänge L des beinhalteten Produkts an Generatoren der Untergruppe der reinen Braids (s. Kapitel 8.4) bestimmt. Sie kann nach folgender Formel berechnet werden [5, 15]:

$$L \;=\; \lceil SL \div \big(3 \log_2 \big(n\,(n-1)\,\big)\big) \rceil \tag{8.1}$$

Für die Mindestlänge l des privaten Schlüssels existiert eine nicht-lineare Gleichung, welche mittels des Newton-Verfahrens gelöst werden kann [5, 14]:

$$l + (n-2) log_2(l) \;=\; SL + log_2\big((n-1)!\big) \tag{8.2}$$

Es sei gesagt, dass die Cloaking-Elemente nur bei der Signaturgenerierung verwendet werden. Es ist also prinzipiell möglich, unterschiedliche Sicherheitslevel für die Erstellung eines Schlüsselpaars und die Generierung einer dazu passenden Signatur zu verwenden. Ob dies sinnvoll ist oder lieber vermieden werden sollte, geht nicht aus der Arbeit über WalnutDSA hervor. Zudem fehlen Informationen der Autoren darüber, welches Sicherheitslevel ein Minimum für ausreichende Sicherheit des digitalen Signaturverfahrens darstellt.

8.2 Schlüsselgenerierung

Zur Generierung eines Schlüsselpaars benötigt man die Parameter n, q und die T-Werte.

Der private Schlüssel ist ein zufälliger, frei-reduzierter Braid aus der Zopfgruppe B_n mit der entsprechenden Mindestlänge l [5, 6]. Es bietet sich offensichtlich an, schon während der Schlüsselgenerierung sicherzustellen, dass keine frei reduzierbaren Vorkommnisse bb^{-1} bzw. $b^{-1}b$ im Braid enthalten sind.

Ein öffentlicher Schlüssel ist ein Paar aus einer $n \times n$ Matrix sowie einer n-stelligen Permutation, welche der zugrundeliegenden Permutation des privaten Schlüssels entspricht. Um den zugehörigen öffentlichen zu einem privaten Schlüssel zu generieren, wendet man die E-Multiplikation an [5, 6]:

$$PubKey = (Id_n, Id_{S_n}) \star PrivKey \tag{8.3}$$

Hierbei stellt Id_n die $n \times n$ Identitäsmatrix und Id_{S_n} die n-stellige Identitäspermutation dar.

Wie in Kapitel 8.1 angedeutet, müssen bei der Signaturverifizierung die besagten drei Parameter n, q und die T-Werte desjenigen privaten Schlüssels angewendet werden, der zur Signierung verwendet wurde. Daher sollten die Parameter an das Schlüsselpaar gekoppelt und bei dessen Speicherung miteinbezogen werden, um die korrekten Werte wiederherstellen zu können.

8.3 Kodierung des Nachrichtenhashwerts

Bei der Verwendung von digitalen Signaturalgorithmen muss der Hashwert einer Nachricht signiert werden, um neben der Authentizität des Senders auch die Integrität der Nachricht sicherzustellen. Bei WalnutDSA ist die Signatur selbst ein zusammengesetzter Braid, in den der Nachrichtenhash eingebettet wird. Daher muss der Hashwert in einen Braid konvertiert werden, wozu eine eigene Kodierungsfunktion verwendet wird. Um die Verifizierung der Signatur nicht zu vereiteln, muss der aus dem Hashwert kodierte Braid eine triviale zugehörige Permutation besitzen. Hierfür verwendet man die Untergruppe der reinen Braids. Zusätzlich muss der Braid frei reduziert sein, damit eine einzigartige Zuordnung von einem Hashwert zu seinem kodierten Braid gewährleistet ist [5, 7].

Zur Kodierung wählt man vier beliebige, verschiedene Generatoren der Untergruppe der reinen Braids (s. Kapitel 5.3). Diese benötigen eine feste Reihenfolge untereinander, sodass jeder Generator eindeutig über einen Index angesprochen werden kann. Man betrachtet den Hashwert dann als Bitfolge und iteriert über diese in 4-Bit-Inkrementen. Die zwei niedrigigwertigen Bits bestimmen, welcher der vier Braidgeneratoren für diesen Teil des Hashwerts verwendet wird. Die zwei höherwertigen Bits legen einen Exponenten im Intervall 1–4 fest, zu dem der Generator potenziert wird [5, 7]. Bei der Potenzierung handelt es sich um die übliche Multiplikation in der Braidgruppe, also einer Hintereinanderreihung des entsprechenden Generators.

Bei Verwendung der reinen Braidgeneratoren $p_{1,4}, p_{3,8}, p_{5,6}$ und $p_{7,8}$ in der Zopfgruppe B_8 würde die 4-Bitfolge 1001 also den zweiten Generator zur dritten Potenz bestimmen und somit wiefolgt kodiert werden:

7, 6, 5, 4, 3, 3, -4, -5, -6, -7, 7, 6, 5, 4, 3, 3, -4, -5, -6, -7, 7, 6, 5, 4, 3, 3, -4, -5, -6, -7

Diesen Schritt wiederholt man in 4-Bit-Schritten für den gesamten Hashwert und reiht dabei die einzelnen Teile aneinander. Die theoretische maximale Länge des resultierenden Braids lässt sich also — ohne dabei auf Einschränkungen oder Muster im Aufbau des Hashwerts einzugehen — einfach nach folgender Formel bestimmen:

$$length_{max} \;=\; 2 \cdot bytes_{hash} \cdot 4 \cdot (j_{max} - i_{min}) \cdot 2 \tag{8.4}$$

Hierbei steht $bytes_{hash}$ für die Größe des Hashwerts in Bytes sowie i_{min} und j_{max} für die Werte i und j des längsten verwendeten reinen Braidgenerators $p_{i,j}$ (mit der größten Differenz $j - i$).

Wie angesprochen, muss der engültige Braid jedoch frei-reduziert sein. Die frei-reduzierte Version des beispielhaften Braids lautet wiefolgt:

7, 6, 5, 4, 3, 3, 3, 3, 3, 3, -4, -5, -6, -7

Als einfache Möglichkeit zur Reduzierung ergibt sich, nach der Kodierung des Hashwerts den Braid zu durchlaufen und sämtliche frei-reduzierbare Vorkommnisse an Artingeneratoren zu entfernen, bis der Braid im gewünschten Zustand ist. Dies benötigt jedoch eine zusätzliche Iteration über den gesamten Braid. Zudem müssen die resultierenden Lücken des Braids wieder beseitigt werden, oder der gesamte Braid als frei-reduzierte Version neu gespeichert werden. Daher bietet es sich an, die freie Reduktion während der Kodierung selbst durchzuführen. Bei der Potenzierung kann einfach der quadratische Mittelteil des reinen Braidgenerators je nach Exponent entsprechend häufig wiederholt werden.

Die theoretische maximale Länge des Braids verringert sich hierbei, da die freie Reduktion zumindest die Länge der potenzierten Generatoren verkleinert. Sie lässt sich nun wiefolgt bestimmen:

$$length_{max} \;=\; 2 \cdot bytes_{hash} \cdot \big((j_{max} - i_{min} - 1) \cdot 2 + 8 \big) \tag{8.5}$$

Um den Braid noch weiter zu verkleinern, kann man sich bei der Wahl von j einschränken. Wählt man einen festen Wert j für alle vier reinen Braidgeneratoren $p_{i,j}$, so ergeben sich weitere frei-reduzierbare Sektionen an den Anfängen und Enden der kodierten 4-Bit-Inkremente. Man kann vor dem Aufbau der hinteren Hälfte eines Inkrements im Hashwert vorausschauen und dabei nur die Artingeneratoren anhängen, welche nicht durch den kommenden reinen Braidgenerator frei reduziert werden würden.

Folgendes Beispiel zeigt diese Situation bei Aufeinanderfolgen der beiden reinen Braidgeneratoren $p_{3,8}$ und $p_{5,8}$, wobei die frei-reduzierbaren Artingeneratoren unterstrichen wurden:

7 6 5 4 3 3 -4 <u>-5 -6 -7 7 6 5</u> 5 -6 -7

Dies verkleinert wiederum die theoretische maximale Länge des kodierten Braids. Im ungünstigsten Fall folgen jeweils vier reine Braidgeneratoren $p_{i_{min},j}$ und $p_{j-1,j}$ aufeinander. Dadurch wird durch die freie Reduktion jedes zweite Inkrement auf 6 Artingeneratoren reduziert. Die Formel für die Maximallänge verändert sich so zum Folgenden:

$$length_{max} \;=\; bytes_{hash} \cdot \big((j - i_{min} - 1) \cdot 2 + 8 + 6 \big) \tag{8.6}$$

Für $j = n$ entspricht dies genau dem Vorgehen der Autoren von WalnutDSA. Das Kriterium für die Wahl der reinen Braidgeneratoren ist, dass sie eine freie Untergruppe zur Zopfgruppe B_n bilden, in der ein frei-reduzierter Braid niemals das neutrale Element ist [5, 7]. Es darf also offensichtlich kein Generator das leere Braidwort sein. Dies lässt jedoch noch die Möglichkeit offen, für die reinen Braidgeneratoren $p_{i,j}$ nur solche zu wählen, bei denen $i = j - 1$ gilt. Dadurch würde der kodierte Braid auf eine maximale Länge von $16 \cdot bytes_{hash}$ reduziert werden.

Bei den Implementierungen dieser Arbeit wurde auf diese Möglichkeit verzichtet. Zum einen soll die Kompatibilität mit potentiellen anderen Implementierungen gewährleistet werden, die sich streng an die Ausarbeitung der Autoren halten. Zum anderen ist ohne genauere Analyse nicht offensichtlich, ob diese Optimierung nicht die Sicherheitseigenschaften der Kodierung gefähren würde.

Die Sicherheit des Kodierungsverfahrens an sich ist gegeben, solange als Eingabe nur kryptographische Hashwerte verwendet werden. Die Kodierungsfunktion E ist homormorph; sie erhält also die Zusammensetzung der einzelnen Inkremente, sodass auch für zwei Nachrichten m_1 und m_2 gilt $E(m_1 m_2) = E(m_1)E(m_2)$. Würde man die Nachricht selbst in einen Braid kodieren, so könnte man durch besagte Eigenschaft einfach Rückschlüsse auf die Nachricht ziehen [5, 7].

Die Autoren von WalnutDSA spezifizieren kein Verfahren zur generischen asymmetrischen Ver- und Entschlüsselung von Daten. Würde man ein solches definieren wollen, so müsste man zumindest sicherstellen, dass der verschlüsselte Braid einer Umschreibungsfunktion (s. Kapitel 8.9) unterzogen wird. Ob die dadurch erreichte Verschleierung der Struktur des Braids ausreichen würde, um die Sicherheit des Verfahrens zu garantieren, ist uns nicht bekannt und bedarf wohl weiterer Diskussion.

8.4 Cloaking-Elemente

Um die finale Signatur zu erstellen, werden mehrere Teilbraids zu einem ganzen konkateniert. Mit der Generierung des privaten Schlüssels und der Kodierung des Nachrichten-Hashs verfügt der Algorithmus bisher über zwei Braids, die richtig zusammengesetzt die Verifizierung mit dem öffentlichen Schlüssel bestehen würden. Durch die Anwendung einer Umformungsfunktion bliebe das Ergebnis der Verifizierung identisch und der private Schlüssel wäre trotz der vorherigen einfachen Konkatenation nicht direkt ersichtlich. Da jedoch sowohl der Hash, als auch die Kodierungsgeneratoren und damit auch die Länge des kodierten Hashs öffentlich bekannt sind, wäre es möglich, durch einen Algorithmus zur Lösung des *Conjugacy Search Problems* den privaten Schlüssel von einem auf diese Art und Weise erzeugten Braid effizient zu extrahieren.[21] Um die Positionen der Bestandteile der Signatur zu verschleiern werden sogenannte Cloaking-Elemente generiert. Dabei handelt es sich um Braids zufälliger Länge, die das Ergebnis der E-Multiplikation nicht beeinflussen und dennoch bei der freien

Reduktion nicht verschwinden.

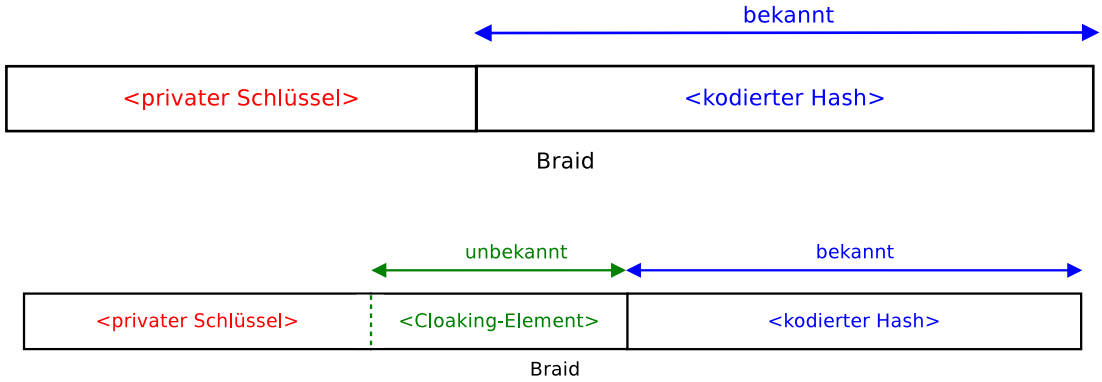


Abbildung 8.1: Konkatenation der Teilbraids mit und ohne Cloaking-Element

Für die Generierung der Cloaking-Elemente wird die Permutation σ des Schlüssels, bzw. des zu „verhüllenden“ Braids, und die Variablen a und b benötigt. Man wählt ein zufälliges $i \in \{1, \ldots, N\}$ und erzeugt einen Braid, der $\sigma^{-1}(a)$ an die Stelle i und $\sigma^{-1}(b)$ an die Stelle $i+1$ verschiebt. [31, p. 5]

Den nächsten Teilbraid bilden zufällig generierte reine Braids, die mithilfe folgender Generatoren für reine Braids und zufällig gewählten l und r erzeugt werden:[31, p. 7]

$$g_{l,r} = b_{r-1}b_{r-2}\ldots b_{l+1} \cdot b_l^2 \cdot b_{l+1}^{-1}\ldots b_{r-2}^{-1}b_{r-1}^{-1}, \qquad 1 \leq l < r \leq N. \tag{8.7}$$

Es werden solange reine Braids aneinander gehängt, bis die vordefinierte Mindestlänge L erreicht wurde.

Sei ω der bisherige Braid.
Das fertige Cloaking-Element ist definiert durch: $v = \omega b_i^2 \omega^{-1}$, wobei es sich bei b_i^2 um ein doppeltes Vorkommen des Artingenerators i und bei ω^{-1} um das Braid-Inverse von ω handelt.[31, p. 5]

Beispiel der Generierung eines Cloaking-Elements:

gegeben: $a = 4$; $b = 5$; $i = 5$; T-Werte = {22, 56, 76, 1, 1, 128, 15, 33}; $L = 10$
und Braid β: {-3 6 4 -1 2}

ω Teil 1:
Permutation von β: (2 4 1 5 3 7 6 8);
$\sigma^{-1}(a) = 2$;
$\sigma^{-1}(b) = 4$
$\rightarrow$ Braid, der Position 2 nach 5 und 4 nach 6 verschiebt: 4 5 2 3 4

ω Teil 2:
Braid mind. der Länge $L = 10$, bestehend aus zufällig gewählten reinen Braids: $g_{3,5}\ g_{4,7}\ g_{3,7}$
= 4 3 3 -4 6 5 4 4 4 3 3 -4 -5 -6 (frei reduziert)

Cloaking-Element $v = \omega b_i^2 \omega^{-1}$
= 4 5 2 3 4 4 3 3 -4 6 5 4 4 4 3 3 -4 -5 -6 5 5 6 5 4 -3 -3 -4 -4 -4 -5 -6 4 -3 -3 -4 -4 -3 -2 -5 -4

E-Multiplikation von β und v:

Braid β in der Colored-Burau-Form: $(CB(\beta), \sigma_\beta) =$

$$\left(\begin{pmatrix} 81 & 163 & 242 & 0 & 0 & 0 & 0 & 0 \\ 22 & 22 & 1 & 0 & 0 & 0 & 0 & 0 \\ 22 & 22 & 76 & 76 & 1 & 0 & 0 & 0 \\ 0 & 0 & 76 & 76 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 128 & 128 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}, (2,4,1,5,3,7,6,8) \right)$$

e-multipliziert mit dem Cloaking-Element v: $(CB(\beta), \sigma_\beta) \star (v, \sigma_v) =$

$$\left(\begin{pmatrix} 81 & 163 & 242 & 0 & 0 & 0 & 0 & 0 \\ 22 & 22 & 1 & 0 & 0 & 0 & 0 & 0 \\ 22 & 22 & 76 & 76 & 1 & 0 & 0 & 0 \\ 0 & 0 & 76 & 76 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 128 & 128 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}, (2,4,1,5,3,7,6,8) \right) = (CB(\beta), \sigma_\beta)$$

Weder die Colored-Burau-Matrix noch die Permutation von β ändern sich durch die E-Multiplikation mit dem Cloaking-Element.

8.5 Signaturbildung

Die Generierung von Signatur für einen Hashwert H(m) von der zu signierenden Nachricht m läuft in folgende Schritten:
1. Generiere Cloaking-Elemente v, v_1 und v_2, wobei v die Colored-Burau-Representation von leeren Braid versteckt(D.h. $(Id_N, Id_{S_N}) \star v = (Id_N, Id_{S_N})$ mit der Identitätsmatrix Id_N und die triviale Permutation Id_{S_N}) und v_1, v_2 die Colored-Burau-Representation von den privaten Schlüssel versteckt.
2. Codiere den Hashwert H(m) in E(H(m)). 3. Konkateniere Elemente, sodass $(v_2 v_1^{-1} Priv(S)^{-1} vE(H(m)) Priv(S) v_1)$ gebildet wird. Das Ergebnis von diesem Schritt ist nicht sicher, da der privaten Schlüssel noch unverändert steht.

4. Forme das Ergebnis vom Schritt 3 um.
Hier wird die Signatur mit Hilfe einer oder mehre Umformungsalgorithmus verändert, um das beim dritten Schritt erwähnte Problem zu lösen. Ein Umformungsalgorithmus soll in der Lage sein, die konkatenierte Elemente z.B. durch Normalization von Braid zu mischen und dadurch genug Diffusion zu erzeugen. Als den Umformungsalgorithmen kann man BKL-Normalform verwendet.[5, S. 8]

8.6 Verifizierung

Die Verification von Signatur haben folgende Schritte:
1. Codiere die Nachricht in E(H(m)). 2. Definiere Colored-Burau-Representation von E(H(m)) als Pub(E(H(m)). 3. E-Muliplizieren den öffentlichen Schlüssel mit der Signatur(Pub(S) $\star$ Sig). 4. Überprüfe die Gleichheit von 2 Matrizen Matrix(Pub(S) $\star$ Sig) und (Matrix(Pub(E(H(m) * Matrix (Pub(S)). Die Verifizierung ist positiv, wenn die beide Matrizen gleich sind.[5, S. 8]

8.7 Umformungen

Für die Umformung von Signatur kann man den BKL-Normalform und den Algorithmus von Dehornoy verwenden. Die notwendige Diffusion wird überwiegend durch BKL-Normalform erzeugt. Der Henkelreduktionsalgorithmus von Dehornoy wird zusätzlich verwendet, um die Gesamtlänge von Signatur zu reduzieren, da die Signatur über BKL-Normalform oft viel länger wird.

8.7.1 BKL(Birman-Ko-Lee)-Normalform

Ein BKL-Normalform verwandelt beliebeiges Braid mit n Strähne zu diesem Form:

$$W = \begin{cases} \delta_n^u A_1 A_2 ... A_k \text{ (Linksnormalform)} \\ A_1 A_2 ... A_k \delta_n^u \text{ (Rechtsnormalform)} \end{cases} \text{, wobei } u \in Z \tag{8.8}$$

δ ist ein *Funtamental Braid* und A_i sind *Kanonische Factoren*, die aus Produkt von positiven Band-Generatoren entstehen. Dabei ist u möglichst groß und k möglichst kleine Zahl[32, S. 150]. D.h., man verschiebt alle verschiebbare Elemente nach Links(oder nach Rechts, wenn es um Rechtsnormalform geht), um möglichst viele Generatoren als dieses Fundamental Braid zusammenzufassen.

Band-Generator

Bis zu diesem Kapitel hat man Artin-Generatoren verwendet, die Transposition zwischen nebeneinanderliegende Strähne darstellen. Für die BKL-Normalform nutzt man andere Generatoren für Braid. Diese Generatoren vertauscht Position von zwei beliebige Strähne, ohne dabei Position von andere Strähne zu ändern[33, S. 325]. Folgende Abbildung zeigt die Struktur von Band-Generatoren.
s und r sind die Indizies von den Strähne und man schreibt die Strähne zuerst, die auf anderen Strähne laufen wird. Ein Band-Generator ist positiv, wenn $s \geq r$ gilt, und negativ, wenn $s<r$ gilt. Die Artin-Generator sind also spezielle Fälle von Band-Generatoren, in der s und r

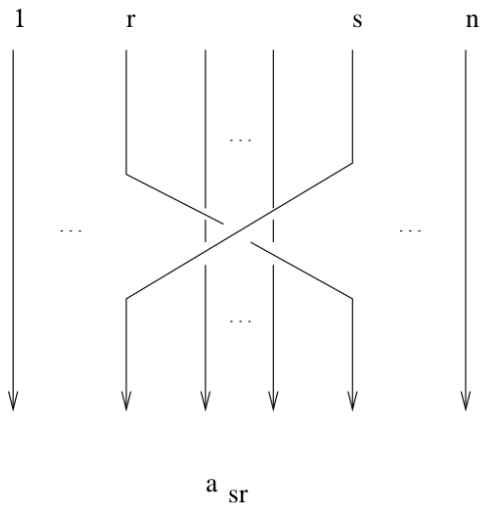


Abbildung 8.2: Band-Generator[34, S. 8]

benachbart sind[33, S. 324-325]. Jeder Band-Generator hat auch eine äquivante Darstellung als Artin-Generatoren:

$$a_{sr} = \begin{cases} (\sigma_{s-1}\sigma_{s-2}...\sigma_{r+1})\sigma_r(\sigma_{r+1}^{-1}...\sigma_{s-2}^{-1}\sigma_{s-1}^{-1}) & \text{für } s \geq r \\ (\sigma_{r-1}\sigma_{r-2}...\sigma_{s+1})\sigma_s^{-1}(\sigma_{s+1}^{-1}...\sigma_{r-2}^{-1}\sigma_{r-1}^{-1}) & \text{für } s<r \end{cases} \tag{8.9}$$

Hier muss man aufpassen, dass alle positive Artin-Generatoren auch positive Band-Generatoren sind, aber nicht alle positive Band-Generatoren sind mit positiven Artin-Generatoren darstellbar.

Fundamental Braid

Das Fundamental Braid δ_n ist spezielles Braid mit mehre wichtigen Eigenschaften. Das Fundamental Braid sieht so aus:

$$\begin{aligned} \delta_n &= a_{n(n-1)}a_{(n-1)(n-2)}...a_{21} \\ &= \sigma_{n-1}\sigma_{n-2}...\sigma_1 \text{ (in Artin-Generator)} \end{aligned} \tag{8.10}$$

Eigenschaft (1): Für alle positive Band-Generator a gilt:

$$\delta_n = aA = Ba \text{ , für Braids A und B} \in B_n^+ \tag{8.11}$$

B_n^+ ist die Menge aus Produkt von positive Generatoren[32, S. 150].

Eigenschaft (2): Für alle positive Band-Generator a gilt:

$$\begin{aligned} a\delta_n &= \delta_n\tau(a) \\ \delta_n a &= \tau^{-1}(a)\delta_n \end{aligned} \tag{8.12}$$

wobei τ ist ein Automorphismus von B_n, der als $\tau(a_{rs}) = \delta_n^{-1} a \delta_n = a_{(r+1)(s+1)}$ definiert wird(Falls $r+1 > n$, dann $\tau(a_{rs}) = a_{(s+1)((r+1)\%n)}$)[32, S. 150]. Diese Regel ist bei der Berechnung von Normalform ganz nützlich, um Fundamental Braids nach links oder nach rechts zu verschieben.

Kanonische Factor

Die kanonische Factoren sind die Produkt aus positiven Band-Generatoren, die noch folgende Bediengung erfüllen sollen:

$$e \leq A \leq \delta_n \text{ (e ist ein neutrales Braid)} \tag{8.13}$$

Für zwei Braids V und W in B_n gilt $V \leq W$, wenn es $PVQ = W$ ist, wobei P und Q aus B_n^+ sind. Es gibt $\frac{(2n)!}{n!(n+1)!}$ kanonische Factoren, die als A anpasst. Das ist ein großer Vorteil von BKL-Normalform im Vergleich zu andere Normalformen, die ihre Berechnung mit Atrin-Generatoren durchführen, da sie andere Fundamental Braid benötigen, wodurch die Anzahl von kanonischen Factoren viel größer mit $n!$ ist. Da es wengiere kanonische Factoren gibt, kann man einige Operation leichter berechnen[32, S. 150].
Nun muss man überlegen, wie man alle kanonische Factoren einzigartig darstellen kann. Diese Darstellung heißt *Descending Cycle Decomposition Table*. Dieses Table ist von der Permutation von Braid ableitbar und jedes Table ist äquivalent zu genau einem kanonischen Faktor und diese gelten auch umgekehrt. Wie sie heißen, kann dieses Table mehre absteigende Zyklen enthalten. Hier ist ein Beispiel von Descending Cycle Decomposition Table, das ein Braid A mit $a_{65}a_{52}a_{43}$ aus B_6 darstellt:

$$\begin{aligned} &\text{Permutation von A: } \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 6 & 4 & 3 & 2 & 5 \end{pmatrix} \\ &\text{Descending Cycle Decomposition Table von A: } \begin{pmatrix} 1 & 6 & 4 & 4 & 6 & 6 \end{pmatrix} \end{aligned} \tag{8.14}$$

Man kann 2 Zyklen aus der Permutation von A herleiten. Diese sind (3 4) und (2 6 5). Da in einem Zyklus beliebige Verschiebung erlaubt ist, solange die Reihenfolge identisch ist, sind dann (4 3) und (6 5 2). (4 3) entsteht aus a_{43} und (6 5 2) aus $a_{65}a_{52}$. Sowie dieses Beispiel haben ein Descending Cycle immer den Form von $(s_i s_{i-1} ... s_1)$ mit $s_i > s_{i-1} > ... > s_1$. Die Zyklen werden in dem Descending Cycle Decomposition Table geschieben, in dem man die alle Zellen, die zu jeweiligem Zyklus gehören, mit höchtste Zahl von Zyklus auffüllt(z.B. bei (4 3) werden die dritte und vierte Zellen mit 4 geschrieben).
Der Zyklus (4 3) und (6 5 2) lassen sich nicht gegeneinander „separieren“, da die Anfangsindex und Endesindex von Zyklus (4 3) zwischen 5 und 2 sind. Die Zyklen wie diese sind *parallel* und erfüllen diese Bedingung:

$$\begin{aligned} &\text{Die Zyklen S und T sind parallel } \leftrightarrow (s_a - t_c)(s_a - t_d)(s_b - t_c)(s_b - t_d) > 0, \\ &ie \text{wobei } 1 \leq a < b \leq i, 1 \leq c < d \leq j \end{aligned} \tag{8.15}$$

Der Band-Generator a_{31} mit dem Zyklus (1 3) kann nicht zusätzlich mit $a_{65}a_{52}a_{43}$ in einem Table dargestellt werden, da es den Zyklus (4 3) zu (4 3 1) erweitert, was aber nicht mehr parallel zu (6 5 2) ist. In diesem Fall benötigt man zwei Tables, da sie sonst keine kanonische Factoren sind. Demgegenüber kann a_{21} hinzugefügt werden, da dann (6 5 2) zu (6 5 2 1) erweitern wird, der noch parallel zu (3 1) ist.

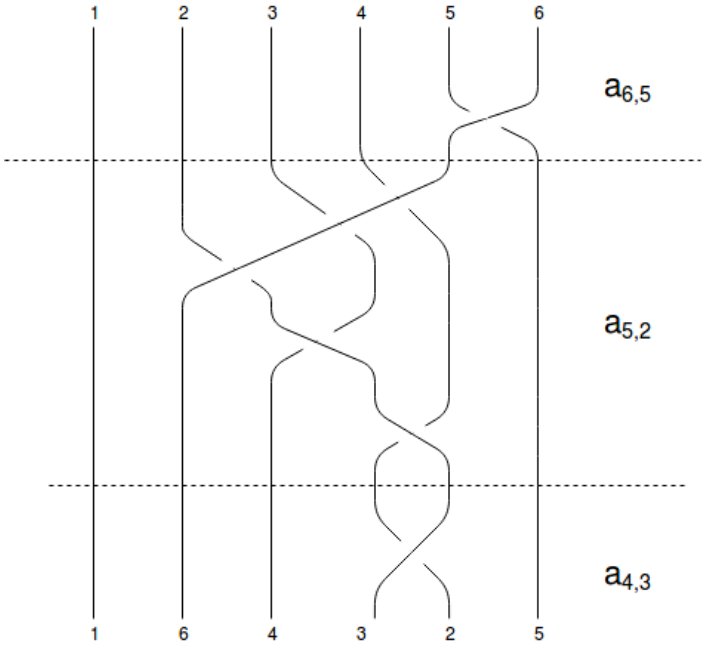


Abbildung 8.3: Graphische Darstellung von $a_{65}a_{52}a_{43}$

Hier sieht man nochmal, wieso das Fundamental Braid im Sinne von der partiellen Ordnung größer oder gleich zu alle andere kanonische Factoren ist.

$$\begin{aligned}
&\text{Permutation von } \delta_n\text{: } \begin{pmatrix} 1 & 2 & 3 & \dots & n-1 & n \\ n & 1 & 2 & \dots & n-2 & n-1 \end{pmatrix} \\
&\text{Zyklus von } \delta_n\text{: } \begin{pmatrix} 1 & n & n-1 & n-2 & \dots & 3 & 2 \end{pmatrix} = \begin{pmatrix} n & n-1 & n-2 & \dots & 3 & 2 & 1 \end{pmatrix} \\
&\text{Descending Cycle Decomposition Table von } \delta_n\text{: } \begin{pmatrix} n & n & n & \dots & n & n \end{pmatrix}
\end{aligned} \tag{8.16}$$

Da der Zyklus von n bis 1 füllt, kann er nicht mehr erweitert werden und es gibt auch kein „Zwischenbereich“, wo andere Zyklus reingelegt werden kann(Jeder Zyklus entsteht aus positive Generatoren).
Für die Berechnung von Normalform nutzt man sowohl Permutation Table(Permutation von Braid ohne die erste Zeile, da sie immer gleich ist) als auch Descending Cycle Decomposition Table, da manche Operation wie z.B. Multiplikation von Kanonischen Factoren als Permutation einfacher zu berechnen ist, während die Meet-Operation, die später noch genauer betrachten wird, aber als Descending Cycle Decomposition Table einfacher zu berechnen ist[32, 147]. Darum braucht man folgende Algorithmen1 und 2, so dass man von einem Darstellungsform auf anderen hin und her konvertieren kann.

Algorithm 1: Konvertierung von Permutation auf Descending Cycle Decomposition Table[32, 147]

Input: Permutation Table A in Länge von n
Output: Descending Cycle Decomposition Table X in Länge von n
for $i \leftarrow 1$ **to** n **do**
  $X[i] \leftarrow 0$;
end
for $i \leftarrow n$ **to** 1 **do**
  if $X[i] = 0$ **then**
    $X[i] \leftarrow i$;
  end
  if $A[i] < i$ **then**
    $X[A[i]] \leftarrow X[i]$;
  end
end

Algorithm 2: Konvertierung von Descending Cycle Decomposition Table auf Permutation[32, 148]

Input: Descending Cycle Decomposition Table X in Länge von n
Output: Permutation Table A in Länge von n
(We need an array Z of size n.)
for $i \leftarrow 1$ **to** n **do**
  $Z[i] \leftarrow 0$;
end
for $i \leftarrow 1$ **to** n **do**
  if $Z[X[i]] = 0$ **then**
    $A[i] \leftarrow X[i]$;
  else
    $A[i] \leftarrow Z[X[i]]$;
  end
  $X[A[i]] \leftarrow X[i]$;
end

Operationen

Um den Normalform zu berechnen, braucht man diverse Operationen, die nun auf kanonische Factoren einsetzbar sind. Diese sind der Vergleich, die Produkt, die Inverse, der Automorphismus und das *Meet*.

Vergleich Mit dem Vergleich überprüft man die Gleichheit von zwei kanonischen Factoren. Dabei sehen wir, ob das Permutation Table oder das Descending Cycle Decomposition Table identisch ist. Der Aufwand ist für die beide Darstellungen gleich mit O(n)[32, S. 148].

Produkt Diese Operation ist als das Permutaion Table leichter durchführbar. Die Komplexität vom folgenden Algorithmus3 ist O(n)[32, S. 148].

Algorithm 3: Multiplikation von Permutation Tables

Input: Permutation Table A, B in Länge von n
Output: Permutation Table C in Länge von n

for $i \leftarrow 1$ **to** n **do**
  $C[i] \leftarrow A[B[i]]$;
end

Inverse Diese Operation ist auch als das Permutaion Table leichter durchführbar mit der Komplexität von O(n)[32, S. 148]. Der ALgorithmus4 beschreibt genauere Ablauf.

Algorithm 4: Inverse von Permutation Table

Input: Permutation Table A in Länge von n
Output: Permutation Table B mit $B = A^{-1}$ in Länge von n

for $i \leftarrow 1$ **to** n **do**
  $B[A[i]] \leftarrow i$;
end

Automorphismus Der Automorphismus ist für einen kanonischen Factor A als $\tau(A) = \delta * A * (\delta^{-1}$ definiert. D.h., zwei Multiplikation soll durchgeführt werden.
Für die Berechnung von BKL-Normalform braucht man aber sehr häufig $\tau(A)^z$ mit $\delta^z * A * (\delta^{-z}$ statt einfacher Automorphismus. Dabei ist Permutation Table von δ^n noch mal zum Betrachten:

$$\begin{aligned}
\delta &= \begin{pmatrix} 8 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{pmatrix} \\
\delta^2 &= \begin{pmatrix} 7 & 8 & 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix} \\
\delta^3 &= \begin{pmatrix} 6 & 7 & 8 & 1 & 2 & 3 & 4 & 5 \end{pmatrix} \\
&\dots \\
\delta^8 &= \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{pmatrix}
\end{aligned} \tag{8.17}$$

Nun sehen wir, dass für jede δ die Permutation Table nach rechts(links, wenn die Exponent negativ ist) rotiert. D.h., ein beliebiges δ^z kann man mit der $nmod8$ Verschiebungen nach rechts oder nach links gewinnen, ohne dabei tatsächliche die Produkt von Permutationen durchführen zu müssen.

Meet Die Meet-Operation ist einer von wichtige Stellen, wo das Descending Cycle Decomposition Table verwendet wird. Mit dieser Operation muss man alle gemeinsame Teilzyklen herausfinden können. Hier ist ein Beispielsein- und ausgabe.

$$\begin{aligned}
&\text{Eingaben:} \\
&\text{Descending Cycle Decomposition Table A} = \begin{pmatrix} 2 & 2 & 4 & 4 \end{pmatrix} \\
&\text{Descending Cycle Decomposition Table B} = \begin{pmatrix} 1 & 4 & 4 & 4 \end{pmatrix} \\
&\text{Ausgabe:} \\
&\text{Descending Cycle Decomposition Table C} = \begin{pmatrix} 1 & 2 & 4 & 4 \end{pmatrix}
\end{aligned} \tag{8.18}$$

A = (2 2 4 4) B = (1 4 4 4)

Bucket

	Array 1
1	
2	1, 2
3	
4	3, 4

Sortiere Bucket 2 von Array 1:

	Array 2
1	1
2	
3	
4	2

Kein Zyklus

Sortiere Bucket 4 von Array 1:

	Array 2
1	
2	
3	
4	3, 4

Ein Zyklus mit (4 3).
F"ulle Index 4 und 3 von
Table C mit 4

Ergebnis: Table C mit (1 2 4 4)

Abbildung 8.4: Ein Beispielsablauf von Berechung der Meet-Operation

A enthält den Zyklus (4 3), (2 1) und B (4 3 2). Also (4 3) ist dann der gemeinsame Teilzyklus.
Hier ist eine mögliche Variante, diese Operation zu berechnen. Dabei braucht man zwei zweidimensionale Arrays als Buckets. Die Indizies von Descending Cycle Decomposition Table A wird nach ihren Wert im ersten Array eingelegt. Als nächsten Schritt muss man Buckets einzeln betrachten. Falls es mehre Elemente haben, wird das in zweiten Array verteilt, wobei diesmal der Wert von Table B genommen wird. Wenn ein Bucket im zweiten Array mehre Elemente hat, enthält es gemeinsame Zyklen. Die Abbildung 8.4 stellt diese Schritte graphisch dar.

Umwandlung von Artin-Generatoren in kanonische Factoren

Hier fängt der erste Schritt vom Normalform. Dazu muss man alle Artin-Generatoren in Kanonische Factoren umwandlen. Da manche Operation als Permutation Table einfacher ist, wird die Daten zu erst nur in Permutation Table konvertieren. Für die positive Artin-Generatoren muss man deren Permutation gleich verwenden. Z.B. für σ_2 wäre es dann $\big(1 \quad 3 \quad 2 \quad 4 \quad 5 \quad 6 \quad ... \quad n\big)$. Für die negative Artin-Generatoren braucht man noch zusätzliche Schritte. Die negative Generatoren werden mit $\delta^{-1} * (\delta * \sigma_n^{-1})$ für Linksnormalform dargestellt(Mit $(\sigma_n^{-1} * \delta)\delta^{-1}$, falls es um Rechtsnormalform geht). $\delta * \sigma_n^{-1}$ wird als ein kanonischer Factor beschrieben. σ_2^{-1} ist also $\delta^{-1} * (\big(1 \quad 3 \quad 2 \quad 4 \quad 5 \quad 6 \quad ... \quad n\big) * \big(n \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad ... \quad n-1\big) = \delta^{-1} * \big(n \quad 1 \quad 3 \quad 2 \quad 4 \quad 5 \quad ... \quad n-1\big))$.
Nun muss man noch alle D^{-1} zu einer Seite verschieben. Dafür werden der Automorphismus verwendet. So wird der Schritt aussehen[32, S. 151]:

$$A * (D^{-n} * B_1 B_2 ... B_n) = D^{-n} * \tau^n(A) * B_1 B_2 ... B_n \tag{8.19}$$

Der Algorithmus 5 zeigt der Gesamtprozess. Dieser hat die Komplexität von $\emptyset(l^2 n$, wobei n die Breite von Braid und l die Anzahl von Permutation Table ist.

Algorithm 5: Konvertierung von Artin-Generatoren in kanonische Factoren für Linksnormalform

Input: Braid A mit i Artin-Generatoren
Output: (funds, P), mit $funds \in Z$ und P als Permutation Tables

$funds = 0$;
if $Linksnormalform$ **then**
  for $i \leftarrow n$ **to** 1 **do**
    $temp = to\_perm\_left(A[i])$;
    $P[i] = \tau^n(temp)$;
    if $A[i] < 0$ **then**
      $funds = funds + 1$;
    end
  end
else
  `// falls Rechtsnormalform`
  for $i \leftarrow 1$ **to** n **do**
    $temp = to\_perm\_right(A[i])$;
    $P[i] = \tau^{-n}(temp)$;
    if $A[i] < 0$ **then**
      $funds = funds + 1$;
    end
  end
end

Normalization von kanonische Factoren

Algorithm 6: Linksnormalizierung von kanonische Factoren[32, S. 152]

Input: (funds, P), mit $funds \in Z$ und P als kanonische Factoren
Output: (funds, Q), mit Q als normalisierte kanonische Factoren

$l = Length(P)$;
$i = 1$;
while $i < l$ **do**
  $t \leftarrow l$;
  for $j \leftarrow (l-1)$ **to** i **do**
    `// berechne gemeinsame Braids zwischen Rechtskomplement von P[j] und P[j+1]`
    $B = meet(P[j]^{-1} * \delta, P[j+1])$;
    if B *ist nicht trivial* **then**
      $t \leftarrow j$;
      `// verschiebe B nach links`
      $P[j] \leftarrow P[j] * B$;
      $P[j+1] \leftarrow B^{-1} * P[j+1]$;
    end
  end
  $i \leftarrow t + 1$;
end
`// fasse Fundamental Braids an der linken Seite in funds zusammen`
while $l > 0$**and**$P[1] = \delta$ **do**
  lösche P[1] von P; $l = l - 1$;
  $funds = funds + 1$;
end
`// lösche triviale Braids an der rechten Seite`
while $l > 0$**and**$P[i]$ *ist trivial* **do**
  lösche P[i] von P; $l = l - 1$;
end

Der Algorithmus 6 normalisiert eingegebene kanonische Factoren nach links. Dabei betracht der für jeden Schritt zwei kanonische Factoren. Da wird ein Meet zwischen der Rechtskompliment vom linken Element(Es ist $A_j^{-1} * D$) und dem rechten Element. Falls das Ergebnis von der Meet-Operation nicht trivial ist, wird es nach links verschoben.
Für den Rechtsnormalform muss man von rechts nach links iterieren. Trivialerweiser muss man Linkskompliment von rechten Element berechnen und das Ergebnis von Meet-Operation nach links verschieben. Und Zusammenfassung von Fundamental Braids und Löschen von triviale Braids soll nun an andere Seite passieren.
Die durschnittliche Länge von BKL-Links- und Rechtsnormalform von Walnut-DSA Signatur ist beim Linksnormalform kürzer. Genauere Konfiguration und Werte sind bei [35] befindlich.

Konvertierung von Normalform in Artin-Generatoren

Hier konvertiert man zu erst Permutation Tables in Descending Cycle Decomposition Table, da die Zyklen damit leichter lesbar sind. Dann schreibt man alle Zyklen mit Artin-

Generatoren. Dann bleibt noch die Fundamental Braids übrig. Bei dem Linksnormalform wird am Anfang des Braids alle Fundamental Braids hinzugefügt(Bei dem Rechtsnormalform am Ende des Braids). Der Algorithmus 7 zeigt den konkrete Ablauf.

Algorithm 7: Konvertierung von kanonischen Factoren auf Artin-Generatoren

Input: (funds, Q), mit l kanonische Factoren
Output: A mit x Artin-Generatoren
`// Braid hat n Strähne`

if *Leftnormalform* **then**
  Schreibe Fundamental Braids;
end
for $i \leftarrow 1$ **to** l **do**
  for $j \leftarrow n$ **to** 2 **do**
    `// suche Descending Cycle mit j als höchste Index`
    $last \leftarrow Q[i][j]$;
    for $k \leftarrow j-1$ **to** 1 **do**
      if *Q[i][j] = Q[i][k]* **then**
        Schreibe Band-Generator $a_{last\ k}$ in A;
      end
      $last \leftarrow Q[i][k]$;
    end
  end
end
if *Rechtsnormal* **then**
  Schreibe Fundamental Braids;
end

8.7.2 Algorithmus von Dehornoy

Dieser Algorithmus ist ein anderer Normierungsalgorithmus, der zum Vergleich mehre unterschiedliche Braids verwendet wird[36, S. 1]. Der interessante Punkt ist das, dass dieser Verfahren unsere über BKL-Normalform umgeformte Signatur in der Regel stark verkürzen.

Henkel

In dieser Algorithmus wird folgende spezielle Folge von Braids als Henkel definiert:

$$\begin{aligned} &\sigma_j^e * v_0 * v_1 * ... * v_{m-1}\sigma_j^{-e}, \\ &\text{mit } e \in \{1,-1\} \text{ und } v_0,...,v_{m-1} \notin \{\sigma_{j-1}^1, \sigma_{j-1}^{-1}, \sigma_j^1, \sigma_j^{-1}\} \end{aligned} \tag{8.20}$$

Da ein Henkel keine σ_{j-1}^1 Generatoren enthalten, wird ein Henkel wie in Abbildung 8.5 aussehen. Der Generator σ_{j-1}^1 und σ_{j-1}^{-1} bilden also ein Henkel mit der Strähne $j+1$.

8.7.3 Reduktion von Henkeln

Die Henkelreduktion hat den Effekt die Abbildung 8.6 zeigt.
Für ein Henkel σ_j^e muss man lediglich die σ_{j+1}^f in der Henkel mit $\sigma_{j+1}^{-e} * \sigma_j^f * \sigma_{j+1}^e$ ersetzen

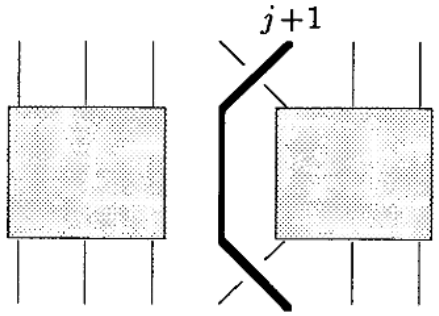


Abbildung 8.5: Ein σ_j Henkel[36, S. 205]

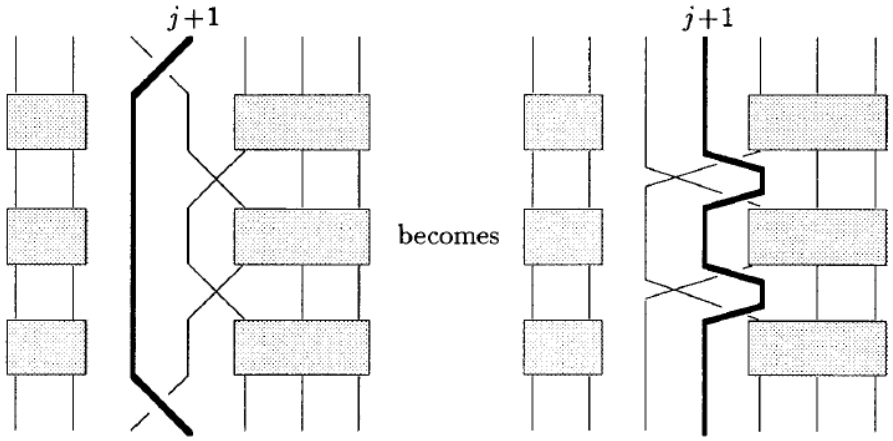


Abbildung 8.6: Henkelreduktion für σ_j[36, S. 206]

und σ_j^e, σ_j^{-e} am Anfang und am Ende der Henkel löschen.
Hier kommt doch eine Bedingung, wann ein Henkel reduzierbar ist. Ein Henkel σ_j^e ist reduzierbar, wenn er keinen anderen Henkel mit σ_{j+1}^f enthält, wobei $e, f \in -1, 1$. Sonst ist es möglich, dass die Reduktion das Braid unendlich verlängert, wie im folgenden Beispiel:

$$\begin{aligned} &\sigma_1\sigma_2\sigma_3\sigma_{-2}\sigma_{-1} \\ &\Leftrightarrow \sigma_{-2} * \sigma_1\sigma_2\sigma_3\sigma_{-2}\sigma_{-1} * \sigma_2 \\ &\Leftrightarrow \sigma_{-2}^n * \sigma_1\sigma_2\sigma_3\sigma_{-2}\sigma_{-1} * \sigma_2^n \text{ (nach n Reduktionen auf Henkel } \sigma_j) \end{aligned} \tag{8.21}$$

Gesamtablauf von Algorithmus

Es kann mehre Variante geben, in welcher Reihenfolge Henkeln reduzieren, weil man lediglich darauf achten muss, ob ein Henkel reduzierbar ist. Im Paper [36] wird dazu zwei Variante vorgeschlagen.

„greedy handle reduction“ Diese Variante erfüllt minimale Bedinung, zu Vergleich unterschiedliche Braids benötigt wird. Hier sieht man einen Artin-Generator mit den niedrigsten Absolut als ein *Main-Generator*. Man reduziert alle Henkeln, die gleiche Absolut wie der

Main-Generator haben. Dabei werden alle verschachtele Henkeln in betroffenen Henkeln zu erst reduziert.

„full handle reduction“ Diese Variante sucht einen reduzierbaren Henkel von links und reduziert es. Es wird beliebig oft wiederholt, während noch ein Henkel im Braid befindlich ist. Diese Variante könnte interessant sein, wenn es viel mehr darum geht, Anzahl von Artin-Generatoren zu verkleinern. Da diese aber auch alle Henkeln reduziert, braucht es mehr Berechnung

9 Implementierung unter Linux auf dem Raspberry Pi

Im Folgenden soll die Implementierung von WalnutDSA unter Linux auf dem Raspberry Pi dargestellt werden. Hierbei wird zunächst kurz das verwendete Gerät und Betriebssystem vorgestellt. Anschließend werden die wichtigsten Teile der generischen Implementierung des Signaturverfahrens aufgezeigt. Darauf folgt eine Darstellung der Einbettung des Quellcodes in ein Kernelmodul für Linux sowie die Einbindung in die kryptographische API des Betriebssystems. Zum Schluss soll die Implementierung bezüglich ihrer Korrektheit und der Fehlerfreiheit ihrer Speicherverwaltung sowie ihrem Speicherplatzverbrauch und Laufzeitverhalten evaluiert werden.

9.1 Raspberry Pi

Raspberry Pi bezeichnet eine Reihe sogenannter *Einplatinencomputer*, also Geräte die einen vollständigen Computer auf einer kleinen Leiterplatte beinhalten. Die Rechner werden von der Raspberry Pi Foundation im Vereinten Königreich entwickelt und erfreuen sich großer Beliebtheit; bis Juni 2017 wurden beinahe 15 Millionen Exemplare verkauft [37]. Der primär vorhergesehene Verwendungszweck besteht darin, Kindern und Jugendlichen mithilfe eines kostengünstigen Rechners Informatik und Programmierung beizubringen. Die Geräte wurden jedoch auch außerhalb von Bildungseinrichtungen in zahlreichen Projekten wie z.B. zur Automatisierung im Smarthome verwendet. [38]

Auf den Raspberry Pi's lassen sich eine Vielzahl an Betriebssysteme verwenden. Dazu zählen neben FreeBSD und Windows 10 IoT Core auch viele Linuxdistributionen. Hierzu gehört auch Raspbian, eine Abwandlung von Debian, die speziell auf die Rechner zugeschnitten ist.

Für die vorliegende Arbeit wurde ein Raspberry Pi Model B der ersten Generation benutzt. Dieses Modell verwendet die 32-Bit ARMv6Z-Architektur, speziell einen ARM1176JZF-S Einkernprozessor mit einer Taktfrequenz von 700 MHz. Auf dem Gerät sind 512 MB Arbeitsspeicher vorhanden, welche allerdings mit der GPU geteilt werden müssen. Zur Kommunikation mit anderen Geräten verfügt der Raspberry Pi über eine Ethernetbuchse für kabelgebundene Netzverbindungen. Für kabellose Datenübertragungen können USB-basierte WiFi-Adapter verwendet werden.

Bei dem Modell handelt es sich also um ein leistungsstärkeres Gerät als viele andere eingebettete Geräte. Da jedoch auf dem Raspberry Pi Linux unterstützt wird, kann so eine generische Implementierung von WalnutDSA für das Betriebssystem geschaffen werden, die auch auf vielen anderen Geräten mit unterschiedlichen Leistumgsumfängen verwendet werden kann.

9.2 Linux

Linux wurde 1991 von Linus Torvalds ins Leben gerufen und mittlerweile zum wohl weit verbreitesten Betriebssystem-Kernel aufgestiegen. Der Kernel wird in Geräten unterschiedlichster Größenordnungen wie Android-basierten Smartphones, Desktop-PCs, Serversystemen und Supercomputern benutzt, was nicht zuletzt der kostenlosen Verfügbarkeit und der verwendeten GPL2-Lizenz zuzuschreiben ist. Auch im Bereich der eingebetteten Systeme ist das Betriebssystem an oberster Stelle anzutreffen; im IoT-Bereich kommt bereits auf mehr als 80% aller Geräte Linux zu Einsatz. [39]

Bei Linux handelt es sich um einen monolithischen Kernel, für den jedoch separate Module dynamisch zur Laufzeit nachgeladen werden können, um neue Funktionalität bereitzustellen. Die Module verwenden dann nicht den normalen Userspace-Speicherbereich wie normale Anwendungsprogramme, sondern laufen im privilegierten Kontext des Kernels. Die Umsetzung von WalnutDSA geschieht in Form eines solchen Kernelmoduls. Zudem wurde das Verfahren in die sog. *Crypto-API* des Kernels — eine Programmierschnittstelle für kryptographische Verfahren — eingebunden. Für die Implementierung des Moduls wurde Linux 4.14.0 gewählt.

9.3 Implementierung von WalnutDSA

Zur Implementierung von WalnutDSA in C wurde das Verfahren in eine Kernel- und eine Userspacekomponente aufgeteilt. Das Kernelmodul ist für die Signaturgenerierung und -verifizierung verantwortlich und kann über besagte Crypto-API des Linux Kernels angesprochen werden. Das Userspaceprogramm dient zur Generierung von Schlüsselpaaren und kann zugleich dazu verwendet werden, die Funktionen des Kernelmoduls über Systemaufrufe anzusprechen. Das Kernelmodul enthält eine recht generische Implementierung von WalnutDSA in einer eigenen Quelldatei. Das eigentliche Kernelmodul bindet diese dann in die Crypto-API ein und kümmert sich um die Speicherverwaltung bei der Verwendung der Funktionen im Kernel.

Im Folgenden sollen einige Komponenten der Implementierung von WalnutDSA erläutert und dabei auf wichtige bzw. plattformspezifische Details eingegangen werden. Die Teile der Implementierung, welche hier nicht behandelt werden, sind einfache Umsetzungen der theoretischen Definitionen zu C-Quellcode und benötigen keine weitere Erklärung.

9.3.1 Datenstrukturen und Speicherverwaltung

Zur Repräsentierung der Daten, die beim Signaturverfahren verwendet werden, wurden passend große Datentypen verwendet, um nicht unnötig Speicherplatz zu verbrauchen.

Ein Artingenerator wird in einem Byte repräsentiert. Hierfür wird der Datentyp `int8_t` verwendet, welcher Platz für eine ganze Zahl im Intervall -128–127 bietet. Da in der Zopfgruppe B_n der geometrisch betrachtet am weitesten rechts liegende Artingenerator b_{n-1} ist, können hiermit alle Artingeneratoren sowie deren Inversen bis zur Zopfgruppe B_{128} dargestellt werden. Die zugrundeliegende Permutation eines Braids wird in einem normalen Integer-Array repräsentiert. Somit können hiermit auch alle zugehörigen Permutation für die Braids bis zur Zopfgruppe B_{128} dargestellt werden. Da die Permutationen nur zu Berechnungen verwendet werden und nicht in großer Anzahl dauerhaft gespeichert werden müssen, wurde auf die Verwendung von Datentypen kleinerer Größe abgesehen. Bei Verwendung der üblichen Arithmetik auf ganzzahligen Datentypen kleiner als `int` werden diese vor

der Berechnung ohnehin zu einem normalen Integer konvertiert [40, 6.2.1.1]. Für die Colored Burau-Matrizen wurde ein `uint8_t` pro Matrixeintrag verwendet. Das ganzzahlige Intervall 0–255 ermöglicht die Darstellung von allen Werten bis zum endlichen Körper $GF(256)$. Die T-Werte wurden wie auch die Braidpermutationen in einem normalen Integer-Array gespeichert. Die öffentlichen Informationen wurden in einer eigenen Struktur namens `pub_params` zusammengefasst. Diese ist wiefolgt aufgebaut:

```
struct pub_params {
  unsigned int braid_group;
  unsigned int galois_order;
  unsigned int *t_values;
  unsigned int *generators;
};
```

Für die reinen Braidgeneratoren $p_{i,j}$ in `generators` wird nur der Wert i gespeichert und für j stets der Maximalwert n aus B_n angenommen.

Die verwendeten Datentypen `int8_t` und `uint8_t` können im Userspace über die Headerdatei `stdint.h` eingebunden werden. Im Kernel sind diese unter denselben Namen in `linux/types.h` definiert, was zur einfachen Nutzung in beiden Kontexten beiträgt.

In allen Prozeduren der generischen WalnutDSA-Implementierung wurde in den entsprechenden Funktionen für die generierten Datenstrukturen selbst Speicher alloziiert, welcher anschließend vom Funktionsaufrufenden wieder freigegeben werden muss. Die einzige Ausnahme hierzu stellt die E-Multiplikation dar, welche den Speicherbereich der Operanden der Operation auch für die Speicherung des Ergebnisses benutzt.

9.3.2 Generierung von reinen Braids

In einigen Teileprozeduren von WalnutDSA werden reine Braids benötigt. Daher wurde eine eigene Funktion zu Generierung von frei-reduzierten, zufälligen Produkten aus reinen Braidgeneratoren definiert:

```
static void gen_pure_braids(int8_t **dest, unsigned int *dest_size,
    unsigned int min_len, unsigned int n)
{
  int8_t *braid = kcalloc(min_len + PURE_BRAID_SIZE(1, n), sizeof(int8_t), 0);
  unsigned int gen1, gen2; // Random generators being used.
  int reduced_middle = 0; // Flag: Remove one reduced middle generator?
  unsigned int rand = 0;  // Variable to store RNG values.
  unsigned int mark = 0;  // Marker for braid index.
  int i;

  // Start with gen2 for better loop management.
  get_random_bytes(&rand, sizeof(unsigned int));
  gen2 = abs(rand) % (n - 1) + 1;
  for (i = n - 1; i > gen2; i--, mark++)
    braid[mark] = i;

  while (true) {
    // Update generators.
    gen1 = gen2;
    get_random_bytes(&rand, sizeof(unsigned int));
    gen2 = abs(rand) % (n - 1) + 1;

    // Build first generator middle part.
```

```
    for (i = 0 + reduced_middle; i < 2; i++, mark++)
      braid[mark] = gen1;

    if (mark >= min_len)
      break;

    // Build freely reduced start and end connecting the two generators.
    if (gen1 >= gen2) {
      for (i = gen1; i > gen2; i--, mark++)
        braid[mark] = i;
      reduced_middle = 0;
    } else if (gen1 < gen2) {
      for (i = gen1 + 1; i < gen2; i++, mark++)
        braid[mark] = i * -1;
      reduced_middle = 1;
    }
  }

  // Build last generator ending.
  for (i = gen1 + 1; i <= n - 1; i++, mark++)
    braid[mark] = i * -1;

  *dest = braid;
  *dest_size = mark;
}
```

Eine leicht abgewandelte Version dieser Funktion wird auch bei der Kodierungsfunktion verwendet. Die Werte `gen1` und `gen2` definieren wiederum nur die unteren Grenze i für einen reinen Braidgenerator $p_{i,j}$ mit $j = n$ aus B_n. Der generierte Braid wird auch bereits während dem Aufbau frei-reduziert. Durch die Angabe des Funktionsparameters `min_len` kann die Mindestlänge des Braids festgelegt werden. Da mehrere zufällige reine Braidgeneratoren aneinandergehängt werden, kann der Braid jedoch auch größer sein. Daher müssen für den Zweifelsfall immer $length_{min} + 2 \cdot (j_{max} - i_{max})$ Bytes für eine Mindestlänge $length_{min}$ reserviert werden.

9.3.3 Galois-Arithmetik

Für die Arithmetik im endlichen Körper $GF(q)$ existiert eine fremde Softwarebibliothek in C [41]. Diese enthält jedoch Abhängigkeiten zur C-Standardbibliothek, welche nicht im Kernel verwendet werden kann. Da für die Implementierung von WalnutDSA zudem nur zwei Funktionen (Multiplikation und Inverse) benötigt werden, wurde auf die Umschreibung und Einbindung der Bibliothek verzichtet und einfache Versionen der besagten Operationen selbst implementiert.

Die Multiplikation im endlichen Körper wurde mithilfe der *Russischen Bauernmultiplikation* implementiert:

```
uint8_t galois_mult(uint8_t a, uint8_t b, int q) {
  uint8_t result = 0;
  int poly = irr_polys[(int) log2(q) - 5];
  bool overflow;
  while (a != 0) {
    if ((a & 1) != 0)
      result ^= b;
    overflow = (b & (q / 2)) != 0;
    b <<= 1;
    if (overflow)
      b ^= poly;
    a >>= 1;
```

```
    }
    return result;
}
```

Das irreduzible Polynom ist für jeden unterstützten endlichen Körper $GF(q)$ unterschiedlich. Daher wurden die Polynome in einem statischen Array gespeichert, sodass in der Funktion je nach Angabe der Ordnung q des endlichen Körpers das korrekte Polynom verwendet werden kann. Die verwendeten Polynome wurden für optimale Rechenzeit ausgewählt [42].

Die Inverse eines Elements im endlichen Körper wird durch eine Bruteforce-Suche erreicht. Für ein Element a findet man die Inverse, indem man alle möglichen Produkte $a \cdot b$ ausprobiert. Der Wert b, für den das Ergebnis 1 lautet, ist die Inverse des Elements a.

Die Summe und Differenz im endlichen Körper kann durch ein bitweises exklusives Oder realisiert werden, welches schon als Operator `^` in der Programmiersprache C enthalten ist.

9.3.4 E-Multiplikation

Die Funktion `emult(...)` stellt die Implementierung des einfachen Multiplikationsschritts für einen Artingenerator und seine zugehörige Permutation dar. Um die E-Multiplikation auf einem zusammengesetzten Braid zu berechnen, muss man wiederholt diese Funktion anwenden.

Die Umsetzung richtet sich ganz nach der formalen Definition. Zunächst werden die T-Werte permutiert und ggf. die Inverse des T-Werts berechnet. Anschließend konstruiert man die Colored Burau-Matrix und multipliziert sie mit der Matrix aus dem Funktionsparameter `matrix`. Die Permutation des Artingenerators wird mit der aus dem Funktionsparameter `perm` verkettet. Zum Schluss werden die Matrix und Permutation in die Speicherbereiche der Argumente `matrix` und `perm` kopiert und damit die alten Werte überschrieben. Dies ermöglicht eine einfachere Abarbeitung der E-Multiplikation bei zusammengesetzten Braids.

Die Funktion ist sowohl im Kernelmodul als auch im Userspaceprogramm definiert, da man sie auch für die Generierung des öffentlichen Schlüssels benötigt.

9.3.5 Cloaking-Elemente

Auch die Funktion zur Erzeugung von Cloaking-Elementen funktioniert genau gemäß der Beschreibung in Kapitel 8.4. Das erste Stück des Cloaking-Elements muss die Permutation des Braids verändern, damit in dieser die Abbildungen $i \mapsto \sigma^{-1}(a)$ und $i+1 \mapsto \sigma^{-1}(b)$ erfüllt sind. Dies geschieht durch folgendes Quellcodefragment:

```
if (mid <= perm_a && mid < perm_b) {
  for (i = 0; i < perm_a - mid; i++, mark++)
    braid[mark] = perm_a - i - 1;
  if (perm_b < perm_a)
    braid[mark++] = perm_b;
  for (i = 0; i < perm_b - (mid + 1); i++, mark++)
    braid[mark] = perm_b - i - 1;
} else if (mid < perm_b && mid >= perm_a) {
  for (i = 0; i < perm_b - (mid + 1); i++, mark++)
    braid[mark] = perm_b - i - 1;
  for (i = 0; i < mid - perm_a; i++, mark++)
    braid[mark] = perm_a + i;
} else if (mid < perm_a && mid >= perm_b) {
  for (i = 0; i < (mid + 1) - perm_b; i++, mark++)
    braid[mark] = perm_b + i;
  if (perm_a != mid + 1) {
```

```
      for (i = 0; i < perm_a - mid; i++, mark++)
        braid[mark] = perm_a - i - 1;
      braid[mark++] = mid + 1;
    }
  } else {
    for (i = 0; i < (mid + 1) - perm_b; i++, mark++)
      braid[mark] = perm_b + i;
    if (perm_b < perm_a)
      braid[mark++] = perm_a - 1;
    for (i = 0; i < mid - perm_a; i++, mark++)
      braid[mark] = perm_a + i;
  }
```

Hier steht `mid` für die Variable i sowie `perm_a` und `perm_b` für die Werte der inversen Permutation σ^{-1} an den Stellen a und b. Es werden je nach Wert dieser Variablen Artingeneratoren an den Braid gehängt, sodass die Elemente der Permutation an die richtige Stelle ,,geschoben" werden. Der restliche Quellcode der Generierungsfunktion ist trivial und bedarf keiner weiteren Darstellung.

9.3.6 Umformungsfunktionen

Als Umformungsfunktionen wurden wie in der Arbeit zu WalnutDSA die BKL-Normalform und Dehornoys Henkelreduktionsalgorithmus implementiert.

Die Implementierung der links-kanonischen BKL-Normalform richtet sich ganz nach *Efficient Implementation of Braids* [32]. Es wurden die darin vorgestellten Algorithmen von Pseudocode nach C übersetzt. Für die Permutationstabellen und die *Descending Cycle Decomposition Tables* wurden `uint8_t`-Arrays verwendet, um den Speicherplatzverbrauch der Umformungsfunktion zu reduzieren. Es wurden einzelne Funktionen für die Multiplikation, die Berechnung der Inversen und die Äquivalenz von Braids-Permutationen geschrieben. Für die Prozedur zur Berechnung des *Meet* zweier kanonischer Faktoren wurde aufgrund der besseren Laufzeiteigenschaften die Version für Descending Cycle Decomposition Tables gewählt, wobei an entsprechender Stelle zwischen den beiden Repräsentierungen konvertiert wird. Für die Sortierungsoperation im Meet-Algorithmus wurde ein Bucketsort verwendet. Da Braids für WalnutDSA als Artingeneratoren repräsentiert werden, musste zudem der zur links-kanonischen BKL-Normalform konvertierte Braid am Ende der Funktion wieder in ein Produkt aus Artingeneratoren umgewandelt werden.

Für Dehornoys Henkelreduktion existiert eine C-Implementierung für die normale ,,greedy" Version des Algorithmus. Diese ist jedoch sehr unflexibel bezüglich der Speicherverwaltung und hat einige Abhängigkeiten zur C-Standardbibliothek. Daher wurde das Verfahren selbst neu programmiert. Wie in Kapitel 8.7 angesprochen gibt es zwei Varianten des Henkelreduktionsalgorithmus von Dehornoy. Da auf dem verwendeten Raspberry Pi genügend Ressourcen verfügbar sind, wurde die vollständige Reduktion implementiert. Dies führt zu kleineren Signaturen als die normale Version, benötigt jedoch mehr mehr Arbeitsschritte. Zudem kann der Braid während der Reduktion zwischenzeitlich größer als die Ausgangsform werden, wodurch mehr Speicherplatz benötigt wird.

9.4 Einbettung in Linux

Im Folgenden soll die Einbettung der WalnutDSA-Implementierung in Linux dokumentiert werden. Dabei wird zunächst kurz auf das Kernelmodul selbst sowie die Speicherverwaltung

im Kernel eingegangen. Anschließend wird die Einbindung in die Crypto-API dargestellt und ihre Funktionsweise erklärt.

9.4.1 Implementierung des Kernelmoduls

Wie bereits angesprochen bietet Linux die Möglichkeit, neue Funktionalität in Form von Kernelmodulen dynamisch zur Laufzeit nachzuladen. Für die Implementierung wurde ein einfaches Modul mit den entsprechenden Funktionen `module_init` und `module_exit` entworfen, dass als Eintritt in den Kernel dient. Die eigentliche Arbeit wird bei der Implementierung der Funktionen der Crypto-API verrichtet.

Zur Speicherverwaltung stehen eigene Funktionen im Kernel zur Verfügung. Darunter fallen z.B. `kmalloc`, `kcalloc` und `kfree`, welche die Gegenstücke zu den üblichen Funktionen `malloc`, `calloc` und `free` im Userspace bilden. Für die Verwendung des Kernelmoduls vom Userspaceprogramm aus müssen Daten vom einen zum anderen Bereich kopiert werden. Da zusammengehörige Speicherbereiche im Userspace jedoch häufig über den physikalischen Speicher verstreut sind, können sie nicht einfach durch einen simplen Speicherdirektzugriff in den Kontext des Kernels transferiert werden. Daher werden für die Kommunikation zwischen den beiden Bereichen sog. *Scatterlisten* verwendet. Dies sind Strukturen, die eine Abstraktion über im Speicher verstreute Daten bieten und sie als augenscheinlich kontinuierliche Speicherbereiche ansprechbar machen. Dem Benutzer stehen auch einige Hilfsfunktionen zur Verfügung, um Daten einfach in eine Scatterliste zu schreiben und sie wieder aus ihr herauszukopieren. Die relevanten Strukturen und Funktionen sind in `linux/scatterlist.h` definiert.

9.4.2 Eindbindung in die Crypto-API

Die Crypto-API des Linux Kernels stellt eine Programmierschnittstelle bereit, die sowohl für Benutzer von kryptographischen Diensten als auch deren Entwickler ausgerichtet ist. Die durch die API vorhandenen kryptographischen Algorithmen werden *Transformationen* genannt. Dabei kann es sich z.B. um symmetrische und asymmetrische Chiffren, Hashverfahren oder Zufallszahlengeneratoren handeln, welche zum Teil untereinander kombiniert werden können um zusammengesetzte Verfahren wie z.B. einen *Keyed-Hash Message Authentication Code (HMAC)* zu bilden. Für die Transformationen stehen synchrone und asynchrone Schnittstellen zur Verfügung. Die asynchronen Aufrufe können allerdings nur innerhalb des Kernels und nicht vom Userspace aus verwendet werden. [43]

Um eine neue Transformation zur API hinzuzufügen, muss eine entsprechende Registrierungsfunktion aufgerufen werden, der eine Struktur mit den nötigen Informationen der Transformation übergeben wird. Diese Struktur definiert neben Zeigern zu den Implementierungen der entsprechenden API-Funktionen auch den Namen, mit dem das kryptographische Verfahren folglich angesprochen werden kann, sowie die Speichergröße der Kontextstruktur der Transformation. Diese Kontextstruktur wird bei Verwendung der Transformation für den Benutzer erstellt und dient dazu, die zur Operation benötigten Daten über mehrere Funktionsaufrufe hinweg zu speichern. Sie ist wiefolgt aufgebaut:

```
enum operation {
  SIGN,
  VERIFY
};
```

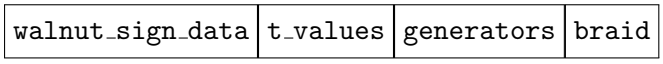
```
union walnut_context {
  enum operation operation;       // Kennzeichnung für die Operation
  struct walnut_sign_data sign_data;   // Daten zur Signierung
  struct walnut_verify_data verify_data; // Daten zur Verifizierung
};
```

Nachdem die Transformation in der Crypto-API registriert wurde, kann der Benutzer über einen Funktionsaufruf ein neues Transformationsobjekt, also eine Instanz der Transformation erzeugen. Anschließend können die verschiedenen Funktionen des Verfahrens verwendet werden.

Zu Beginn wird der Aufruf einer Funktion zur Festlegung des privaten bzw. öffentlichen Schlüssels sowie der damit einhergehenden benötigten Informationen für die Signierung bzw. Verifizierung benötigt. Bei der Signierung wird für die Ansammlung der gefragten Daten die Struktur `walnut_sign_data` verwendet:

```
struct walnut_sign_data {
  int8_t *braid;        // Privater Schlüssel
  unsigned int braid_len;  // Schlüssellänge in Artingeneratoren
  unsigned int a;       // Parameter a für Cloaking-Elemente
  unsigned int b;       // Parameter b für Cloaking-Elemente
  unsigned int sec_level;  // Sicherheitslevel
  enum rewrite_func rewrite; // Kennzeichnung der Umformungsfunktionen
  struct pub_params params; // Öffentliche Informationen
};
```

Wie man sehen kann, enthält die Struktur nur Zeiger zum privaten Schlüssel, den T-Werten und den Kodierungsgeneratoren. Die eigentlichen Daten werden nach folgendem Schema hinter die Struktur im Speicher angehängt:



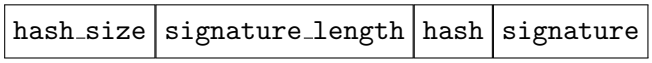
Für die Verifizierung lautet die Struktur `walnut_verify_data`:

```
struct walnut_verify_data {
  uint8_t *matrix;      // Matrix des öffentlichen Schlüssels
  unsigned int *perm;    // Permutation des öffentlichen Schlüssels
  struct pub_params params; // Öffentliche Informationen
};
```

Die Matrix und Permutation des öffentlichen Schlüssels sowie die T-Werte und Kodierungsgeneratoren werden hierbei wiefolgt an die Struktur angehängt:

Die aufbereiteten Daten werden dann an das Kernelmodul übergeben und für die folgenden Operationen gespeichert. Für die Signierung kann anschließend über einen Funktionsaufruf noch die maximale Länge der endgültigen Signatur in Erfahrung gebracht werden. So kann festgestellt werden, wieviel Speicherplatz für den Signaturbraid alloziiert werden muss. Zum

Schluss muss der Anwender die Signierungs- bzw. Verifizierungsfunktion aufrufen, wobei in beiden Fällen ein Nachrichtenhashwert und für die Verifizierung zusätzlich eine Signatur übergeben werden muss. In letzterem Fall müssen beide Datensätze nach folgendem Schema aneinandergehängt werden:



Ist die Operation abgeschlossen, so erhält der Benutzer das entsprechende Ergebnis. Im Falle der Signierung wird die Signatur zusammen mit ihrer Länge in einer Scatterliste in der Kontextstruktur gespeichert, sodass sie anschließend nur noch in den lokalen Speicher kopiert werden muss. Für die Verifizierung erhält der Anwender einen Rückgabewert, der ihn über die erfolgreiche oder fehlgeschlagene Verifizierung informiert.

Nachdem alle Operationen abgeschlossen sind, sollte der Speicher des Transformationsobjekts wieder freigegeben werden. Beim endgültigen Entfernen des Moduls aus dem laufenden Kernel werden alle zwischengespeicherten Daten wieder gelöscht und die Transformation aus der Crypto-API entfernt.

Die Funktionen der Crypto-API werden über Systemaufrufe dem Userspace zugänglich gemacht. Für die asymmetrischen Transformationen ist ein solcher Systemaufruf jedoch nicht im Mainline-Kernel vorhanden. Daher muss ein selbstkompilierter Kernel verwendet werden, für den drei verschiedene Patches [44] eingespielt sowie eine Reihe an Optionen in der Kernelkonfiguration [45] aktiviert werden müssen. Die Kernelpatches sind auf die Version 4.14.0 von Linux ausgelegt, weswegen auch genau diese Version zur Implementierung von WalnutDSA gewählt wurde. Es steht zudem die Softwarebibliothek *libkcapi* [46] zur Verfügung, welche eine einfache Programmierschnittstelle über den besagten Systemaufrufen der Crypto-API bietet. Diese wurde für die Userspacekomponente der Implementierung von WalnutDSA eingesetzt. Die Bibliothek übernimmt die komplette Kommunikation mit dem Kernel, inklusive dem Transferieren der Daten über Scatterlisten, sodass der Anwender sich nicht mit der Kernel-internen Speicherverwaltung auseinandersetzten muss.

Es sei gesagt, dass die Crypto-API für asymmetrische Kryptosysteme auch Funktionen zur generischen Ver- und Entschlüsselung von Daten bereitstellt. Da diese Operationen jedoch nicht für WalnutDSA definiert sind, wurden die entsprechenden Funktionen nicht implementiert. Beim Aufruf wird der Benutzer über eine Fehlermeldung auf die fehlende Unterstützung hingewiesen.

9.5 Evaluierung der Implementierung

Die vorgestellte Implementierung soll nun evaluiert werden. Hierbei wird die Korrektheit des Verfahrens sowie ihre fehlerfreie Speicheralloziierung untersucht. Es wird auf die potentielle Kompatibiliät zu anderen Implementierungen und die Erfüllung von gängigen Sicherheitsanforderungen eingegangen. Zudem wird der Speicherplatzverbrauch und das Laufzeitverhalten der Implementierung analysiert.

9.5.1 Korrektheit der Implementierung

Um die Korrektheit der Implementierung zu überprüfen, wurden drei verschiedene Tests unternommen. Zunächst wurde die Implementierung an sich selbst getestet, indem X Signaturen mit jeweils zufällig generierten Parametern und Schlüsselpaaren erstellt wurden. Bei der Hälfte der Signaturen wurde ein zufälliger Fehler in den Datensatz eingeführt, um die Verifizierung zu vereiteln. Bei der Verifizierung der unmodifizierten Signaturen war diese in jedem Fall erfolgreich, bei den verfälschten Signaturen schlug sie in jedem Fall fehl.

SecureRF bietet ein IoT-SDK an, welches eine Reihe an Implementierungen von WalnutDSA im Binärformat beinhaltet. Leider konnte diese Software jedoch nicht für die Arbeit beantragt werden. Die wissenschaftliche Arbeit zu WalnutDSA enthält jedoch einen beispielhaften Datensatz zur Signierung und Verifizierung. Die Implementierung wurde mit diesem Beispiel getestet und erhielt die korrekten Ergebnisse.

Um die Kompatibilität mit anderen Implementierungen sicherzustellen, wurde ein Testaufbau mit einem ESP8266 und einem Arduino M0 Pro entworfen. Dieser wird in Kapitel 12.1.1 vorgestellt.

9.5.2 Fehlerfreie Speicherverwaltung

Für eine sichere Implementierung muss gewährleistet werden, dass die Speicherbereiche aller verwendeten Daten nach der Benutzung wieder freigegeben werden, sodass keine überbleibenden privaten Informationen mehr im Arbeitsspeicher liegen. Daher wurde sichergestellt, dass keine Speicherlecks in der Implementierung von WalnutDSA auftreten.

Für die Userspacekomponente wurde hierfür das Memcheck-Programm von *Valgrind* [47] eingesetzt. Diese Software klinkt sich in die Speicherverwaltung der C-Standardbibliothek ein und untersucht die Gültigkeitsdauer und Addressierbarkeit der verwendeten Speicherbereiche. Beim Testen des Programms kamen keinerlei Speicherfehler auf.

Zur Überprüfung des Kernelmoduls wurde der *Kernel Memory Leak Detector* [48] eingesetzt. Dieser bietet äquivalent zu Valgrind eine Möglichkeit, Speicherlecks im Kernelquellcode zu finden. Der Test des implementierten Kernelmoduls lieferte auch keine Fehler.

9.5.3 Kompatibilität und Sicherheitsparameter

Da auf dem verwendeten Raspberry Pi genügend Arbeitsspeicher zur Verfügung steht, konnte dem Benutzer die Wahl der zu verwendenden Zopfgruppe B_n und die Ordnung des endlichen Körpers $GF(q)$ weitestgehend freigelassen werden. Die beiden Werte sind nur durch die Verwendung von einem Byte pro Artingenerator bzw. Element im endlichen Körper beschränkt. Somit ergibt sich eine hohe Kompatibilität des implementierten Verfahrens mit potentiellen anderen Implementierungen.

Auch die Wahl des Sicherheitslevels ist nicht durch die Implementierung eingeschränkt. Sowohl die Mindestlänge der Cloaking-Elemente als auch die des privaten Schlüssels werden dynamisch zur Laufzeit des Programms berechnet, wodurch dem Wert des Sicherheitslevels keine künstlichen Grenzen gesetzt sind. Sollten zukünftig höhere Anforderungen bezüglich der Mindestlängen dieser beiden Teile gefordert werden, so kann die Implementierung ohne Änderungen weiterverwendet werden. Das *Bundesamt für Sicherheit in der Informationstechnik* empfiehlt bspw. ab 2022 ein Sicherheitslevel von 128 Bits für die bekannten Verfahren (also z.B. AES-128 oder RSA mit 3072-Bit-Schlüsseln) [49]. Unter der Annahme, dass sich die Empfehlungen des Sicherheitslevels auf WalnutDSA übertragen lassen und keine für

das Verfahren relevanten Angriffe gefunden werden, kann die Implementierung auch nach 2022 noch problemlos diese Sicherheitsanforderungen erfüllen.

9.5.4 Speicherplatzverbrauch und Laufzeitverhalten

Zur Untersuchung der Effizienz der Implementierung wurden der benötigte Speicherplatz und die verwendete Rechenzeit des Verfahrens gemessen.

Der öffentliche Schlüssel ist unabhängig vom verwendeten Sicherheitslevel konstant in seiner Größe. Durch die Verwendung von einem Byte pro Matrixeintrag und einem `unsigned int` (welches auf der verwendeten Plattform 4 Bytes entspricht) pro Permutationseintrag ergibt sich dafür ein Speicherplatzverbrauch von $n^2 + 4n$ Bytes bei Verwendung der Zopfgruppe B_n. Ein Artingenerator wird in einem Byte repräsentiert. Die Speicherung des privaten Schlüssels benötigt bei der Einhaltung seiner Mindestlänge l also l Bytes in Abhängigkeit des Sicherheitslevels. Für eine Signatur mit der Länge l_{sig} ergibt sich analog ein Speicherplatzverbrauch von l_{sig} Bytes.

Zur Analyse der benötigten Rechenzeit wurden Schlüsselpaare für die Sicherheitslevel 112, 138 nud 256 generiert. Mit jedem Schlüsselpaar wurde eine Signatur für einen zufälligen 256-Bit-Hashwert gebildet. Die Zeit zur Generierung des Hashwerts wurde nicht in die Ergebnisse mit einbezogen, da die zur Verfügung stehenden Vergleichsdaten dies auch nicht beinhalten (s. Kapitel 12.1.2). Die Signaturen wurden anschließend verifiziert. Diese Messungen wurden zum Vergleich jeweils ganz ohne Umformungsfunktion, nur mit der BKL-Normalform und mit der BKL-Normalform und Dehornoys Henkelreduktionsalgorithmus durchgeführt. Es wurde stets der endliche Körper $GF($256 eingesetzt, da dieser die Rechenzeit nicht sonderlich stark beeinflusst. N Es wurde der Speicherplatzverbrauch und die Rechenzeit jeder Signatur dokumentiert. Die Ergebnisse sind im Folgenden dargestellt:

SL	l	L	BKL	Deh	min. l_{sig}	max. l_{sig}	∅ l_{sig}	T(Key)	T(Gen)	T(Ver)
112	86	8	nein	nein	696	772	726	7,79 ms	141 μs	92,6 ms
			ja	nein	1544	3168	2551		3,18 s	233 ms
			ja	ja	594	750	659		5,31 s	94,4 ms
128	101	9	nein	nein	682	822	750	9,64 ms	155 μs	89,4 ms
			ja	nein	1700	3018	2362		4,05 s	263 ms
			ja	ja	626	836	704		6,77 s	96,6 ms
256	222	18	nein	nein	1042	1126	1076	22,7 ms	214 μs	133 ms
			ja	nein	2962	4332	3639		7,85 s	394 ms
			ja	ja	1030	1302	1127		18,7 s	135 ms

Wie man sehen kann, nimmt die Umformung der Signatur zur BKL-Normalform die meisten Ressourcen in Anspruch und verzeichnet einen starken Anstieg in der benötigten Rechenzeit. Auch der Speicherplatzverbrauch wird durch diese Funktion stark erhöht, was jedoch durch die anschließende Anwendung von Dehornoys Henkelreduktionsalgorithmus wieder rückgängig gemacht werden kann. Zum Teil ist die Signatur nach der Reduktion sogar kleiner als die ursprüngliche Form vor der Umformung. Wie dargestellt wurde, kann jedoch auf die BKL-Normalform nicht verzichtet werden, da sie essentiell für die Vermischung der einzelnen Elemente der Signatur ist. Eine andere, effizientere Umformungsfunktion könnte dieses Problem beheben. Derzeit ist jedoch keine Alternative bekannt. Die für die Verifizie-

rung benötigte Rechenzeit steigt linear mit dem Sicherheitslevel und der Länge der Signatur. Dies war gemäß den Eigenschaften der E-Multiplikation zu erwarten. Auch die Zeit für die Signaturgenerierung steigt linear mit der Braidlänge.

Zusätzlich wurde die beschriebene Messung mit den Zopfgruppen B_{12} und B_{16} für das Sicherheitslevel 128 durchgeführt, um eine Vergleich bezüglich der verwendeten Zopfgruppen zu bilden. Dies führte zu folgenden Ergebnissen:

n	BKL	Deh	min. l_{sig}	max. l_{sig}	$\varnothing$ l_{sig}	T(Key)	T(Gen)	T(Ver)
8	nein	nein	682	822	750	9,64 ms	155 μs	89,4 ms
	ja	nein	1700	3018	2362		4,05 s	263 ms
	ja	ja	626	836	704		6,77 s	96,6 ms
12	nein	nein	876	1030	932	24,7 ms	148 μs	364 ms
	ja	nein	2900	3950	3535		6,55 s	1,15 s
	ja	ja	604	880	754		11,3 s	313 ms
16	nein	nein	872	1184	1013	68,8 ms	134 μs	820 ms
	ja	nein	3786	5370	4555		11,1 s	3,27 s
	ja	ja	606	848	765		14,1 s	766 ms

Man kann erkennen, dass steigen die benötigte Rechenzeit und der verwendete Speicherplatz für die Signatur weitestgehend linear mit der Anzahl an Strähnen in der Zopfgruppe. Die Zeit für die Verifizierung wächst allerdings weniger stark, während die Schlüsselgenerierung einen höheren Anstieg verzeichnet.

10 WDSA Implementierung für FreeRTOS

Nach genauer Betrachtung der Funktionsweise von WalnutDSA wird im folgenden Kapitel näher auf die Details der Implementierung der WDSA-Bibliothek für FreeRTOS eingegangen.

10.1 FreeRTOS

Mit einer seit über 12 Jahren wachsenden Community, 35 offiziell unterstützten Architekturen und über 113000 Downloads im Jahr 2014 ist FreeRTOS eine der marktführenden Echtzeitbetriebssysteme.[50] Der spezifische Verwendungszweck und die limitierten Ressourcen eingebetteter Systeme sind Punkte, die eine Implementierung eines vollständigen und wiederverwendbaren Betriebssystems erschweren. *Real Time Engineers Ltd.* bietet mit FreeRTOS daher lediglich eine Implementierung der Kernelfunktionalitäten, wie Taskverwaltung, die Kommunikation zwischen den Tasks, Timing und Synchronisation, an. Der Kern der Implementierung besteht dadurch nur aus drei wiederverwendbaren, plattformunabhängigen C-Files.[51] Ohnehin schon leicht zu bedienen, lässt sich die Arbeitsweise des Kernels in den Konfigurationsdateien einfach für die jeweilige Anwendung anpassen.[52] Zusätzliche Features, wie Netzwerkfähigkeit oder kryptographische Funktionen müssen importiert werden.

10.1.1 Tasks

Das (pseudo-) parallele Bearbeiten verschiedener Aufgaben auf eingebetteten Systemen wird auf FreeRTOS mithilfe sogenannter Tasks realisiert. Ähnlich wie Threads handelt es sich hierbei um voneinander unabhängige Befehlsfolgen, die abwechselnd auf einem Prozess ausgeführt werden. Anders als übliche Threads verfügt jeder Task über seinen eigenen Stack, in den sämtliche CPU-Register bei einem Kontextwechsel abgespeichert werden. Die Tasks wissen nicht voneinander und die Verantwortung des Kontextwechsels liegt allein beim *FreeRTOS Scheduler*. [53] Der Scheduler vergibt das Recht auf Rechenzeit abhängig von Status und Priorität der Tasks, sodass unter den Tasks im Zustand *ready*, derjenige mit der höchsten Priorität immer arbeiten kann. Verfügen zwei oder mehr Tasks über dieselbe Priorität, wird die Rechenzeit per Round Robin mit gleichen Zeitspannen aufgeteilt. Die anderen Tasks warten währenddessen entweder im Zustand *ready* auf Rechenzeit, im Zustand *blocked* auf ein Event oder im Zustand *suspended* darauf, von einem anderen Task wieder aufgeweckt zu werden. Da FreeRTOS hauptsächlich für den Echtzeitbetrieb entwickelt wurde, darf die Funktion nicht durch *return* oder *exit* terminieren, jedoch kann sich ein Task selber beenden [54].

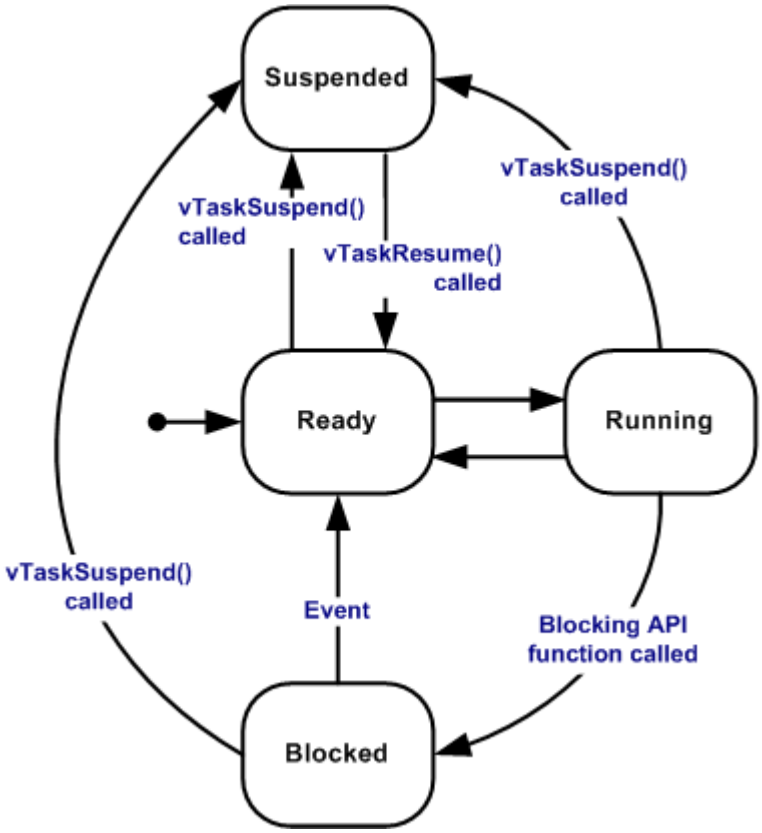


Abbildung 10.1: Zustände eines FreeRTOS Task

10.1.2 Queues

Für die Kommunikation der einzelnen Tasks untereinander werden hauptsächlich *Queues* benutzt, eine separate Datenstruktur, die nach dem FIFO-Prinzip arbeitet und deren Speicherallozierung vom Kernel übernommen wird. Will ein Task Daten an einen anderen Task senden, dann kann er Variablen in eine Queue kopieren und direkt danach weiterverwenden. Für Nachrichten variabler Länge besteht die Möglichkeit, Structs in der Queue zu lagern. Dabei handelt es sich bei einem Member um einen Pointer auf die Daten und beim anderen um eine Variable, in der die Länge der Nachricht gespeichert wird. Abhängig von der vordefinierten Anzahl an Elementen, die eine Queue tragen kann, kann sie auch als Binär- oder Zählsemaphore verwendet werden. [55]

10.1.3 RTOS Tick

Manche Funktionen von FreeRTOS sind zeitabhängig, unter anderen die Methode $vTaskDelay()$, welche einen Task für eine bestimmte Zeit in den rechenzeitlosen Zustand *blocked* versetzt. Um die Zeit zu messen, verwendet FreeRTOS eine Zählvariable, die bei jedem Auftreten des „*RTOS tick interrupt's*“ um 1 inkrementiert wird. Das Zeitintervall zwischen zwei Interrupts ist vordefiniert und mit einer hohen zeitlichen Genauigkeit konstant. [56]

10.2 WDSA-Bibliothek

In diesem Teilkapitel wird der Hauptteil der Implementierung von WalnutDSA vorgestellt und die wichtigsten Aspekte der Umsetzung der theoretischen Teiloperationen beschrieben.

10.2.1 FreeRTOS Speicherverwaltung

Auch wenn FreeRTOS so gut wie alle Funktionen des C99-Standards, und damit auch dynamische Speicherverwaltung, unterstützt, wird vom Benutzen der Standardfunktionen zur Speicherallokation *malloc()* und *free()* abgeraten, da diese auf eingebetteten Systemen weder immer verfügbar noch threadsicher sind und mehr Speicher als notwendig brauchen. Daher bietet *Real Time Engineers Ltd.* mehrere eigene Bibliotheken zur dynamischen Speicherverwaltung an. Für die WDSA-Bibliothek wurde eine Variante ausgesucht, die benachbarte, freigegebene Speichersegmente kombiniert, um schlechter Fragmentierung vorzubeugen. Die wichtigsten Funktionen heißen *pvPortMalloc()* und *vPortFree()*. Im Gegensatz zu den Versionen der anderen Bibliotheken sind die Funktionen nicht deterministisch, dennoch immer noch effizienter als ihre Standard-Varianten und genauso zu benutzen.[57]

10.2.2 Datenstrukturen der Braids

Die Generierung und Verifizierung der Signaturen wird von vielen Parametern beeinflusst. Vor allem die finale Braidlänge, die Braidgruppe N und die Größe des Binärkörpers der CB-Einträge q wirken sich stark auf sowohl Laufzeit, Speicherverbrauch als auch das erreichte Sicherheitslevel aus. Aufgrund des linearen Wachstums der Laufzeit der E-Multiplikation mit wachsender Braidlänge, kann auch bei klein gewähltem N und q ein gutes Sicherheitslevel mithilfe eines längeren Braids erreicht werden, ohne dass dabei die Laufzeit beträchtlich darunter leidet. In der Implementierung werden q und N festgesetzt und dafür sämtliche Parameter, die nur die Braidlänge bestimmen, variabel gelassen. Dadurch wird etwas Flexibilität eingetauscht für die Möglichkeit durch Strukturen die beschränkten Ressourcen besser ausnutzen zu können.

Artinbraids

Die Braids werden (außer bei der Berechnung der BKL-Form) durchgehend als Kette von Artin-Generatoren abgespeichert. Dabei benötigt jeder Generator $log_2(2N)$ Bits. Für $N = 8$ sind es daher $log_2(16) = 4$ Bits. Für das Abspeichern längerer Braids bieten sich Bitfelder an.

```
struct artinTuple{
  int8_t gen1 : 4;
  int8_t gen2 : 4;
};

struct braid{
  struct artinTuple *tuples;
  uint16_t length;
};
```

CB-Matrizen

Mit $N = 8$ besitzt eine CB-Matrix 64 Einträge. Mit jeweils 1 Byte an verfügbaren Speicherplatz, kann jeder Eintrag $q = 2^8 = 256$ Werte annehmen. Da die E-Multiplikation mit Tupel aus CB-Matrix und Permutation arbeitet, ist hier ebenso die Verwendung von Strukturen sinnvoll.

```
struct CB {
  unsigned char** matrix;
  int8_t* permutation;
};
```

10.2.3 Implementierung der E-Multiplikation

Handelt es sich beim 2. Faktor der E-Multiplikation um einen Braid aus mehreren Artin-Generatoren, so erfolgt die Multiplikation iterativ mit jedem einzelnen Generator.(siehe 6.3) Im folgenden Ausschnitt aus der Funktion $eMultiplication()$ wird die gesamte Multiplikation in Teiloperationen zerlegt:

```
//..//
//einmalige Berechnung der Inversen der T-Werte
uint8_t *inverseTValues = computeInverseTValues(tValues);

//iterative E-Multiplikation
uint16 numberOfTuples = getNumberOfTuples(braid.length);
for(uint16 i=0; i< numberOfTuples-1; i++) {
  result = eMultiPart(result, braid.tuples[i].gen1, tValues, inverseTValues);
  result = eMultiPart(result, braid.tuples[i].gen2, tValues, inverseTValues);
}
//Fall ungerade Anzahl an Generatoren: last.gen2=0 beachten
result = eMultiPart(result, braid.tuples[numberOfTuples-1].gen1, tValues,
    inverseTValues);
if(braid.length%2 == 0) result = eMultiPart(result,
    braid.tuples[numberOfTuples-1].gen2, tValues, inverseTValues);
//...//
```

In der Unterfunktion $eMultiPart()$ wird mit einer weiteren Funktion $getCB()$ die Colored Burau Form des Artin-Generators erzeugt (siehe 6.4). Dazu wird zuvor der einzusetzende T-Wert bestimmt.

```
//...//
//Bestimmen des T-Werte
int8 tIndex = b-1;
uint8_t tValue;
if(b<0) tIndex = b*(-1);
tIndex = cb1.permutation[tIndex];    //Anwenden der Permutation
if(b<0) tValue = inverseTValues[tIndex];
else tValue = tValues[tIndex];

struct CB cb2 = getCB(b, tValue);    //Erzeugen der CB-Matrix des 2. Faktors
//...//
```

Ist die Colored Burau Form beider Faktoren bekannt, erfolgt der eigentliche Multiplikationsschritt. Da es sich bei den Einträgen der CB-Matrix um Elemente im Binärkörper handelt,

müssen die Addition und Multiplikation der Matrixmultiplikation angepasst werden (siehe 1).

```
//.../
uint8_t result;
for(int8 i=0; i<N; i++){
  for(int8 j=0; j<N; j++){
    result = 0;
    for(int8 k=0; k<N; k++) {
      result = result ^ galMult(cb1.matrix[i][k], cb2.matrix[k][j]);
    }
    matrix[i][j] = result;
  }
}
//.../
```

Bei $galMult()$ handelt es sich um eine Implementierung des „$RussianPeasantAlgorithm$“. Bei der Wahl des irreduziblen Polynoms wird darauf geachtet, dass es effizient in der Multiplikation verwendet werden kann. [42, p. 3]

10.2.4 Generieren der Cloaking-Elemente

Die Funktion $generateCloakingElement()$ nimmt als Eingabe die Permutation des privaten Schlüssels σ, L und die geheimen Parameter a und b. Zunächst wird ein positives $i < N$ zufällig gewählt und $aPos = \sigma^{-1}(a)$ und $bPos = \sigma^{-1}(b)$ bestimmt. Zur Erzeugung eines Teilbraids, der $aPos$ nach i und $bPos$ nach $i + 1$ verschiebt, wird mithilfe verschachtelter Abfragen eine Reihe von Artin-Generatoren gewählt, sodass die Permutation des Teilbraids auch in komplizierteren Fällen, wie z.B. wenn sich die Stränge von $aPos$ und $bPos$ kreuzen müssen, stimmt. Es folgen zufällig generierte, reine Braids, die solange erzeugt und konkateniert werden, bis der Teilbraid die Länge L erreicht hat. Zum Schluss wird der bisherige Braid mit zweimal den Artin-Generator i und seinem Inversen verknüpft (siehe 8.4).

10.2.5 Kodierung des Hashs

Die Funktion $encodeHash()$ nimmt als Eingabe einen Hash der zu signierenden Daten und 4 Kodierungsgeneratoren. Der Hash sollte das Ergebnis einer guten kryptographischen Hashfunktion sein, um ein homomorphes Verhalten der Kodierung zu vermeiden. Die Hashlänge ist im Header *wdsa.h* unter HASHSIZE als 32 (Byte) definiert, kann aber nach eigenem Ermessen angepasst werden. Die vier Generatoren gehören zu den öffentlichen Informationen und müssen wie der öffentliche Schlüssel ausgetauscht werden.

$encodeHash()$ zerlegt jedes Byte des Hashwerts mithilfe von Bitshifts is zwei Hexadezimalzahlen, die jeweils wieder in zweimal 2 Bit aufgeteilt werden, wobei das niedrigere Bitpaar aus den vier Generatoren einen auswählt und das höhere bestimmt, wie oft der Generator benutzt wird. Eine Unterfunktion erzeugt wie bei den Cloaking-Elementen die reinen Braids. 8.7 Beim Zusammensetzen der reinen Braids wird der resultierende Braid frei reduziert (siehe 8.3).

10.2.6 Implementierung der Umformungsfunktionen

Nach dem Erzeugen der Rohsignatur, muss eine Umformungsfunktion auf den gesamten Braid angewendet werden. Die Autoren des WDSA-Papers schlagen dafür den *Birman-Ko-Lee-* und den *Dehornoy-Algorithmus* vor und verwenden beide in ihrer Beispielsignatur. BKL verändert viele Generatoren der Signatur und fügt viele hinzu, woraufhin Dehornoy den Braid wieder auf eine annehmbare Länge kürzt. Daher stellt eine Kombination aus beiden Algorithmen eine effektive und im Bezug auf den Speicher sparsame Möglichkeit, die Signatur, insbesondere den privaten Schlüssel, zu verschleiern, dar [31, p. 16].

Permutations-Braids

Die Umformung der Rohsignatur in ihre BKL-Normalform besteht zum Großteil aus Operationen auf Permutations-Braids, Zahlenreihen der Länge N[35]. Bei der Umformung der Rohsignatur der Länge $l > 1000$ würde der naive Ansatz, die Permutationen in ein zweidimensionales Array abzuspeichern kurzfristig mehr als 8000 Bytes Speicherplatz kosten. Um eine Permutation als Zahlenreihe darzustellen werden aber theoretisch nur $log_2(N) * N$ benötigt. Mit einem konstanten $N = 8$ besteht die Möglichkeit mithilfe von Bitfeldern eine Permutation in 3 Byte zu speichern, wodurch der benötigte Speicherplatz auf weniger als die Hälfte reduziert wird.

```
struct permutation{
  unsigned char pos1 : 3;
  unsigned char pos2 : 3;
  unsigned char pos3 : 3;
  unsigned char pos4 : 3;
  unsigned char pos5 : 3;
  unsigned char pos6 : 3;
  unsigned char pos7 : 3;
  unsigned char pos8 : 3;
};
```

Trotz Komprimierung benötigt ein 3 Byte Permutations-Braid jedoch immer noch 6 mal soviel Speicherplatz wie ursprünglich der Artin-Generator.

In *wdsa.h* ist unter REWRITE definiert, ob die Signatur vollständig, nur mit BKL oder nicht umgeformt werden soll.

10.2.7 Verifizierung

Bei der Verifizierung wird der kodierte Hashwert mithilfe der E-Multiplikation zunächst in seine CB-Repräsentation umgewandelt und daraufhin seine Matrix mit der Matrix des öffentlichen Schlüssels multipliziert. Dieser wird anschließend ebenso mit der Signatur e-multipliziert, woraufhin beide Teilergebnisse miteinander verglichen werden. Sind die Matrizen identisch, ist die Verifizierung positiv, ansonsten negativ.

```
//...//
//Colored Burau Form des kodierten Hashs
struct CB hashCB = eMultiStart(idCB, encodedHash, tValues);

//Matrixmultiplikation von hashCB und öffentlichen Schlüssel -> Teilergebnis 1
uint8_t matrix[N][N];
for (int8 i = 0; i < N; i++) {
  for (int8 j = 0; j < N; j++) {
    uint8_t result = 0;
    for (int8 k = 0; k < N; k++) {
      result = result ^ gmul(hashCB.matrix[i][k], publicKey.matrix[k][j]);
    }
    matrix[i][j] = result;
  }
}

//E-Multiplikation vom öffentlichen Schlüssel und Signatur -> Teilergebnis 2
struct CB pubSig = eMultiStart(publicKey, signatureBraid, tValues);

//Vergleich der Teilergebnisse. Wenn gleich -> verifiziert
_Bool verified = 1;
for (int8 i = 0; i < N; i++) {
  for (int8 j = 0; j < N; j++) {
    if (pubSig.matrix[i][j] != matrix[i][j]) {
      verified = 0;
    }
  }
}
//...//
```

10.3 Implementierung auf dem ESP8266

Dieser Abschnitt soll die Verwendung der WDSA-Bibliothek auf einem eingebetteten System anhand einer möglichen Implementierung auf dem ESP8266 demonstrieren.

10.3.1 ESP8266 - Technische Details

Ein wichtiger Faktor im Bereich der IoT neben Stromverbrauch, Größe und Speicherplatz ist die Konnektivität. Beim ESP8266 handelt es sich um einen MCU der chinesischen Firma *Espressif* mit integrierter Wlan-Funktionalität und bietet daher vor allem in Kombination mit anderen Architekturen, wie z.B. Arduinos, eine beliebte Möglichkeit, ein Netzwerk aus miteinander kommunizierenden Mikrocontroller-Einheiten aufzubauen. Allein verfügt der ESP8266 über einen *Tensilica L106 32-bit Microcontroller* mit einer Taktfrequenz zwischen 80-160 MHz, einem 64 KB großen Befehlsspeicher (RAM), einem 96 KB großen Datenspeicher (RAM) und einen mehrere MB großen Flash, sowie über UART und SPI Schnittstellen für den Datenaustausch. Zum Testen der Implementierung wurde ein *NodeMCU Lua Amica V2* benutzt.[58]

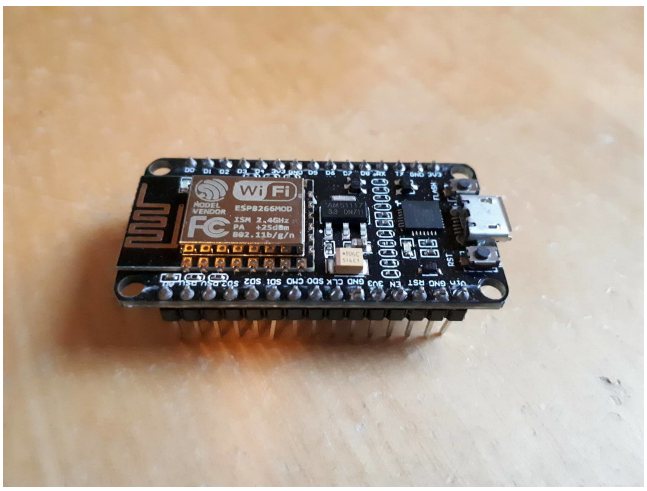


Abbildung 10.2: NodeMCU Lua Amica V2

10.3.2 Portierung der WDSA-Bibliothek

Applikationen, die FreeRTOS verwenden sind bis auf die Port-Files zum Großteil Hardware-unabhängig. Daher lässt sich die WDSA-Bibliothek leicht auf den ESP8266 oder auf andere unterstützte 32-Bit-Maschinen portieren.

10.3.3 Software externer Quellen

Für den ESP8266 existiert kein offizieller FreeRTOS Port. Das Open-Source-Projekt „*ESP-Open-RTOS*“beinhaltet neben einer Version von FreeRTOS auch die benötigten Files für die Portierung.

Um den Quellcode zu kompilieren und die Binary über die UART-Schnittstelle auf den Flash zu laden, wurde das „*Xtensa lx106 architecture toolchain*“ des „ESP-Open-SDKs“ benutzt.

Sicherlich signifikant für die Sicherheit ist die Wahl des Zufallszahlengenerators. Bei den „*Wdev Control Registers*“ handelt es sich um eine Speicherregion des ESP8266s, welche Register für die speichergebundene Adressierung des Wifi-Moduls enthält. Nach Angaben des Herstellers ändert sich ihr Inhalt schnell und unvorhersehbar genug, um als Quelle für Zufallszahlen dienen zu können.[59]

Um den Nachrichtenaustausch per WLAN zu realisieren nimmt die Implementierung Gebrauch von der Wifi-Bibliothek des ESP8266, um ein WPA2-gesichertes WLAN-Netzwerk zu erzeugen. Zum Versenden und Empfangen von UDP-Paketen wurde der Netzwerkstack *„lwIP“* (lightweight IP) importiert. Ursprünglich entwickelt von Adam Dunkels am *Swedish Institute of Computer Science* handelt es sich dabei um eine RAM-sparsame und damit speziell für eingebettete Systeme implementierte Version des TCP/IP Stacks, die heutzutage in vielen Geräten Anwendung findet.

10.3.4 Aufteilen der Dienste auf verschiedene Tasks

Da die genannten Dienste in vielen Fällen voneinander unabhängig sind, macht es Sinn, sie auf verschiedenen Tasks laufen zu lassen . Der Task *listenTask* bearbeitet eingehende Anfragen per UDP. Die Schlüsselgenerierung, der einfachste der drei Dienste, wird vom *listenTask* selbst durchgeführt. Für die Verifikation und die viel länger dauernde Signaturgenerierung gibt es zwei weitere Tasks *verificationTask* und *generationTask*. Wird eine der beiden längeren Dienste angefordert, leitet *listenTask* die jeweiligen Parameter an die zuständigen Tasks über eine Queue weiter.

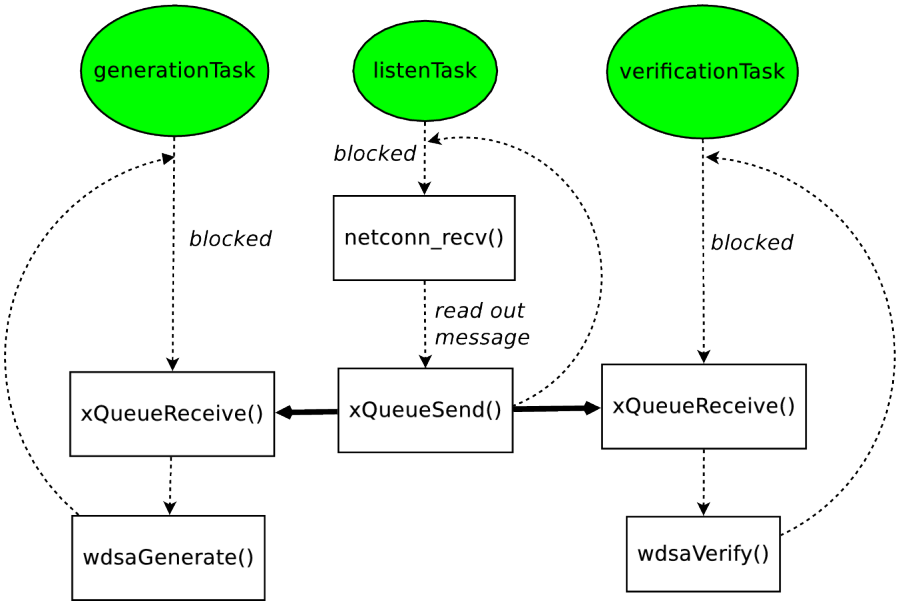


Abbildung 10.3: Kommunikation zwischen den Tasks

10.4 Evaluierung der Implementierung

Der ESP8266 nimmt für die Simulierung des praktischen Einsatzes von WalnutDSA die Serverrolle für die Dienste Schlüsselgenerierung, Signaturgenerierung und Verifizierung an. In verschiedenen Tests wird die Implementierung auf Korrektheit, Speicherplatzanforderungen und Laufzeit getestet.

10.4.1 Testaufbau

Im Testbetrieb wartet der ESP8266 auf einkommende UDP-Pakete mit Nachrichten folgender Form (Leerzeichen dienen nur der Übersichtlichkeit):

- „genKey“ # securityLevel
- „genSig“ # securityLevel # Kodiergeneratoren # tValues # Hashwert

- „verSig“ # Kodierungsgeneratoren # tValues # Hashwert # öffentlicher Schlüssel: Matrix # öffentlicher Schlüssel: Permutation # Signaturlänge # Signatur
- „genRand“

Die Nachrichten werden zerlegt und abhängig vom ersten Segment weiterverarbeitet.

„genKey“ Es wird ein privater Schlüssel erzeugt, der mindestens das angegebene Sicherheitslevel erfüllt und im Flash gespeichert, sodass er später beliebig oft verwendet werden kann.

„genSig“ Aus dem Sicherheitslevel wird die Schlüssellänge l und L berechnet. Ist ein ausreichend langer Schlüssel im Flash gespeichert, wird dieser weiter verwendet, ansonsten wird ein neuer Schlüssel generiert. Anhand der T-Werte werden a und b bestimmt. Mit den gegebenen Parametern wird eine Signatur erstellt und bei erfolgreicher Verifizierung zusammen mit dem öffentlichen Schlüssel zurück an den Client geschickt.

„verSig“ Verifizierung der Signatur mit angegebenen Parametern.

„genRand“ Es werden zufällige Signaturen mit Sicherheitslevel 128 erzeugt und zurück gesendet.

Die vom ESP8266 für die Tests generierten Signaturen wurden alle von einer zweiten, in der Funktionsweise unveränderten Implementierung auf einem Notebook erfolgreich verifiziert.

10.4.2 Fehlerfreie Speicherallokation

Das Tool zur Prüfung der Speicherverwaltung *„Memcheck“* von *„Valgrind“* hat in der WalnutDSA-Bibliothek keine Speicherlecks entdeckt. Auch nach längerem Betrieb und Generieren von mehr als 100 Signaturen nahm der freie Speicherplatz auf dem Heap nach jeder Iteration nicht ab, weshalb von einer fehlerfreien dynamischen Speicherverwaltung ausgegangen werden kann.

10.4.3 Sicherheit der Implementierung

Die zentralen Faktoren für die Sicherheit des Verfahrens sind die Parameter *N, q, l* und *L*. In der Implementierung wurde N konstant mit 8 und q mit 256 belegt. Da die letzte Zeile der Matrix des öffentlichen Schlüssels immer bis auf das letzte Element nur aus Nullen besteht und zwei 1en in den T-Werten dafür sorgen, dass bei der E-Multiplikation viele Einträge kopiert werden, gibt es schätzungsweise $q^{N*(N-3)} = 256^{40} = 2^{320}$ mögliche öffentliche Schlüssel. Weil es sich bei der E-Multiplikation um eine Einwegfunktion handelt, ist die bisher beste Möglichkeit, WDSA zu knacken, ein Brute-Force-Angriff entweder direkt auf den privaten Schlüssel oder indirekt über die Cloaking-Elemente.[31, p. 9] Sowohl die Schlüssellänge l als auch der Parameter für die reinen Braids der Cloaking-Elemente L werden anhand des erwünschten Sicherheitslevels berechnet. Auf dem ESP8266 werden erfolgreich Signaturen mit einem 256-Bit Sicherheitslevel generiert.

10.4.4 Benötigter Speicherplatz öffentlicher Schlüssel

Der für die Matrix des öffentlichen Schlüssels benötigte Speicher ist nur abhängig vom Grad des Braids N und vom Exponenten der Größe des Binärkörpers m. Er lässt sich berechnen durch die Formel $Anzahl\ der\ Bits = N^2 * m$. Bei $N = 8$ und $m = log_2(q) = log_2(256) = 8$ sind es konstant 512 Bits bzw. 64 Bytes. Zusammen mit weiteren 8 Bytes für die Permutation und 8 Bytes für die T-Werte benötigt ein öffentlicher Schlüssel insgesamt 80 Bytes. Es ließe sich noch Speicher sparen, indem man die letzte Zeile der Matrix, die immer abgesehen von der 1 am Ende nur aus 0en besteht, nicht speichert [31, p. 20] und für die Permutation nur $N * log_2(N) = 24$ Bits $= 3$ Bytes verwendet. Das ergibt insgesamt einen benötigten Speicherplatz von $56 + 3 + 8 = 67$ Bytes.

10.4.5 Benötigter Speicherplatz und Laufzeitverhalten der Signaturgenerierung und Verifizierung

Da ein Artin-Generator mit einem konstanten $N = 8$ jeweils 4 Bits benötigt, werden zwei Generatoren in 1 Byte gespeichert. Der für eine Signatur der Länge k benötigte Speicherplatz in Bytes beträgt also $\lceil k/2 \rceil$ Bytes (+ 2 weitere Bytes, wenn man k mit angeben will). Es wurden für verschiedene Sicherheitslevel und wechselnden Umformungsgrad (Rohsignatur, nur BKL, BKL + Dehornoy) jeweils 100 Signaturen mit zufälligen Hashwert, Kodierungsgeneratoren, T-Werten, privaten Schlüsseln, a und b generiert und die minimale, maximale und durchschnittliche Länge, sowie die durchschnittlichen Laufzeiten für die Berechnung des öffentlichen Schlüssels, für die Signierung sowie für die Verifizierung auf dem ESP8266 ausgewertet. Hierbei muss beachtet werden, dass es sich bei den Rohsignaturdaten nur um Vergleichswerte handelt. Eine Verwendung der nicht umgeformten Rohsignatur gilt nicht als sicher und sollte daher auch nicht praktiziert werden.

SL	l	L	BKL	Deh	min. k	max. k	∅ k	T(ö. S.)	T(Gen)	T(Ver)
-			nein	nein	410	628	522		1.789 μs	248 ms
32	19	2	ja	nein	946	2342	1713	7,9 ms	18.423 ms	579 ms
32			ja	ja	454	916	686		18.934 ms	291 ms
-			nein	nein	486	674	587		1.865 μs	266 ms
64	44	4	ja	nein	1090	2722	1897	14,3 ms	23.187 ms	620 ms
64			ja	ja	592	1020	747		27.301 ms	316 ms
-			nein	nein	642	834	734		2.046 μs	305 ms
128	101	8	ja	nein	1652	3254	2352	29,3 ms	35.925 ms	756 ms
128			ja	ja	706	1260	921		50.312 ms	366 ms
-			nein	nein	942	1132	1027		2.420 μs	381 ms
256	222	15	ja	nein	2048	4398	3269	60,9 ms	70.997 ms	1002 ms
256			ja	ja	1088	1672	1370		150.187 ms	489 ms

Beim Vergleich der Braidlängen erkennt man einen deutlichen Unterschied im Wachstum zwischen der ungekürzten BKL-Form und den anderen beiden Varianten. Auch wenn der vollständig umgeformte Braid im Vergleich dazu nicht viel länger als die Rohsignatur ist, so benötigt die Umformung für die BKL-Form vor allem bei einem höheren erwünschten Sicher-

heitslevel zeitweise viel Speicherplatz. Eine Umformung wie durch den BKL-Algorithmus ist aber notwendig, um die Positionen der Teilbraids zu verschleiern. Die Dehornoy-Umformung allein würde dafür nicht ausreichen, da sie den Braid nur reduziert und der Großteil des privaten Schlüssels am Ende in Klartext vorliegen würde. Ohne Dehornoy wiederum würde die Signatur aber unnötig lang werden. Letztendlich ist aktuell keine bessere Lösung als eine kombinierte Anwendung beider Algorithmen öffentlich bekannt, um sichere Signaturen mit akzeptabler Länge zu erhalten. Selbst die Autoren des Haupt-Papers zu WalnutDSA scheinen noch keine bessere Methode zu kennen, da sie abgesehen von BKL und Dehornoy lediglich das Anwenden *„einen der vielen bekannten Umformungsalgorithmen zur Verschleierung“* empfehlen.[31, p. 18]. Ganz anders als bei der Generierung wächst die Laufzeit der Verifizierung linear mit der Braidlänge, weswegen auch Signaturen höherer Sicherheitslevel schnell verifiziert werden.

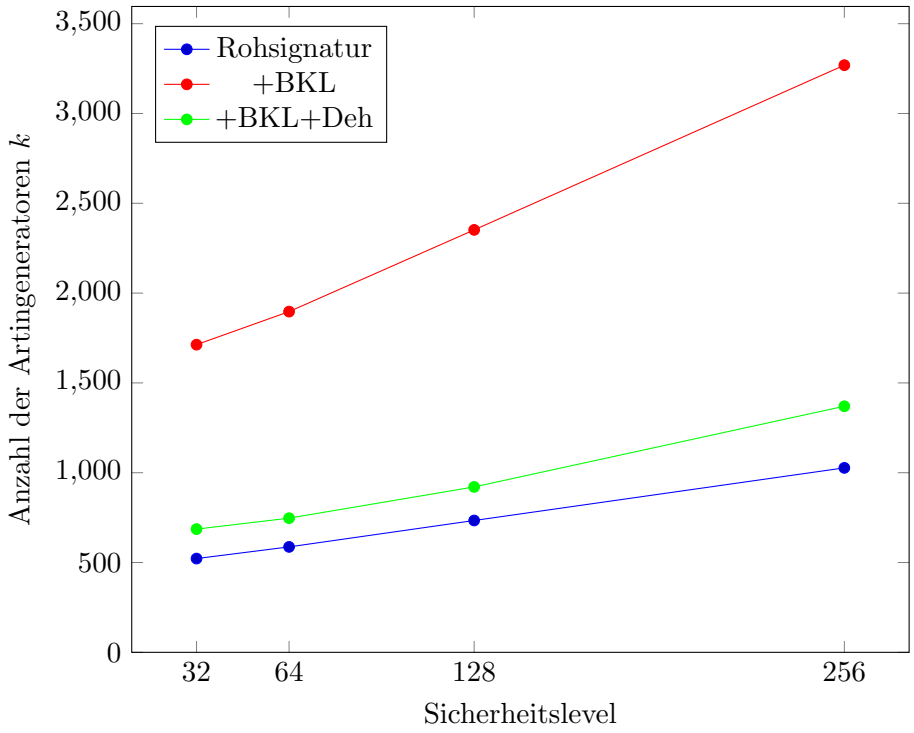


Abbildung 10.4: Durchschnittliche Länge einer Signatur

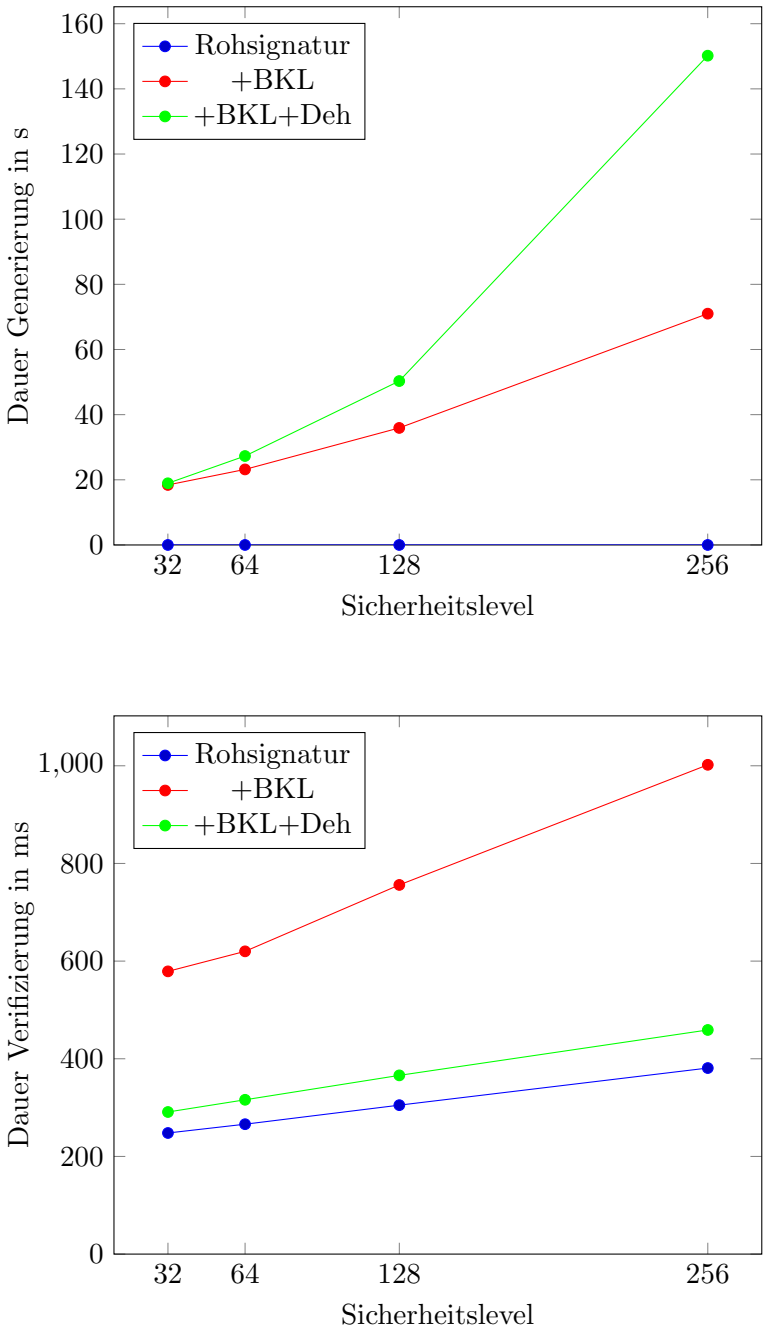


Abbildung 10.5: Durchschnittliche Laufzeiten auf dem ESP8266

11 Implementierung Riot + Arduino

In diesem Kapitel geht es um Implementierung von RIoT mit Arduino M0 Pro als Testumgebung. Die vollständige Implementierung ist bei `https://gitlab.cip.ifi.lmu.de/kimdo/Walnut-DSA-RIOT` befindlich.

11.1 RIOT

RIOT ist ein Betriebsystem, das für die Geräte mit eingeschränkten Ressuourcen entwickelt wurde. Dabei kann es schon mit 1,5kB RAM und 5kB ROM laufen. Die standardmäßige C und C++ Codes sind in der Regel(abgesehen von physische Einschränkungen von Zielplattformen) ohne große Aufwand verwendbar. Die eher niedrige Plattformabhänigkeit ist auch einen Vorteil von RIOT OS. Zudem gibt es ein *Native* Port, bei dem man RIOT auf einem Linux oder auf einem Mac als ein Prozess laufen lässt. Auf Native Port steht mehre Tools wie z.B. Valgrind, gprof, die zum Debuggen oder Analyiseren von Code sehr hilfreich sind.[60]

11.2 Arduino M0 Pro

Arduino M0 Pro ist ein Board, der von RIOT OS unterstützt wird und auf dem die Implementierung getestet wurde. Das hat 256kB Flash und 32kB SRAM zur Verfügung. Das Board hat den Mikrocontroller ATSAMD21G18, der sich auf 32-Bit ARM Cortex-M0 Kern basiert. Dieser Board wird von RIOT OS unterstützt. Flashen erfolgt über OpenOCD.

11.3 Darstellung von Daten

11.3.1 Braids

Für die Breite der Braids wurde 8 in der Implementierung genommen. Die Artin-Generatoren werden also mit 4 Bits dargestellt, da es 14 Generatoren $\sigma_1, ..., \sigma_7, \sigma_{-1}, ..., \sigma_{-7}$ gibt. Dabei wurde das erste Bit als Vorzeichen verwendet. Für ein Permutation Table nutzt die Implementierung jeweils 32 Bits. Man könnte pro Eintrag in Permutation Table 3 Bits nutzen mit 0 bis 7, aber in der Implementierung ist es entschieden 4 Bits zu nutzen. Es ist ziemlich nützlich z.B. für die Konvertierung zwischen Permutation Table und Descending Cycle Decomposition Table das neunte Element neben die Zahlen zu haben, um „unbesetzte“ Zellen zu makieren. Außerdem kann man dadurch den Datentyp _textttuint32_t verwenden, der ohne zusätzlich Definition einsetzbar ist.
Um ein Braid in Artin-Generatoren darzustellen, verwende ich ein Array von `uint8_t` und für das Permutation Table `uint32_t`. Die Lese- und Schreiboperationen erfolgen wie im folgdenen Macro über $(1111)_2$ als Bitmaske.

```
struct CB {
#define GET_4_BITS(data, index) (((data) >> ((index) << 2)) & 15)
#define SET_4_BITS(data, index, value) ((data) = ((data) & ~(15 << ((index) <<
    2))) | ((value) << ((index) << 2)))
```

`data` soll ein Datentyp(`uint8_t` bei Artin-Generatoren und `uint32_t` bei Permutation Table) sein und `index` entspricht das Index von gewüschte Daten in dem `data`. Hier muss man aufpassen, dass die erste 4 Bits von `uint8_t` als höhere Index gezählt wird.

11.3.2 Colored Burau Representation

```
struct colored_burau_representation {
  uint8_t matrix[BRAID_SIZE * BRAID_SIZE];
  uint32_t permutation;
};
```

Für den Matrix-Teil wurde 64 Bytes lange eindimensionale Array verwendet, die ohne sonstige Allokationen im Gegensatz zu zweidimensionale Array benötigen. Hier ist es sinnvoller ein volles Byte für jedes Element zu vergeben, da es höchstens zwei Matrizen(Signaturgenerierung braucht ein Matrix und Signaturverifizierung braucht zwei Matrizen) alloziert wird, weswegen man auch nicht so viel Speicher daduch sparen kann.

11.4 Sicherheitslevel und Konfigurationen

Die Implementierung orientiert sich am Sicherheitslevel von 128 Bits. Daraus ergibt die Konfiguration mit L = 15(minimale Länge von der Produkt von Pure Braid Generatoren in Cloaking Element) und l = 132(Länge vom privaten Schlüssel). Diese Werte sind höcher als theoretische Minimum gesetzt. Die kodierte Nachricht belegen großer Anteil von Signatur. Da es dabei potenzielle Angriffe geben können, wurde L und l als Sicherheitsmaßnahme dagegen erhöht von dem theoretisch bekannten Minimum[5, S. 17].

11.5 Generierung vom privaten Schlüssel

Ein private Schlüssel wird mit Hilfe einem Shell-Skript beim Kompilieren von Code generiert und bleibt in der Binärdatei. Dadurch bleibt der privaten Schlüssel unverändert auch wenn der Microcontroller ausgeht. Außerdem gibt es Funktionen, mit der man neue Schlüsselpaar nach Bedarf generieren kann.

11.6 Cloaking Element

Am Anfang wird zu erst $\sigma_{-1} * \sigma_1$ nicht frei reduziert geschrieben. Das ist dafür da, um das Element von andere Elemente zu trennen. Sonst kann ω schon bei Generierung mit vorherigem Element vermischen, das Schwieigkeit bei Berechnung ω^{-1} veruracht. Dise beide Generatoren werden bei Umformung aller erstens gelöscht.
Das erste Teil von ω, die in Section 8.4 erwähnt wird, wurde in der Mehtode `fix_permutation()` mit Hilfe der Band-Generatoren realisiert. Die Band-Generatoren ist deshalb vorteilhaft, da

sie nur die Permutation von gewählte zwei Strähne vertauscht. Die Abbildung 11.1 zeigt den Ablauf.

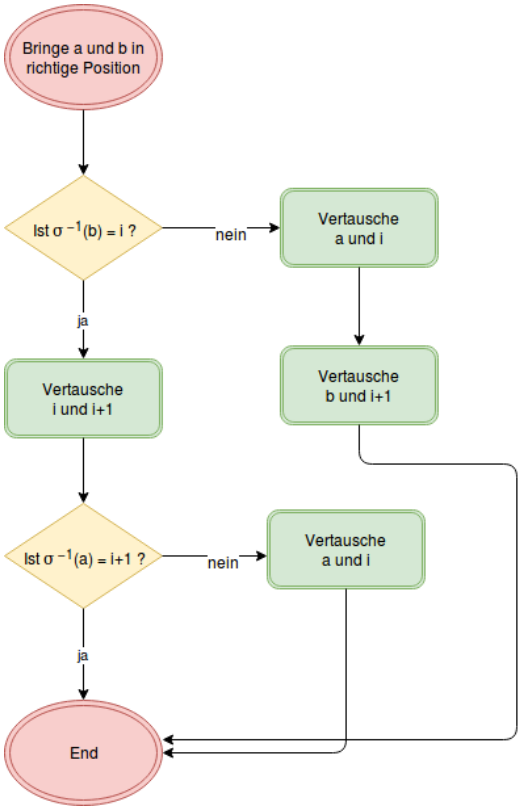


Abbildung 11.1: Ablauf von `fix_permutation()`

11.7 E-Multiplikation

Hier wurde die Matrixmultiplikation ersetzt. Zum betrachten ist eine E-Multiplication von σ_3 und σ_{-3} als ein Beispiel, wobei die Breite von Braid 4 ist:

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \star \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & t_3 & -t_3 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a & b+t_3*c & -t_3*c & d+c \\ e & f+t_3*g & -t_3*g & h+g \\ i & j+t_3*k & -t_3*k & l+k \\ m & n+t_3*o & -t_3*o & p+o \end{pmatrix}$$
$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \star \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & -\frac{1}{t_4} & \frac{1}{t_4} \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a & b+c & -\frac{1}{t_4}*c & d+\frac{1}{t_4}*c \\ e & f+g & -\frac{1}{t_4}*g & h+\frac{1}{t_4}*g \\ i & j+k & -\frac{1}{t_4}*k & l+\frac{1}{t_4}*k \\ m & n+o & -\frac{1}{t_4}*o & p+\frac{1}{t_4}*o \end{pmatrix} \tag{11.1}$$

Hier sehen wir, dass eigentlich nur die 3 Spalten mit Index 3-1, 3, 3+1 geändert wird. Und diese gilt auch für beliebige Braidbreite n und Artin-Generator i. D.h., statt eine Matrix-multiplikation kann ich über alle Zeile iterieren und nur die betroffene 3 Zellen(2, wenn i = 1) pro Iterierung rechnen. Dabei muss man auf die Reihenfolge von Berechnung achten,

da i-1te und i+1te Spalten abhänig von ite Spalte ist. Nun kann man das Ergebnis an der linken Matrix gleich speichern, und die rechte Matrix muss gar nicht erstellt werden, da nur t_i nötig ist.

11.8 Berechnung von BKL-Normalform

Die Implementierung nutzt BKL-Linksnormalform, da sie durchschnittlich kürzere Braid ergibt als der Rechtsnormalform. Von 1000 zufällig generierte Signatur gibt der Rechtsnormalform als durchscnittliche Anzahl von Artin-Generatoren 3278,52 aus und der Linksnormalform 1851,41. Dass die Länge von Braid nach diese Methode ist vor allem deshalb interessant, da die Dauer vom Henkelreduktionsalgorithmus sehr stark davon beeinträchtigt wird.

11.8.1 Konvertierung von Artin-Generatoren zu Permutation Tables

Die Permutation Tables benötigen 8 Mal zu viel Speicher im Vergleich zu den Artin-Generatoren, falls sie 1:1 konvertiert werden(Maximale Anzahl von den Artin-Generatoren vor der Umformung ist in dieser Implementierung 2850. D.h. 11400 Bytes soll reserviert werden. Siehe 11.10.1). Um das zu vermeiden, versucht die Implementierung mehre Artin-Generatoren in einem Permutation Table zu schreiben. Es gibt viele unterschiedliche Folge von Artin-Generatoren, die mit einem Permutation Table darstellbar wären, aber betracht wird eher einfache aber in unserer Signatur sehr häufig auftretende Folge, die im Kapitel Cloaking Element und Message Encoding erwähnte als das Pure Braid Generator vorkommen. Das sind die Folgen mit absteigende Indizies. Eine Folge mit absteigende Indizies, wobei alle Indizes gleiche Vorzeichen haben, bilden eine oder mehre absteigende Zyklen. Hier ist ein Beispiele mit Braid Breite von 8:

$$\sigma_7\sigma_6\sigma_5\sigma_3\sigma_2 \Leftrightarrow \begin{pmatrix} 1 & 4 & 2 & 3 & 8 & 5 & 6 & 7 \end{pmatrix} \tag{11.2}$$

In dem Beispiel bildet $\sigma_7\sigma_6\sigma_5$ den Zyklus (8 7 6 5) und $\sigma_3\sigma_2$ den (4 3 2). Da $\sigma_7\sigma_6\sigma_5$ zusammen einen absteigenden Zyklus bildet. Der Zyklus (4 3 2) darf auch in dem selben Permutation Table geschrieben werden, da diese beide Zyklen parallel sind. Für die Folge mit negative braucht man zusätzliche Rechenschritte:

$$\begin{aligned} &\sigma_2^{-1}\sigma_3^{-1}\sigma_5^{-1}\sigma_6^{-1}\sigma_7^{-1} \Leftrightarrow \\ &D^{-1} * (D * (\sigma_2^{-1}\sigma_3^{-1}\sigma_5^{-1}\sigma_6^{-1}\sigma_7^{-1}) \Leftrightarrow \\ &D^{-1} * \begin{pmatrix} 8 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{pmatrix} * \begin{pmatrix} 1 & 3 & 4 & 2 & 6 & 7 & 8 & 5 \end{pmatrix} \Leftrightarrow \\ &D^{-1} * \begin{pmatrix} 8 & 2 & 3 & 1 & 5 & 6 & 7 & 4 \end{pmatrix} \end{aligned} \tag{11.3}$$

Hier wird das Permutation Table von der Folge zu erst berechnet. Und genauso wie es ein negativen Artin-Generator wäre, multipliziert man das Permutation Table mit D.
Nun betrachten wird den Pure Braid Generator in 8.7 genauer. Nach oben genannten Regel bekommt man drei Permutation Tables mit $b_{r-1}...b_l$, b_l(zwei b_l können nicht in einem Table sein, da sie weder zusammen ein Zyklus baut noch untereinander parallel sind) und $b_{l+1}^{-1}...b_{r-1}^{-1}$. D.h., man bekommt maximal drei Permutation Table pro Pure Braid Generator(zwei, falls l+1 = r). Als Zusatzeffekt wird wengier Iteration bei der Berechnung von

BKL-Normalform benötigt, da es offensichtlich kleinere Anzahl von Permutation Table geben wird.
Es gibt noch Möglichkeiten(wie z.B. $\sigma_4\sigma_6\sigma_3$, da σ_6 parallel zu $\sigma_4\sigma_3$ und das ganze äquivalent zu $\sigma_6\sigma_4\sigma_3$ ist) noch extremer Artin-Generatoren zusammenzufassen, aber diese ist nicht so sinnvoll. Sie kommen im Vergleich zu dem oben beschriebenen Fall eher selten vor(vor allem fast alle Teil von Signatur außer Priv(S) und $Priv(S)^{-1}$ besteht aus Pure Braid Generator), obwohl sie viel mehr Fallunterscheidungen benötigen, die am Ende mehr Rechnleistung verbrauchen wird.

11.9 Dehornoy's algorithm

Von den zwei Variante von den Henkelreduktionsalgorithmen, die im Kapitel 8.7.3 erwähnt sind, wurde der Greedy-Algorithmus verwendet, um zu lange Rechnaufwand zu vermeiden. Von 1000 zufällig generierte Signatur hat die volle Reduktion durchschnittlich 274 kürzeres Braid ergeben. Der Einsatz von der volle Reduktion kann interessanter sein, falls es wichtiger ist, dass Verfizierung kürzer dauert, als die Generierung von Signatur.
Der Henkelreduktionsalgorithmus von Dehornoy wird wegen stark eingeschränkte Speicher von den Zielplattformen von RIOT OS auf in Array 4 bitweise gespeicherte Braids angewendet. Das größete Problem dabei ist, dass in einer Henkelreduktion das erste und letzte Element gelöscht werden und einige Artin-Generatoren mit drei andere Artin-Generatoren ersetzt werden. D.h., im Worst Case müssen alle Array-Einträge wegen den Verschiebungen kopieriert werden. Um die Verschiebungen in Array zu minimieren, hat Reduktion von einem reduzierbaren Henkel(siehe Kapitel **??**) folgende Ablauf:

1. Das erste und letzte Elemente von einem Henkel wird in der Reduktion zu erst mit 0 überschieben(Diese gilt auch bei den freien Reduktionen, die auch immer wieder ausgefürht wird).
2. Alle Einträge im Henkel wird nach linkes verschoben, sodass alle 0 am Ende von Henkel stehen. Dabei werden `zeros_counter` als Anzahl von 0 im Henkel und `to_insert` als zu ersetzende Artin-Generatoren gezählt. Nun gibt es zwei Fälle.
3 1. In diesem gilt `zeros_counter` $\geqslant$ `to_insert`/2. D.h., es gibt genug 0, sodass man alle zu ersetzende Artin-Generatoren ersetzen kann, ohne dabei mehr Platz zu nutzen. Alle Elemente von Henkel wird von Ende des Henkels neu geschrieben. Dabei werden Artin-Generatoren transformiert, falls es nötig ist. Die neu geschriebene Elemente werden mit 0 „gelöscht“ Da alle Elemente schon nach links verschoben sind, Überschreibung von einem Element vor der Neuschreibung ist nicht möglich.
3 2. In dem Fall gibt es nicht genügende leere Plätze vorhanden. Hier wird alle Elemente hinterdem Henkel mit `to_insert` * 2 nach rechts verschoben. Das Ende von Henkel wird aktualisiert und wie bei 3-1 vom Ende des Henkels geschrieben.
Die Anzahl von 0 werden nach der Reduktion betrachtet und die 0 werden gelöscht, wenn sie mehr als `ZEROS_LIMIT` vorhanden ist. Falls die 0 bis zum Ende des Algorithmus nicht gelöscht wird, wird der Länge von Braid inklusiv 0 temporär zu hohe, sodass der Speicher nicht mehr ausreicht. Auch der leere Ablauf wegen 0 bei Suche von Henkel oder sonstige Operationen wird auch in dem Fall groß, weswegen wieder Laufzeit länger werden kann.

11.10 Maximale Länge von Braid

Hier wird die maximal mögliche Länge von Signatur betrachtet.

11.10.1 Maximale Länge von Braid vor der Umformung

Im Kapitel 8.5 sehen wir, dass eine Signatur 3 Cloaking Elements, ein inverse Cloaking Element, gecodete Nachricht, einen privaten Schlüssel und die Inverse davon hat.

Gecodete Nachricht

Jede 4 Bits von den gehashten Nachricht bestimmt, welcher Pure Braid Generator wie oft vorkommen soll[5, S. 7]. Es wird dann 64 Mal von dieser 4 Bits geben, da der Hashwert 256 Bits groß sein soll. Ein Merkmal dabei ist, dass von einem Pure Braid Generator $g_{a,b}$ b immer mit 8 fixiert ist. Im schlimmsten Fall wird so aussehen:

$$\begin{aligned}(g_{7,8}^4 * g_{1,8}^4)^{\frac{64}{2}} = (\sigma_7^2)^4 * \sigma_7\sigma_6 \dots (\sigma_1\sigma_1)^4\sigma_2^{-1}\sigma_3^{-1} \dots \sigma_6^{-1} * (\sigma_7^{-1} * \sigma_7) * \sigma_7^7 * \\ \dots * \sigma_7\sigma_6 \dots (\sigma_1\sigma_1)^4\sigma_2^{-1}\sigma_3^{-1} \dots \sigma_6^{-1} * \sigma_7^{-1}\end{aligned} \tag{11.4}$$

Hier ist das Vorkommen von jedem Pure Braid Generator immer 4 und $g_{7,8}^4, g_{1,8}$ werden abwechselend gewählt. Nur $(\sigma_7^{-1} * \sigma_7)$ wird frei reduziert. Die Länge von Braid ist also $(20-1)+(8-1)) * 32 + 2 = 834$. +2 kommt davon, da die freie Reduktion von $(\sigma_7^{-1} * \sigma_7)$ beim ersten $g_{7,8}^4$ und beim letzten $g_{1,8}^4$ nicht vorkommen.

Cloaking Element

ω in Cloaking Element enthält mindestens eine L(hier 15) lange Produkt von Pure Braid Generatoren und ein Braid, das gewünschte Permutation für ω herstellt.
Beim ersten Teil wird im Worst Case 14 langes Braid zu erst geschrieben. Um die MindestLänge 15 zu erreichen, muss nun noch ein Pure Braid Generator hinzugefügt werden. Dieser ist der längste Pure Braid Generator $g_{1,8}$ mit 14 Artin-Generatoren, wobei N = 8 ist. Dabei kommt noch ein Zusatzbedingung, dass der Pure Braid Generator vor $g_{1,8}$ nicht mit σ_7^{-1} beendet, da sonst freie Reduktion das Braid doch etwas verkürzen wird. Also dieser Teil wird im Worst Case 28 Artin-Generatoren haben.
Das längste Braid für den zweiten Teil besteht aus den Band-Generatoren $a_{6,1}$ und $a_{7,2}$ oder $a_{6,2}$ und $a_{7,1}$ mit 22 Artin-Generatoren.
Da ein Cloaking Element im Form von $(\sigma_{-1} * \sigma_1 * \omega * b_i^2 * \omega^{-1}$ ist die Länge von Cloaking Element im schlimmsten Fall $2 + (28 + 22) * 2 + 2 = 104$.

Der private Schlüssel

Die Länge von den privaten Schlüssel ist 132 und hier findet keine freie Reduktion statt. Dasselbe gilt für die Inverse von den Schlüssel.

Gesamt Länge von Braid vor Umformung

Die Länge von dcer Signatur vor der Umformung $4 * 104 + 2 * 132 + 834 = 1514$.

11.10.2 Länge von Braid nach der Umformung

Die Länge von Braid nach der Berechnung vom BKL-Normalform und dem Henkelreduktionsalgorithmus anhand von einem bestimmmten Signatur vor der Umformung genau vorherzusagen, ist sehr schwer. Als Alternativ gibt es folgende Statistik, die von 1000 zufällig generierte Signatur gewonnen wurde, wobei die Konfiguration gleich wie in 11.4 erwähnt ist.
Durchschnittliche Länge von 1000 Braids nach der Berechnung von BKL-Normalform: 1851,41
Davon am kürzesten: 2510
Davon am längsten: 1128
Durchschnittliche Länge von 1000 Braids nach der Berechnung von BKL-Normalform und Henkelreduktionsalgorithmus: 1120,38
Davon am kürzesten: 1558
Davon am längsten: 730

11.10.3 Maximale Anzahl von Permutation Table

Mit der Methode, die in 11.8.1 dargestellt, brauche ich nicht so viel Permutation Tables wie die Anzahl von Artin-Generatoren reservieren. Auch hier ist es schwer möglich, die Anzahl von nötige Permutation Table ist schwer vorherzusagen.

Gecodete Nachricht

Hier gibt es zwar sehr viel verschiedene Möglichkeiten, die Worst Case entsprechen, aber das oben genannten Fall entspricht auch hier den schlimmsten Fall.

$$\begin{aligned}&\underbrace{(\sigma_7^2)^4}_{jew.1PT} * \underbrace{\sigma_7\sigma_6\dots\sigma_2\sigma_1}_{1PT} \underbrace{\sigma_1^7}_{jew.1PT} \underbrace{\sigma_2^{-1}\sigma_3^{-1}\dots\sigma_6^{-1}}_{1PT} *(\sigma_7^{-1} * \sigma_7) * \underbrace{\sigma_7^7}_{jew.1PT} \\ &* \cdots * \underbrace{\sigma_7\sigma_6\dots\sigma_1}_{1PT} \underbrace{\sigma_1^7}_{jew.1PT} \underbrace{\sigma_2^{-1}\sigma_3^{-1}\dots\sigma_6^{-1} * \sigma_7^{-1}}_{1PT}\end{aligned} \tag{11.5}$$

Das resultiert $7 + (1 + 7 + 1)) * 32 + 1 = 513$ Permutation Tables. Hier kommt $+1$ statt $+2$, da der letzte σ_7^{-1}, der nicht frei reduziert, in diesem Fall mit $\sigma_2^{-1}\sigma_3^{-1}\dots\sigma_6^{-1}$ in einem Table geschrieben wird.

Cloaking Element

Der erste Teil von ω besteht aus Pure Braid Generatoren, die zusammen mindest 15 Artin-Generatoren haben. Der schlimmste Fall ist, wo man keinen Artin-Generator bei erste 14 Artin-Generatoren *zusammenfassen* kann. Folgender Fall wird am meisten Permutation Table benötigen.

$$g_{a,a+1}^7 g_{x,y}, \text{ wobei } a - 1 \neq x \text{ und } x + 1 < y \tag{11.6}$$

Die erste 14 Artin-Generatoren wird mit den Pure Braid Generatoren mit 2 Artin-Generatoren aufgefüllt, die jeweils 2 Permutation Table nutzen(a muss nicht immer gleich sein, aber absteigende Folge wie z.B. $g_{a,a+1} * g_{a-1,a}$ darf für Worst Case nicht vorkomen). Der letzte Pure Braid Generator soll 3 Tables benötigen, der den Überlauf über 15 ausnutzt.
Der zweite Teil von ω besteht aus 2 Band-Generatoren, die jeweils höchstens 2 Permutation Table(1 Table für den positiven Teil und 1 Table für den negativen Teil) brauchen. D.h., ein Cloaking Element braucht höchstens $(17 + 2 * 2) * 2 + 2 = 44$ Permutation Table.

Der private Schlüssel

Im Worst Case benötigt jeder Artin-Generator ein neues Permutation Table. Es braucht also 132 Permutation Tables.

Gesamtanzahl von Permutation Table

Es wird im Worst Case $l(\mathrm{E}(\mathrm{H}(\mathrm{m})_w orst + (v_{1worst} + v_{1worst}^{-1} + v_{worst} + v_{2worst}) + ((\mathrm{Priv}(\mathrm{S})_{worst} + Priv(S)_{worst}^{-1}) = 513 + (44 * 4) + (132 * 2) = 953$ Permutation Table benötigt.

11.11 Speicherverbrauch

Da die Anzahl von nötige Artin-Generatoren und Permutation Table nicht vorhersagbar ist, werden nötige Speicher für die beide Daten nach Schätzung maximal möglichen Bedarf reserviert.

Speicherverbrauch von Signatur: 3500 Artin-Generatoren(4 Bits pro Generator) $\rightarrow$ 1750 Bytes
Die Stelle wird zum Speichern von Signatur vor und nach Umformung verwendet. Da hier gemeinsame Stelle wiederverwendet wird, wurde die Größe nach höcheren Bedarf angespasst. Der Zeitpunkt, an der Signatur möglichst lang wird, ist während der Berechnung vom Henkelreduktionsalgorithmus. Obwohl er die Signatur eventuell verkürzen wird, als Zwischenergebnis wird die Signatur wegen die Substitutionen von σ_i^e Generatoren zu $\sigma_i^f \sigma i - 1^e \sigma i^{-f}$ temporär länger. Da auch die maximale Länge von Braid in BKL-Normalform auch unbekannt ist, wurde 3500, die um 1600 höher als der Durchschnitt ist, als maximal mögliche Länge von Braid genommen.

Speicherverbrauch von Permutation Tables: 953 Permutation Tables(4 Bytes pro Table) $\rightarrow$ 3812 Bytes
Die maximale Menge von Permutation Table ist wie im 11.10.3 erwähnt.

Cloaking-Element v_1: Da die Signatur in einem Array geschrieben wird, aber v_1^{-1} vor v_1 eingefügt wird, muss man sie zu erst an einer anderen Stelle speichern. Dabei wird 52 Bytes für 104 Artin-Generatoren reserviert.

Private Schlüssel: Der private Schlüssel verbraucht 64 Bytes als Konstante.

Öffentliche Informationen: Hier gehören die T-Values und der öffentliche Schlüssel. T-Values braucht 1 Bytes pro ein Element und dabei ist es 8 Bytes. Der öffentlichen Schlüssel hat ein 8*8 Matrix mit 1 Byte große Elemente und ein Permutation Table. Also ein öffentlichen Schlüssel verbraucht 68 Bytes.

11.11.1 Speicheranforderung bei Signaturgenerierung

Hier kommt alle oben genannte Elemente. Es wird also $1750 + 3812 + 52 + 64 + 68 = 5746$ Bytes benötigt(Die lokale Variablen und Argumentübergaben wurden nicht mit gezählt).

11.11.2 Speicheranforderung bei Signaturverfizierung

Für die Verifizierung kommt schon über der Henkelreduktionsalogorithmus reduzierte Signatur, die unwahrscheinlich über 3500 wird. Es muss auch kein Platz für Permutation Tables reserviert werden. Statt dessen wird hier 2 zusätzliche Matrizen und ein Permutation Table für Matrixmultiplikation und Pub(E(H(m)) und ein anderes Container für E(H(m)) verlangt. Also $(2200 * 4/8)$(Signatur, auch eine Schätzungswert wie bei der letzte Abteilung) $+ 834 * 4/8$(E(H(M))) $+ (8 + 68)$(öffentliche Informationen) $+ 64 * 2$(2 Matrizen für eine Matirxmultiplikation) $+ 4$(Permutation Table) $= 1725$Bytes wird bei der Verifizierung abgesehen von lokale Variablen verbraucht.

11.12 Eingesetzte RIOT Module

Für die Implementierung wurde drei Module verwendet. Die Implementierung ist abhänig von Modul crypto und hashes.

11.12.1 crypto

Dieses Modul enthält enthält chacha, die für sichere Zufallszahlgenerierung nötig ist.

11.12.2 hashes

Zur Berechnung von Hashwert der Nachricht wurde sha256 aus diesem Modul verwendet. Davon erhält man 256 Bit lange Hashwert.

11.12.3 xtimer

Dieses Modul wurde für Zeitmessung beim Test verwendet. Das implementierte Modul ist unabhänig von diesem Modul, da dieses Modul nur im Testprogramm aufgeruft wird.

11.13 Test vom Programm

Zur Anwendung von Testtools wie Valgrind wurde das Native Port von RIOT verwendet und die Laufzeitmessung wurde auf Arduino M0 Pro durchgeführt. Der Test lief für das Sicherheitslevel 128 in 11.4 genannten Konfiguration und die minimale Bedinung. Für Sicherheitslevel 112 und 256 wurde das theoretische minimale L und l verwendet.

11.13.1 Überprüfung durch Testtools

Beim Test über Valgrind und Address sanitizer wurde kein Fehler aufgezeigt.

11.13.2 Laufzeit

Hier wird die Laufzeit von Algorithmus im Betracht gezogen. Es wurde 100 Signaturen mit zufällige Nachricht, Schlüssl und Parametern generiert und ein Durchschnitt davon wurde berechnet.

Schlüsselgenerierung

Hier wurde ausnahmsweise 1000 Schlüsselpaare generiert, da hier eher kürzere Laufzeit nötig ist.
Sicherheitslevel 112: 3,8 ms
Sicherheitslevel 128(l = 101, L = 8): 4,5 ms
Sicherheitslevel 128(l = 132, L = 15: 6,1 ms
Sicherheitslevel 256: 10,5ms
Mit höhere Sicherheitslevel wächst die Anzahl von E-Multiplikation linear.

Signaturgenerierung

Sicherheitslevel 112: 15,4 s
Sicherheitslevel 128(l = 101, L = 8): 17,3 s
Sicherheitslevel 128(l = 132, L = 15: 30,5 s
Sicherheitslevel 256: 47,7ms
Hier sieht man, dass die Laufzeit von Signaturgenerierung stark mit Sicherheitslevel steigt.

Verifizierung

Sicherheitslevel 112: 59,7 ms
Sicherheitslevel 128(l = 101, L = 8): 59,6 ms
Sicherheitslevel 128(l = 132, L = 15: 67,7 ms
Sicherheitslevel 256: 87,1 ms
Die Verifizierung von Signatur dauert auch bei Sicherheitslevel von 256 unter 100 ms. Bei Sicherheitslevel 128 mit l=101, L=8 lief die Verfizierung sogar 0,1 ms schneller. Das besagt einerseits, dass das Wachstum von Laufzeit nicht stark ist(Z.B. bei der Schlüsselgenerierung ist die Dauer näher zu 1:1. Hier wurde die Laufzeit bei Sicherheitlevel 256 weniger als die dopplte Laufzeit von 128 mit l=101, L=8), anderersseit sagt es, dass die Varianz von Laufzeit groß ist.
Die Schlüsselgenerierung und Die Berechnung von Hashwert ist oben nicht enthalten. sha256 aus dem Modul hashes benötigt 624 microsekunden für von 0 bis 200 lange Strings.

Umformung und Laufzeit

Noch intersante Information ist wie lang die Signaturbildung ohne Umformung dauert und die Verifizierungszeit davon beeinfloßen wird. Die Daten dafür wurde von 11.4 erwähnte Konfiguration gemessen.
Generierung von Signatur:
ohne Umformung: 9,7 ms
mit BKL-Normalform, ohne Dehornoy: 6,1 s
mit BKL-Normalform, mit Dehornoy: 30,5 s

Verifizierung von Signatur:
ohne Umformung: 57,7 ms
mit BKL-Normalform, ohne Dehornoy: 105,6 ms
mit BKL-Normalform, mit Dehornoy: 67,7 ms

Das Testergebnis zeigt, dass die Umformung um 3000 fach mehr Berechnung als die Berechnung von WalnutDSA Elemente benötigt. Und davon benötigt der Dehornoysalgorithmus der größte Teil von Berechnungszeit. Bei der Verifizierung demgegenüber, dass der Dehornoysalgorithmus die Verifizierungszeit stark reduziert.

11.13.3 Durchschnittliche Braidlänge

Die Länge von Signatur wurde in 11.4 genannte Konfiguration gemessen. Es wurde links-BKL-Normalform und der Greedy-Ansatz von Dehornoysalgorithmus verwendet.
Vor Umformung: 1040
nach BKL-Normalform: 1852
nach beide Umformungsfunktion: 1120
Die Signaturlänge steigt stark durch BKL-Normalform. Nun kann man hier erkennen, dass der implementierte Dehornoysalgorithmus die Signatur wie gewünscht verkürzt.

12 Ergebnisse der Evaluierung und weiterführende Diskussion

Nachdem die WalnutDSA-Bibliothek auf dem eigenen System auf Korrektheit, benötigter Speicherplatz und Laufzeitverhalten geprüft wurde, werden die Ergebnisse in diesem Kapitel mit Vergleichswerten aus Tests mit anderen Implementierungen gegenübergestellt und in einem weiteren Test die Kompatibilität der Systeme untereinander getestet. Anschließend wird diskutiert, warum es sich bei WalnutDSA um ein sicheres kryptographisches Verfahren handelt und wie sich die Implementierungen aktuell umsetzen ließen.

12.1 Plattformübergreifende Evaluierung

WalnutDSA wurde für den Linux Kernel auf einem Raspberry Pi, für FreeRTOS auf einem ESP8266 und für Riot auf einem Arduino M0 implementiert. Nun soll überprüft werden, ob die Implementierungen miteinander kompatibel sind. Das bedeutet, dass die von allen drei Systemen generierten Signaturen auch auf allen dreien positiv verifiziert werden. Anschließend werden die Ergebnisse der Evaluationen der einzelnen Systeme miteinander und mit Testwerten von RSA und ECDSA verglichen.

12.1.1 Kompatibilitätstest

Die drei Testgeräte und ein Notebook sind in einem lokalen Netzwerk miteinander verbunden. Für einen Testdurchlauf nimmt ein Mikrocontroller die Rolle des Signatur-Generator und die zwei anderen die des Verifizierers ein. Der Notebook sendet ein Startsignal an den Generator. Dieser erstellt daraufhin eine Signatur mit 128-bit-Sicherheit mithilfe zufällig gewählten Parametern. Bei Erfolg wird die Signatur mitsamt aller nötigen Parametern zu einer Nachricht geparsed. Da bei diesem Test nur die Kompatibilität der WalnutDSA-Implementierungen überprüft werden soll, wurde darauf verzichtet, ganze zu signierende Nachrichten zu verschicken. Stattdessen beinhalten die unter den Geräten ausgetauschten Nachrichten wie in der Vorlage von SecureRF zufällig generierte Hashwerte.[31, p.17] Die Nachricht wird per UDP an die beiden Verifizierer gesendet. Diese zerlegen das Paket wieder in seine Bestandteile und führen die Verifizierung durch, dessen Ergebnis an das Notebook gesendet und dort angezeigt wird. Dieser Vorgang wird eine bestimmte Anzahl mal wiederholt, woraufhin die Rolle des Generators an einen anderen Mikrocontroller weitergegeben wird.

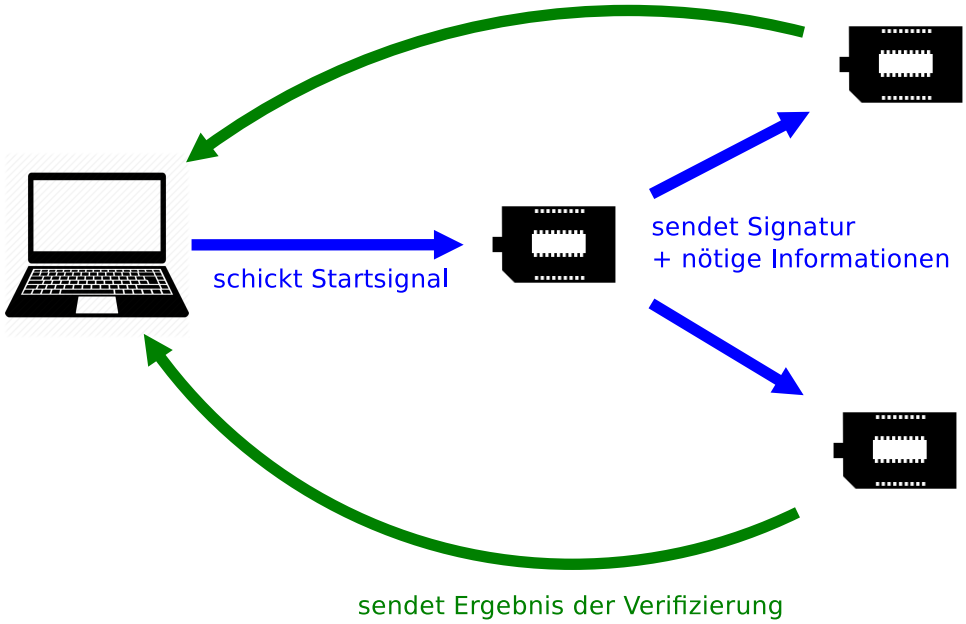


Abbildung 12.1: Aufbau des Kompatibilitätstests

Leider konnte ein derartiger Live-Test aus zeittechnischen Gründen nur mit dem ESP8266 und dem Raspberry Pi durchgeführt werden. Für die Kommunikation mit dem Arduino wurden die Nachrichten separat gespeichert und eingelesen. Alle generierten Signaturen wurden auf den jeweils beiden anderen Geräten erfolgreich verifiziert.

Es konnte leider keine Referenzimplementierung von SecureRF für einen weiteren Kompatibilitätstest verwendet werden. Eine Verwendung des Testdatensatzes aus dem WalnutDSA Paper brachte jedoch dieselben Ergebnisse, weshalb von einer Kompatibilität ausgegangen werden kann.[31, pp. 21-25]

12.1.2 Vergleich von Laufzeit und Signaturgröße

Die Laufzeit der Schlüsselpaargenerierung, Signierung und Verifizierung und der benötigte Speicherplatz der Signatur wurde für verschiedene Sicherheitslevel (n-bit-Sicherheit, angefangen mit der aktuell empfohlenen 112-bit-Sicherheit) auf den drei Geräten ermittelt. Als Vergleich werden Werte einer auf Level O3 optimierten C-Implementierung von SecureRF angegeben, die auf einem ARM Cortex M3 gemessen wurden. In der folgenden Tabelle werden ihnen auf einem Raspberry Pi ermittelte Werte von RSA und ECC gegenübergestellt. Die für das jeweilige Sicherheitslevel benötigte Schlüssellänge wurden aus einem NIST Report entnommen.[61] Es wurden die elliptischen Kurven *secp224k1*, *secp256k1* und *secp521r1* verwendet. Da für ECDSA kein Äquivalent für SL = 256 angegeben ist, wurde 521 Bit ECDSA berechnet. Die Zeitangaben der Verifizierung beinhalten nicht die Berechnung des Hashwertes aus der Nachricht.

	SL	Linux	FreeRTOS	Riot	SecRF	RSA	ECDSA
T(Key)	112	7,79 ms	25,5 ms	3,8 ms	-	6,29 s	75 ms
	128	9,64 ms	29,3 ms	6,2 ms	-	27,0 s	77 ms
	256	22,7 ms	61,0 ms	10,5 ms	-	>2 h	127 ms
T(Sig)	112	5,31 s	46,3 s	15,5 s	-	84,5 ms	3,7 ms
	128	6,77 s	50,3 s	30,5 s	-	263 ms	2,7 ms
	256	18,7 s	150,2 s	47,7 s	-	28,5 s	20,4 ms
T(Ver)	112	94,4 ms	338 ms	59,7 ms	-	2,35 ms	12,6 ms
	128	96,6 ms	366 ms	67,7 ms	5.7 ms	5,03 ms	4,8 ms
	256	135 ms	489 ms	87,2 ms	-	120 ms	78,4 ms
Gr(Sig)	112	659 Byte	337 Byte	371 Bytes	-	256 Byte	64 Byte
	128	704 Byte	461 Byte	560 Bytes	-	384 Byte	72 Byte
	256	1127 Byte	685 Byte	-	-	1920 Byte	138 Byte

Betrachtet man die Laufzeiten der Signaturgenerierung, so fällt auf, dass auf den ersten drei WDSA-Implementierung dazu sehr hohe Werte ermittelt wurden und von SecureRF dazu erst gar keine Angaben gemacht werden. Dies zeugt definitiv von einer aktuellen leistungstechnischen Schwachstelle bei der WalnutDSA Signaturgenerierung. Die Verifizierung wurde hingegen auf allen vier Modulen mittelmäßig bis sehr schnell durchgeführt. Die unterschiedlichen Ergebnisse machen deutlich, wie viel Zeit durch weitere Optimierung der Implementierung eingespart werden könnte. Verglichen mit RSA und ECDSA ist WDSA deutlich schneller.

12.2 Umsetzbarkeit der WalnutDSA-Implementierung

Die Tests der WDSA-Bibliotheken liefern auf allen Systemen selbst bei langen Braids passable Laufzeiten in der Verifizierung, aber zu lange Laufzeiten bei der Generierung. Dieser Abschnitt beinhaltet einige Gedanken, wie aktuelle Implementierungen des Verfahrens (trotzdem) eingesetzt werden könnten.

12.2.1 Auslagerung teurer Operationen

Die Signaturgenerierung benötigt im Vergleich zur Verifizierung sehr viele Rechenschritte und viel Speicherplatz. Will man den Einsatz von WalnutDSA in einem Netz von Geräten mit sehr wenig Ressourcen realisieren, ohne dabei immens hohe Laufzeiten zu erhalten, besteht theoretisch die Möglichkeit, die Generierung für alle Geräte auf eine leistungsfähigere Instanz auszulagern. Die prinzipielle Idee ist folgende: Will Bob eine signierte Nachricht schicken, schickt er zuerst den Hashwert seiner Nachricht an die Signierungsstelle Trent, der daraufhin mithilfe seines privaten Schlüssels eine Signatur erstellt und diese Bob zurückschickt. Bob kann daraufhin die signierte Nachrichten an Alice senden, solange Alice ebenfalls Trent vertraut und seinen öffentlichen Schlüssel kennt.

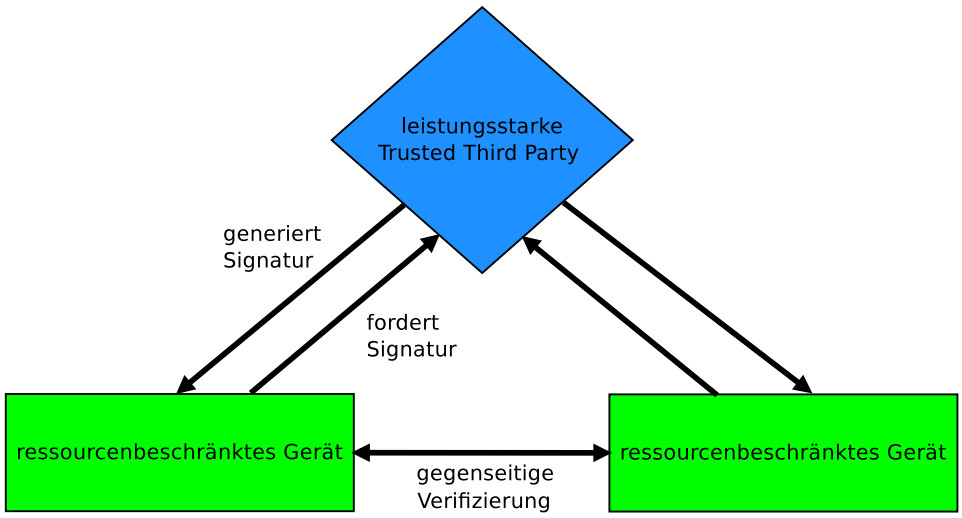


Abbildung 12.2: Prinzipielle Auslagerung der Signaturgenerierung

Eine derartige Struktur würde wahrscheinlich einen effizienteren signierten Nachrichtenaustausch ressourcenarmer Geräte ermöglichen, jedoch verletzt es das Konzept des gegenseitigen Vertrauens der einzelnen Netzteilnehmer. Da Trent Bobs Nachricht an Alice signiert, vertraut Alice niemals Bob, sondern stets nur Trent. Würde Trent ausfallen, wäre keine sichere Kommunikation zwischen den Teilnehmern mehr möglich, weswegen es sich bei der Auslagerung der Signaturgenerierung um keine optimale Lösung zur Effizienzsteigerung von WalnutDSA handelt.

Im Grunde liegt das Problem nicht bei der gesamten Signaturgenerierung, sondern nur bei ihrem letzten Schritt, der Umformung des Signaturbraids. Die Rohsignatur an sich ist sehr schnell erstellt, ohne dabei viel Speicher zu verbrauchen. Die Umformung folgt daraufhin mit der Rohsignatur als einzigen Parameter, daher kann diese genauso gut von einer anderen, evtl. stärkeren Maschine durchgeführt werden. Der Nachteil hierbei ist, dass die Übermittlung der nicht umgeformten Signatur ein Sicherheitsrisiko darstellt, da ohne Umformung die Cloaking-Elemente nur die Position des privaten Schlüssels verschleiern. Dieser liegt aber in Klartext vor und ist daher durch ein Abhören des Nachrichtenaustauschs verhältnismäßig leicht zu extrahieren. Deshalb sollte diese Methode nur in einem sicheren Netz nur mit vertrauenswürdigen Teilnehmern oder in Kombination mit einer (teuren) Verschlüsselung benutzt werden. Ein mögliches Anwendungsszenario wäre ein privates Heimnetz, in dem der Router beim Nachrichtenaustausch nach außen die Umformung übernimmt.

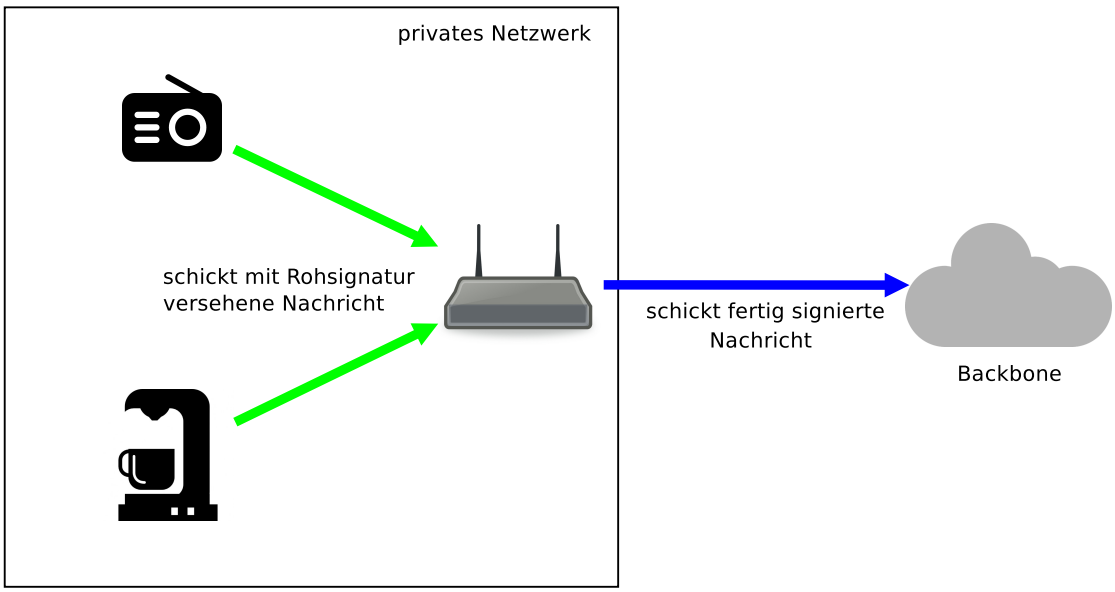


Abbildung 12.3: Beispielszenario für Auslagerung der Umformung

12.2.2 Authentizitätsgarantie durch Zertifizierungsstellen

Eine alleinige Signierung einer Nachricht durch den Absender ist anfällig für einen Man-in-the-middle-Angriff. Schickt Bob eine Nachricht an Alice besteht für Eve unter Umständen die Möglichkeit, die Nachricht abzufangen, zu verändern, den neuen Hashwert berechnen und mit ihrem eigenen privaten Schlüssel zu signieren. Solange die öffentlichen Schlüssel nicht eindeutig an die Identität geknüpft sind, merkt Alice eventuell nicht, dass sie die Nachricht mit Eves Schlüssel verifiziert und nicht mit Bobs. Bei einer Zertifizierungsstelle, im Englischen *Certificate Authority* (CA), handelt es sich um ein Mitglied im Netz, dem alle anderen Teilnehmer vertrauen. Die CA bürgt für die Identität der Mitglieder, indem sie auf Anfrage ein selbst-signiertes Zertifikat erstellt, in dem die Zugehörigkeit eines öffentlichen Schlüssels zu einer Entität beschrieben ist. Da jeder Netzteilnehmer der CA vertraut, kann jeder jedem bekannten oder fremden Gerät mit gültigen Zertifikaten der CA vertrauen.

Eine Möglichkeit der Zertifizierung mit WalnutDSA als Signaturalgorithmus ist die Erstellung von x.509 Zertifikaten. Der Standard bietet ein gutes Gerüst, um alle für die Verifizierung nötigen Informationen zu strukturieren. Die *Signature Algorithm ID* könnte neben den Bezeichner für WDSA zusätzlich die Braidgruppe N, die Größe des Galois-Körpers q und die Kodierungsgeneratoren enthalten. Die T-Werte, die Schlüsselmatrix und die Schlüsselpermutation bilden zusammen das Segment der Informationen zum öffentlichen Schlüssel. Abhängig von N und q beträgt der benötigte Speicher dafür $log_2(q) * N + log_2(q) * N * (N - 1) + N * log_2(N) = N^2 * log_2(q) + N * log_2(N)$ Bits. Bei $N = 8$ und $q = 256$ werden für das Schlüsselsegment 67 Bytes und für eine 128-Bit-Sicherheitslevel-Signatur im Durchschnitt 461 Bytes benötigt.

```
Certificate:
 Data:
 Version: 3 (0x2)
 Serial Number: bb:7c:54:9b:75:7b:28:9d
 Signature Algorithm: WDSAN8Q256G1347
 Issuer: C=MY, ST=STATE, O=CA COMPANY NAME, L=CITY, OU=X.509, CN=CA ROOT
 Validity
  Not Before: Apr 15 22:21:10 2008 GMT
  Not After : Mar 10 22:21:10 2011 GMT
 Subject: C=MY, ST=STATE, L=CITY, O=ONE INC, OU=IT, CN=www.example.com
 Subject Public Key Info:
  Public Key Algorithm: wdsaEMult
  WDSA Public Key: (536 bit)
   Modulus (536 bit):
   00:ae:19:86:44:3c:dd...
   ...
 Signature Algorithm: WDSAN8Q256
 52:3d:bc:bd:3f:50:92...
```

Abbildung 12.4: potenzielles x.509 Zertifikat mit WDSA

12.3 Sicherheit von Braids

Die getesteten Implementierungen für WalnutDSA können Signaturen mit erwünschten Sicherheitslevel erstellen. Diese n-Bit-Sicherheit kann aber nur garantiert werden, wenn kein effizienterer Angriff als ein *Brute-Force-Angriff* auf das Verfahren existiert. In diesem Abschnitt wird erklärt, warum Braids, oder im speziellen WalnutDSA, diese Eigenschaft auch unter Berücksichtigung der theoretischen Möglichkeiten eines Quantencomputers den Autoren des WDSA-Papers nach erfüllen soll.

12.3.1 Unumkehrbarkeit der E-Multiplikation

Als Einwegfunktion bezeichnet man in der Komplexitätstheorie eine mathematische Funktion, die leicht zu berechnen, aber schwer umzukehren ist. Die E-Multiplikation erfolgt in linearer, von der Braidlänge abhängigen Laufzeit. Beim Versuch die E-Multiplikation umzukehren, um den privaten aus den öffentlichen Schlüssel zu berechnen, stößt man jedoch auf ein kompliziertes Gleichungssystem mit N Variablen und langer Polynome, dessen Lösung zusätzlich durch die Permutationen während der E-Multiplikation erschwert wird. Es existiert noch kein Verfahren, das den privaten Schlüssel in angemessener Zeit aus dem öffentlichen Schlüssel berechnen kann.[31, p. 9]

12.3.2 Cloaked Conjugacy Search Problem

Zwei Elemente einer Braidgruppe $u, w \in B_N$ heißen konjugiert, wenn ein $v \in B_N$ existiert, sodass gilt: $w = v^{-1}uv$. Bei dem „*Conjugacy Search Problem*“ geht es darum für zwei konjugierte Braids u und w jenes v zu finden. Ältere auf Braids basierende Verfahren der asymmetrischen Kryptographie waren sicher unter der Annahme, CSP wäre schwierig. Jedoch existieren mittlerweile effiziente Lösungen für das Problem, weshalb auf Braids basierende kryptographische Verfahren, wie z.B. das Diffie-Hellman-Braid-Protokoll, als unsicher galten.[21] Ohne Cloaking-Elemente besäße auch eine Signatur von WalnutDSA die Form $Sig = R(priv(S)^{-1}E(H(M))priv(S))$ und das Verfahren wäre nur so sicher wie CSP. Die Cloaking-Elemente in der WDSA-Signatur $Sig = R(v_2v_1^{-1}priv(S)^{-1}vE(H(M))priv(S)v_1)$

„*verhüllen*“ den privaten Schlüssel und auch wenn sie trivial zu sein scheinen, da sie in der E-Multiplikation keinen Effekt haben, machen sie längenbasierte Angriffe nutzlos und es existiert kein effizientes Verfahren, sie vom Schlüssel zu trennen. Deshalb sind auch die Techniken zur effizienten Lösung des CSPs nicht anwendbar. SecureRF nennt das Problem eine Instanz des „$Cloaked\,Conjugacy\,Search\,Problems$“, für das noch keine Lösung in sub-exponentieller Zeit existiert. [31, pp. 9-10]

12.3.3 Quantenresistenz

Es ist nicht sicher, wann der erste voll funktionsfähige Quantencomputer fertiggestellt sein wird. Einige Versuche mit Maschinen, die es schaffen, sich in bestimmten Fällen wie jene zu verhalten, zeigen jedoch eindeutig, dass Quantenrechner in Zukunft nicht nur in der Theorie eine Gefahr für viele asymmetrische Verfahren der Kryptographie darstellen. Besonders betroffen sind Methoden, die auf der Schwierigkeit, große Zahlen in ihre Primfaktoren zu zerlegen, basieren, wie zB. RSA, aber auch Methoden diskreter Logarithmen, z.B. ECC. Der Shor-Algorithmus nutzt die Quanten-Fourier-Transformation, um beide Probleme in sub-exponentieller Zeit zu lösen. [26] Im Kontext der Kryptographie auf Quantencomputern wird neben Shor oft ein weiterer Name erwähnt. Beim Grover-Algorithmus handelt es sich um ein sehr effizientes Suchverfahren, das bei einer erfolgreichen Implementierung auf einem Quantencomputer das gesuchte Element unter N Kandidaten stets in $O(\sqrt{N})$ Schritten findet. Eine quadratische Beschleunigung in der Suche würde sich auf alle Verfahren mit Schlüsselpaaren, Hashfunktionen oder anderen Input/Output-Probleme auswirken und ihr Sicherheitslevel auf die Hälfte drücken. Auf Braids basierende Kryptographie oder im Speziellen WalnutDSA stellt in beiden Fällen eine gute Lösung dar. Zum einen ist der Shor-Algorithmus auf Signaturen von WDSA nicht anwendbar, da es sich bei Braids nicht um eine zyklische und abelsche Gruppe mit endlich vielen Elementen handelt. Braids verhalten sich nicht kommutativ und können unendlich lang werden. Mit zunehmender Länge wächst der Aufwand der Validierung der Signaturen nur linear, weshalb ein doppelt so hohes Sicherheitslevel im Gegensatz zu z.B. RSA leicht erreicht und dadurch auch dem Grover-Algorithmus leicht entgegengewirkt werden kann. Somit besitzen Braids nach aktuellem Wissensstand keine gravierenden Sicherheitslücken, die durch das erfolgreiche Fertigstellen eines Quantenrechners ausgenutzt werden könnten.[62, p. 5]

12.4 Die Zukunft von WalnutDSA im Bereich Internet der Dinge

Beim WalnutDSA handelt es sich um eine vielversprechendes Signaturverfahren, das die Vorteile von Braids sinnvoll nutzt. Ob es sich im Bereich Internet der Dinge in naher Zukunft effizient implementieren lässt, ist jedoch aufgrund der ineffizienten Umformungsverfahren nicht garantiert. Des Weiteren lässt sich noch nicht sagen, ob das Verfahren langfristig als sicher gelten wird. Zwar scheint mit den Cloaking-Elementen ein wirksames Werkzeug gegen Lösungen des CSPs, die bisher größte bekannte Schwachstelle von Braids, zu existieren 12.3.2, jedoch wurden auf Braids basierende Algorithmen im Vergleich zu anderen asymmetrischen kryptographischen Verfahren bisher wenig untersucht. Braids verhalten sich ganz anders als andere für die Kryptographie verwendeten algebraischen Konstrukte, daher ist es heute noch ungewiss, ob sie nicht eine bisher völlig unbekannte Schwachstelle besitzen, die evtl. sogar von Quantencomputern ausgenutzt werden könnte. Bis es aber soweit ist beherbergen Braids vor allem Dank ihrer Immunität gegenüber Shors und Grovers Algorihmen

12.3.3 und dem mit der Braidlänge linear wachsenden Ressourcenaufwand bei der Verifizierung ein großes Potenzial in der Kryptographie für ressourcenbeschränkte Geräte.

13 Zusammenfassung und Ausblick

Das Ziel der Arbeit, WalnutDSA als Kernelmodul für Linux auf dem Raspberry Pi zu implementieren, wurde erreicht. Auch die geplante Einbindung in die Crypto-API war erfolgreich. Durch die Variabilität der verwendeten Braidgruppe und der Ordnung des endlichen Körpers wurde eine hohe Kompatibilität mit anderen Implementierungen gewährleistet. Die Einbettung in Linux sorgt zudem für eine starke Plattformunabhängigkeit, zumindest bezüglich vergleichsweise leistungsstärkerer Geräte, die vom Linux Kernel unterstützt werden. Zur Evaluation wurde die Umsetzung erfolgreich gegen sich selbst, zwei andere Implementierungen und einen beispielhaften Datensatz aus offizieller Quelle getestet. Die Leistung der Implementierungen von SecureRF konnte nicht erreicht werden, wenngleich eine Optimierung des Programmcodes auch nicht zum Aufgabenbereich dieser Arbeit gehörte. In Hinblick darauf sind einige Verbesserungen an der Implementierung denkbar.

Wie bereits angesprochen, ist seit der Umsetzung des Verfahrens bereits eine neue Auflage der wissenschaftlichen Arbeit zu WalnutDSA erschienen. Um die theoretische Sicherheit der Implementierung auch in Zukunft zu garantieren, sollten die Änderungen am Verfahren eingearbeitet werden. Die Konvertierung der Signatur in die BKL-Normalform nimmt den größten Teil der Laufzeit bei der Signierung ein. Während die Umformung aufgrund der dargestellten Sicherheitseigenschaften nicht ausgelassen werden kann, so könnte speziell die Implementierung dieser Funktion weiter optimiert werden oder gar an neuen, effizienteren Umformungsfunktionen geforscht werden. Es ist prinzipiell auch denkbar, einige Teiloperationen von WalnutDSA zu parallelisieren. So beinhalten bspw. einige Teile der Signatur keine Abhängigkeiten zueinander und könnten auf mehreren Rechenkernen separat generiert und anschließend zusammengesetzt werden. Auch die BKL-Normalform eignet sich gut zur Paralellisierung [32]. Wenngleich auf dem verwendeten Raspberry Pi nur ein Rechenkern vorhanden ist, so würden doch andere Geräte von dieser Verbesserung profitieren können.

Abschließend sei gesagt, dass WalnutDSA im Kontext von ressourcenschwachen Geräten des IoT-Szenarios effektiv zur Sicherung der Kommunikation beitragen kann. Wie dargestellt, ist die Implementierung des Verfahrens um ein Vielfaches schneller als RSA und liefert sogar bessere Ergebnisse als ECDSA. Durch das lineare Wachstum der benötigten Rechenzeit der E-Multiplikation können sich diese Erfolge mit steigenden Sicherheitsanforderungen nur noch weiter verbessern. Zudem stellen Braid-basierte Kryptosysteme wie WalnutDSA durch den Verzicht auf abelsche Gruppen beim Entwurf der Einwegfunktion einen vielversprechenden Lösungsansatz zur Post-Quanten-Kryptographie dar. Das digitale Signaturverfahren beinhaltet damit viel Potential für ein zukunftsträchtiges asymmetrisches Kryptosystem.

14 Schluss

Das Ziel dieser Arbeit war den WalnutDSA-Algorithmus für FreeRTOS zu implementieren und durch eine Evaluierung seine praktische Anwendbarkeit zu überprüfen. Das WDSA-Modul erstellt und verifiziert auf einem ESP8266 erfolgreich 685 Byte große Signaturen mit 256bit-Sicherheit oder niedriger. Seine Kompatibilität mit zwei weitere Systeme für Riot und dem Linux Kernel wurde in einem Test erfolgreich nachgewiesen. Laufzeittests der Verifizierung ergeben durchaus passable Ergebnisse und zeugen von dem erwünschten, mit dem Sicherheitslevel linear wachsenden Ressourcenaufwand der E-Multiplikation. Die gemessenen Zeiten der Signaturgenerierung liegen jedoch noch weit über den für die praktische Anwendung auf Mikrocontrollern erwünschten Wert. Den Grund dafür bilden zum größten Teil die verwendeten Funktionen zur Umformung der Rohsignatur, deren Implementierungen mit Abstand am meisten Ressourcen benötigen. Jedoch zeigen die Testwerte der Referenzimplementierung für Riot auf dem Arduino, dass sich durch Optimierung des Codes nicht nur bei der Generierung, sondern auch bei der E-Multiplikation viel Zeit sparen ließe. Durch eine Parallelisierung oder Programmierung in Assemblersprache könnte die Signierung noch weiter beschleunigt werden, wodurch jedoch tendenziell wieder mehr Speicher benötigt wird. Ob eine deutlich effizientere Methode zur effektiven Umformung der Rohsignatur existiert, ist unklar, aber durchaus denkbar, weshalb in diese Richtung zweifelsohne weiter geforscht werden sollte. Grundsätzlich handelt es sich bei der Braid-basierten Kryptographie im Vergleich zu ihren traditionelleren Richtungen wie die der Faktorisierung oder diskreter Logarithmen um ein noch sehr unerforschtes Gebiet, weshalb sich ihre Rolle in fernerer Zukunft heute nur schwierig vorhersagen lässt. So werden auch von SecureRF, die Entwickler von WalnutDSA, in unregelmäßigen Abständen aktualisierte Versionen des WalnutDSA-Papers mit nicht trivialen Änderungen des Algorithmus veröffentlicht. Durch die *Cloaking-Elemente*, die die Signatur vor Lösungen des *Conjugacy Search Problems* schützt, gewinnen Braids jedoch mehr als ihre Daseinsberechtigung in der Kryptographie zurück. In Kombination mit der Tatsache, dass sich Elemente der Braidgruppe nicht kommutativ verhalten, ist WalnutDSA sowohl resistent gegenüber dem Shor- als auch dem Grover-Algorithmus und bergt daher auch in einer Zukunft mit Quantencomputer großes Potenzial.

15 Schluss

Das Ziel von dieser Arbeit ist der WalnutDSA für RIOT OS zu implementieren. Die wichtige Merkmal von WalnutDSA ist, dass die Verfizierung von Signatur sehr schnell ist und die Laufzeit von Schlüsselgenerierung und Verifizierung mit Sicherheitslevel linear steigt, während z.B. die Laufzeit von Schlüsselgenerierung von RSA exponentiell wächst. Um den Aufbau von WalnutDSA Signatur verständlich zu machen, der aus Braids bestehen, wurde die theoretische Grundlage von Braid erklärt. Das anschließende Kapitel beschreibt, wie die einzelne Komponente und Abläufe von WalnutDSA aufgebaut sind. Im Kapitel 11 wurde über die Implementierungsdetails gehandelt. Die konkrete Umsetzung von Datenstrukturen und Algorthmen wurde beschrieben und sämtliche bereits umgesetzte Optimierungsmöglichkeiten wurden betrachtet. Über die Testergebnisse kann man feststellen, dass die genannte Vorteile von WalnutDSA gewährleistet sind. Die Korrektheit von der Implementierung wurde gegen die andere WalnutDSA Implementierung für andere Plattform getestet. Als Diskussion wurde die mathematische Probleme von WalnutDSA geschildert, über sie WalnutDSA sicher ist, und die Umsetzungbarkeit von WalnutDSA wurde gehandelt.

Die Implementierung gibt es noch Optimierungsmöglichkeiten.
E-Multiplikation ist eine Kernoperation von WalnutDSA, von der das Performance von Verifizierung und Schlüsselgenerierung stark abhängt. Und die Galois-Multiplikation ist die Operation, die bei der E-Multiplikation am meisten Zeit benötigt. Neben dem Einsatz von besseren Galois-Multiplikationsalgorithmus kann man sich überlegen, die Anzahl von Galos-Multiplikation überhaupt zu kürzen. Aktuell in der Implmentierung muss für jede E-Multiplikationschritte 16 Galois-Multiplikation soll berechnet werden. Jedoch kommt es häufig vor, dass keine Rechnung nötig ist, falls 1 oder 0 stehen. Da die letzte Zeile von Matrix bis auf dem letzten Eintrag mit 0 gefüllt ist, kann man wenigstens die Fälle überspringen lassen.
Bei der Umformungsfunktionen, von der die Signaturgenerierungszeit abhängt, stehen mehre Verbesserungsmöglichkeiten offen. Laut [32] ist BKL-Normalform parallelisierbar. Dadurch kann man die Komplexität von BKL-Normalform von $\emptyset(l^2n$ zu $\emptyset(ln$ reduzieren, falls $O(l)$ Prozessoren vorhanden ist. Auch der Henkelreduktionsalgorithmus kann verbessert werden. Neben die weitere Optimierungen mit der Verschiebungen in Array, könnte man sich überlegen auch einen anderen Algorithmus zu verwenden, der von dem selben Entwickler stammt. Dieser heißt Coarse-Reduction.
Beim WalnutDSA finden noch Aktualisierungen statt, die potenzielle Sicherheitslücken auffüllen. Also auch in Zukunft soll die Implementierung an den Neuerungen von WalnutDSA angepasst werden.

Anhang A

WalnutDSA-Bibliothek für FreeRTOS

Dokumentation

von Konrad Haslberger am 16.12.2017

Generated by Doxygen 1.8.14

Inhaltsverzeichnis

Kapitel 1

Verwendung der WalnutDSA-Bibliothek

1.1 Verwendungszweck

Bei der WalnutDSA-Bibliothek handelt es sich um ein Modul für FreeRTOS zur Generierung und Verifizierung von WalnutDSA-Signaturen.

1.2 Vorbereitung

Für die Benutzung der WalnutDSA-Bibliothek wird FreeRTOS Version 9.00 oder neuer benötigt.

1.3 Erstellen einer FreeRTOS-Applikation

Der einfachste Weg, das Schreiben einer FreeRTOS-Applikation zu beginnen ist, bei den gerätespezifischen Demoapplikationen anzufangen und diese zu modifizieren, da sie bereits die hardwarespezifischen Portfiles inkludieren. In der Main-Methode werden die Tasks mithilfe der Methode $xTaskCreate()$ generiert.

1.4 Einbinden der WDSA-Bibliothek

Durch das Einbinden des Headers wdsa.h wird der Zugriff auf alle wichtigen Methoden von WnalnutDSA ermöglicht.

Vor der ersten Anwendung müssen folgende Makros aus wdsa.h bzw. braids.h eventuell angepasst werden:

- **Q** - die Größe des verwendeten Binärkörper - die Implementierung verwendet für ein Galois Element 1 Byte, weswegen der Wert für **Q** nicht größer als 256 sein darf.
- **IRRPOL** - irreduzible Polynom für die Multiplizierung im Binärkörper - muss nach Veränderung von **Q** auch angepasst werden
- **HASHSIZE** - Größe des Outputs der verwendeten kryptographischen Hashfunktion in Bytes
- **RAND_ADDR** - Speicheradresse zur Generierung von Zufallszahlen durch die Hardware - muss angepasst werden, wenn kein ESP8266 benutzt wird
- **REWRITE** - Grad der Umformung nach Generieren der Rohsignatur
  - 0 : keine Umformung (nicht sicher)
  - 1 : Umformung in Links-BKL-Normalform (sicher, dauert lange und benötigt viel Speicherplatz)
  - 2 : erst BKL, dann Kürzen der Signatur durch Dehornoy-Algorithmus (sicher, speicherbillig, dauert sehr lange)
- **DEFAULT_SECURITY_LEVEL** - Sicherheitslevel, an dem sich die Implementierung orientiert, wenn für die Generierung kein Sicherheitslevel angegeben wird

Der Braidgrad **N** darf nicht verändert werden, da die verwendeten Strukturen für $\mathbf{N} = 8$ ausgelegt sind!

Kapitel 2

Übersicht über die wichtigen Funktionen

2.0.0.1 wdsaGenerate()

```
struct braid wdsaGenerate (
             struct braid privateKey,
             uint8_t * encodeGenerators,
             uint8_t * hash,
             uint8_t * tValues,
             uint16_t securityLevel,
             uint8_t A,
             uint8_t B )
```

Hauptfunktion zur Signaturgenerierung.

Parameters

privateKey	der private Schlüssel, ein zufälliger frei reduzierter Braid; muss Sicherheitslevel erfüllen
encodeGenerators	muss vier Generatoren zur Hashkodierung enthalten. Für jeden Generator gen gilt: 1 <= gen < Braidgruppe N
hash	Hash der zu signierenden Daten der Länge HASHSIZE
tValues	N positive T-Values < Q; der a-te und b-te T-Value muss mit 1 belegt sein
securityLevel	Sicherheitslevel n heißt ein Angreifer braucht 2^n Versuche, das Verfahren zu brechen; wenn 0, wird DEFAULT_SEC_LVL benutzt
a	Parameter für Cloaking-Elements; es muss gelten 1 < a < N; bei 0 wird a anhand der T-Values bestimmt
b	Parameter für Cloaking-Elements; es muss gelten 1 < b < N; bei 0 wird a anhand der T-Values bestimmt

2.0.0.2 wdsaVerify()

```
_Bool wdsaVerify (
            struct braid signatureBraid,
            struct CB publicKey,
            uint8_t * tValues,
            uint8_t * encodeGenerators,
            uint8_t * hash )
```

verifiziert Signatur. gibt 0 für negativ und 1 für positiv zurück

Parameters

signatureBraid	zu verifizierende Signatur
publicKey	öffentlicher Schlüssel: Tupel aus NxN Matrix und Permutation mit N Elementen
tValues	N positive T-Values $<$ Q; der a-te und b-te T-Value muss mit 1 belegt sein
encodeGenerators	muss vier Generatoren zur Hashkodierung enthalten. Für jeden Generator gen gilt: 1 $<=$ gen $<$ Braidgruppe N
hash	Hash der zu signierenden Daten der Länge HASHSIZE

2.0.0.3 computePublicKey()

```
struct CB computePublicKey (
            struct braid privateKey,
            uint8_t * tValues )
```

berechnet CB-Repräsentation für einen Braid; wird benutzt um aus dem privaten Schlüssel den öffentlichen zu berechnen

Parameters

privateKey	umzuwandelnder Schlüssel
tValues	N positive T-Values $<$ Q; der a-te und b-te T-Value muss mit 1 belegt sein

2.0.0.4 eMultiplicate()

```
struct CB eMultiplicate (
            struct CB cb,
            struct braid b,
            uint8_t * tValues )
```

e-multipliziert eine CB mit einem Braid

Parameters

cb	1. Faktor: Braid in Colored-Burau-Form
b	2. Faktor: Braid in Artin-Repräsentation
tValues	N positive T-Values $<$ Q; der a-te und b-te T-Value muss mit 1 belegt sein

2.0.0.5 generatePrivateKey()

```
struct braid generatePrivateKey (
            uint16_t securityLevel )
```

generiert einen zufälligen, frei reduzierten Braid für das angegebene Sicherheitslevel

Parameters

securityLevel	Sicherheitslevel n heißt ein Angreifer braucht 2^n Versuche, das Verfahren zu brechen; wenn 0, wird DEFAULT_SEC_LVL benutzt

2.0.0.6 copyBraidInArray()

```
int8_t* copyBraidInArray (
            struct braid b )
```

wandelt ein struct braid in ein int8-Array um

Abbildungsverzeichnis

Literaturverzeichnis

[1] Chris Ip. Hong kong iot conference. In *The IoT opportunity — Are you ready to capture a once-in-a lifetime value pool?*, 2016.

[2] Z. K. Zhang, M. C. Y. Cho, C. W. Wang, C. W. Hsu, C. K. Chen, and S. Shieh. Iot security: Ongoing challenges and research opportunities. In *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*, 2014.

[3] S. Notra, M. Siddiqi, H. Habibi Gharakheili, V. Sivaraman, and R. Boreli. An experimental study of security and privacy risks with emerging household appliances. In *2014 IEEE Conference on Communications and Network Security*, 2014.

[4] Kerry A. McKay, Larry Bassham, Meltem Sönmez Turan, and Nicky Mouha. Report on lightweight cryptography. Technical report, US National Institue of Standards and Technology, 2017.

[5] Iris Anshel, Derek Atkins, Dorian Goldfeld, and Paul E Gunnells. Walnutdsa(tm): A quantum-resistant digital signature algorithm. Cryptology ePrint Archive, Report 2017/058, 2017. https://eprint.iacr.org/2017/058.

[6] Derek Atkins. Walnut digital signature algorithm: A lightweight, quantum-resistant signature scheme for use in passive, low-power, and iot devices, 2016.

[7] Mark Weiser. The computer for the 21st century.

[8] Matthias Reinwarth. Iot - die sicherheit der dinge.

[9] RSA Key Lengths. https://www.javamex.com/tutorials/cryptography/rsa_key_length.shtml.

[10] Hans Delfs. *Introduction to Cryptography.* Springer, 2015.

[11] Stephan Spitz. *Kryptographie und IT-Sicherheit : Grundlagen und Anwendungen.* Viweg + Teubner, 2011.

[12] Information technology – Security techniques – Non-repudiation – Part 2: Mechanisms using symmetric techniques. Standard, International Organization for Standardization, December 2010.

[13] Elaine Barker. Digital signature standard(dss), 2013. https://csrc.nist.gov/publications/detail/fips/186/4/final.

[14] Daniel J. Bernstein. *Post-quantum cryptography.* Springer, 2009.

[15] Jyrki T. J. Penttinen. *Wireless Communications Security: Solutions for the Internet of Things.* Wiley, 2016.

[16] Amir Manzoor. *Securing Device Connectivity in the Industrial Internet of Things*. Springer, 2016.

[17] Emil Artin. Theorie der Zöpfe. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg 4*, 1925.

[18] I. Anshel, M. Anshel, and D. Goldfield. An algebraic method for public-key cryptography. *Mathematical Research Letters 6*, 1999.

[19] Ki Hyoung Ko, Sang Jin Lee, Jung Hee Cheon, Jae Woo Han, Ju-sung Kang, and Choonsik Park. *New Public-Key Cryptosystem Using Braid Groups*, pages 166–183. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[20] Maurice Chiodo. An introduction to braid theory, 2005.

[21] David Garber. Braid group cryptography. 2007.

[22] Iris Anshel. Colored burau matrices, e-multiplication, and the algebraic erasertm key agreement protocol. 2015.

[23] Scott Vanstone Darrel Hankerson, Alfred J. Menezes. *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[24] Ecrypt ii yearly report on algorithms and keysizes (2011-2012), 2012.

[25] Nicholas Jansma and Brandon Arrendondo. Performance comparison of elliptic curve and rsa digital signatures, 2004.

[26] Peter W. Shor. Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Sci. Statist. Comput.*, 26, 1997.

[27] Chris Peikert. What does gchq's ,,cautionary tale" mean for lattic cryptography?, 2016. https://web.archive.org/web/20160317165656/http://web.eecs.umich.edu/ cpeikert/soliloquy.html.

[28] Vadim Lyubashevsky Tim Güneysu and Thomas Pöppelmann. *Practical Lattice-Based Cryptography: A Signature Scheme for Embedded Systems*, pages 530–547. Springer Berlin Heidelberg, 2012.

[29] Iris Anshel, Derek Atkins, Dorian Goldfield, and Paul E. Gunnells. Post quantum group theoretic cryptography, 2016.

[30] Ki Hyoung Ko, Doo Ho Choi, Mi Sung Cho, and Jand Won Lee. New signature scheme using conjugacy problem, 2002.

[31] Dorian Goldfeld Iris Anshel, Derek Atkins and Paul E. Gunnells. Walnutdsa: A quantum-resistant digital signature algorithm. 2017.

[32] Jae Choon Cha, Ki Hyoung Ko, Sang Jin Lee, Jae Woo Han, and Jung Hee Cheon. An efficient implementation of braid groups. In *Advances in Cryptology - ASIACRYPT 2001*, 2001.

[33] Joan Birman, Ki Hyoung Ko, and Sang Jin Lee. A new approach to the word and conjugacy problems in the braid groups. *Advances in Mathematics 139*, 1998.

[34] David Garber. Braid group cryptography. *World Scientific Review*, 2008.

[35] Do Hwan Kim. Implementierung und Evaluation von WalnutDSA als Modul für das Betriebssystem RIOT. Bachelorarbeit, Ludwig-Maximilians-Universität München, 2017.

[36] Patrick Dehornoy. A fast method for comparing braids. *Advances in Mathematics 125*, 1997.

[37] Wayne Williams. Raspberry pi founder eben upton talks sales numbers, proudest moments, community projects, and raspberry pi 4 [q&a], 2017.

[38] Markus Montz. c't raspberry pi: Smart home im eigenbau und mehr, 2017.

[39] Cho Jin-young. Iot operating system - linux takes lead in iot market keeping 80% market share, 2017.

[40] As 3955-1991 (c90) standard für die progammiersprache c.

[41] Fast galois field arithmetic library in c/c++.

[42] Gadiel Seroussi. Table of low-weight binary irreducible polynomials. 1998.

[43] Kernel crypto api interface specification.

[44] Github: smuellerdd/libkcapi – linux kernel crypto api user space interface library.

[45] libkcapi dokumentation – kernel interfaces.

[46] libkcapi - linux kernel crypto api user space interface library.

[47] Valgrind home.

[48] Kernel memory leak detector documentation.

[49] Kryptographische verfahren: Empfehlungen und schlüssellängen. Technical report, Bundesamt für Sicherheit in der Informationstechnik, 2017.

[50] Freertos - anzahl downloads 2014. https://www.fs-net.de/de/support/freertos/.

[51] Freertos - homepage. https://www.freertos.org/index.html.

[52] Freertos - customisation of applications. https://www.freertos.org/a00110.html.

[53] Freertos - tasks. https://www.freertos.org/taskandcr.html.

[54] Freertos - task states. https://www.freertos.org/RTOS-task-states.html.

[55] Freertos - queues. https://www.freertos.org/Embedded-RTOS-Queues.html.

[56] Freertos - tick. https://www.freertos.org/implementation/a00011.html.

[57] Freertos - speicherverwaltung. https://www.freertos.org/a00111.html.

[58] Espressif. Esp8266ex datasheet. 2017.

[59] ESP8266 - Zufallszahlengenerator. esp8266-re.foogod.com/wiki/Random_Number_Generator.

[60] Emmanuel Baccelli, Oliver Hah, Mesut Günes, Matthias Wählisch, and Thomas C. Schmid. Riot os: Towards an os for the internet of things. In *Proceedings of the 32nd IEEE International Conference on Computer Communications (INFOCOM) Poster Session*, April 2003.

[61] Recommendation for Key Management. https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-4/final.

[62] SecureRF Corporation Derek Atkins. Walnut digital signature algorithm tm : A lightweight, quantum-resistant signature scheme for use in passive, low-power, and iot devices. 2016.