Universität Bremen

Fachbereich 3: Mathematik und Informatik

Masterarbeit

Analyse von SHA256 mit Hilfe von CryptoMiniSat

Lars Schmertmann

Matrikel-Nr.2469187

28. April 2017

1. Gutachter: Prof. Dr. Rolf Drechsler
2. Gutachter: Dr.-Ing. Olaf Bergmann
Betreuer: Dr. Stephan Eggersglüß und Dr. Daniel Große

Lars Schmertmann

Analyse von SHA256 mit Hilfe von CryptoMiniSat

Masterarbeit, Fachbereich 3: Mathematik und Informatik

Universität Bremen, April 2017

Datum in der settings.tex anpassen

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Bremen, den 28. April 2017

Lars Schmertmann

Danksagung

Auf diesem Wege möchte ich mich bei Stephan Eggersglüß und Daniel Große für die hervorragende Unterstützung und die nützlichen Tipps bedanken.

Bedanken möchte ich mich auch bei der AG Rechnerarchitektur dafür, dass ich die für diese Arbeit notwendigen Berechnungen auf ihrem Server durchführen konnte. Ebenfalls bedanken möchte ich mich bei der AG-Rechnernetze, in deren Räumlichkeiten ich die notwendig Ruhe für das Schreiben dieser Arbeit finden konnte.

Ein weiterer Dank gilt Jens Trillmann, mit dem ich zahlreiche Diskussionen rund um das Thema führen konnte und der das in dieser Arbeit erstellte Framework dazu genutzt hat, den Data Encryption Standard (DES) in eine konjunktive Normalform zu überführen.

Schließlich geht ein Dank an Dominik Menke, der die LaTeX-Vorlage für diese Arbeit erstellt, und für die Öffentlichkeit zur Verfügung gestellt hat.

Zusammenfassung

Ziel dieser Arbeit ist es, mit Hilfe von CryptoMiniSat Wissen über den Hash-Algorithmus SHA-256 zu erwerben. Dafür wird eine konjunktive Normalform erstellt und für die Analyse mit CryptoMiniSat genutzt. Die Analyse erfolgt iterativ. Nach einem Lösungsversuch werden gelernte Klauseln extrahiert und den betreffenden Bereichen von SHA-256 zugeordnet. Diese Klauseln werden in weitere Lösungsversuche direkt mit eingebunden. Es zeigt sich, dass dadurch der Lösungsprozess beschleunigt werden kann. Ein praktisch relevanter Angriff ist damit jedoch nicht möglich.

Inhaltsverzeichnis

List of TODOs

remove \listoftodos again

Kapitel 1

Einleitung

Diese Arbeit beschäftigt sich mit der Erzeugung und Optimierung einer Eingabe für einen SAT-Solver. In den folgenden Abschnitten wird erläutert, welche Motivation dahinter steht und kurz auf andere Arbeiten eingegangen, die sich mit diesem Thema beschäftigt haben. Im Anschluss wird das Ziel definiert und das Vorgehen erklärt.

1.1 Motivation

Hash-Algorithmen zeichnen sich dadurch aus, dass die Berechnung eines Hash zwar einfach, aber nicht umkehrbar ist. Verwendet werden sie in vielen sicherheitskritischen Umgebungen, in denen der Hash als eine Art Fingerabdruck von Daten genutzt wird, um eine Manipulation auszuschließen. Dazu gehört auch die digitale Währung „Bitcoin“ (siehe Abschnitt 2.6), in der sich durch „Mining“ Geld verdienen lässt. Mining bedeutet in diesem Fall so lange Fingerabdrücke zu erzeugen, bis ein für das System passender Fingerabdruck dabei herauskommt. Eine effiziente Umkehrung der Berechnung würde diesen Prozess wesentlich einfacher machen.

An diesem Punkt rücken SAT-Solver (siehe Abschnitt 2.4) in den Fokus. Diese versuchen eine Lösung für ein Problem zu finden, ohne dass dabei die Richtung der Berechnung eine Rolle spielt. Das allgemeine Problem ist in diesem Fall die Beschreibung eines Hash-Algorithmus, wobei auf die Vorgabe einer Eingabe und Ausgabe verzichtet wird. Ohne diese Vorgabe kann der SAT-Solver eine beliebige Lösung generieren, die dem Problem entspricht. Im Fall eines Hash-Algorithmus ist die Lösung eine Eingabe und der dazu passende Fingerabdruck. Wird zusätzlich der Fingerabdruck vorgegeben, versucht der SAT-Solver dafür eine gültige Eingabe zu finden.

Andere Arbeiten zu diesem Thema (siehe Abschnitt Abschnitt 1.2) haben bereits einen Versuch unternommen, Hash-Berechnungen auf diesem Weg umzukehren. Diese Arbeiten beschränken sich jedoch darauf, eine mehr oder weniger optimierte Eingabe für SAT-Solver zu erzeugen und diese mit verschiedenen SAT-

Solvern zu testen. Im besten Fall werden außerdem noch einige Versuche zu unternommen, die optimalen Einstellungen spezifischer SAT-Solver zu ermitteln.

An diesem Punkt setzt diese Arbeit an und geht einen Schritt weiter. Während eines Lösungsversuchs erwerben SAT-Solver zusätzliches Wissen über das gestellte Problem. Im Allgemeinen wird dieses Wissen bei Programmende verworfen. In dieser Arbeit wird das erworbene Wissen analysiert, verallgemeinert und von Anfang an in weitere Lösungsversuche integriert, in der Hoffnung diese dadurch zu beschleunigen.

1.2 Verwandte Arbeiten

Vegard Nossum beschäftigt sich in seiner Masterarbeit [V N12] mit SHA1 und MiniSat. Seinen dafür geschriebenen Programmcode hat er auf GitHub [V N13] veröffentlicht. Hauptbestandteil seiner Masterarbeit ist die Erzeugung unterschiedlicher Repräsentationen von SHA1 für MiniSat um diese auf ihre Performance zu vergleichen. Außerdem versucht er die richtigen Parameter für MiniSat zu finden, die den Lösungsprozess beschleunigen. Interessant ist sein Ansatz, am Ende seiner Masterarbeit gelernte Klauseln zu sammeln und für weitere Lösungsversuche einzusetzen. Problematisch ist dabei sein Vorgehen, für SHA1 allgemeingültige Klauseln anhand einer Statistik zu ermitteln. Allgemeingültige Klauseln ermittelt er durch 100 Versuche, bei denen er den SAT-Solver „clasp“ jeweils 300 Sekunden nach Eingaben für unterschiedliche Hashwerte suchen lässt. Wird eine Klausel in 90 Fällen gelernt, gilt sie als allgemeingültig. Weitere Tests mit diesen zusätzlichen Klauseln beschleunigen den Lösungsprozess laut seiner Aussage nicht.

Martin Maurer baut in seinem Projekt auf GitHub [M M13] auf der Masterarbeit von Vegard Nossum auf. Hierbei handelt es sich nur um einen oberflächlichen Versuch den Programmcode von Vegard Nossum zu nutzen, um damit eine Repräsentation von SHA-256 für CryptoMiniSat zu erzeugen, die schließlich zur Lösung eines Bitcoin-Blocks dienen soll. In seinen Versuchen kommt er zu dem Schluss, dass „die Berechnung zu lang dauert, um damit sinnvolle Dinge zu erledigen“. Er stellt auch fest, dass die Version mit „Tseitin-Addierern“ schneller ist, als die Version mit Addierern die durch Espresso generiert wurden.

Einen anderen Versuch unternimmt Jonathan Heusser in seinem Blog [J H13a]. Sein Fokus liegt dabei auf der Lösung eines Bitcoin-Blocks mit unterschiedlichen SAT-Solvern. Im Gegensatz zu Vegard Nossum und Martin Maurer erzeugt er sich die Eingabe für den SAT-Solver jedoch mit Hilfe eines Modelcheckers aus C-Programmcode. Im Anschluss führt er Versuche mit unterschiedlichen SAT-Solvern durch und vergleicht die Laufzeiten. Speziell bei CryptoMiniSat versucht er auch, die Parameter zu optimieren und kann damit die Laufzeit wesentlich reduzieren.

1.3 Ziel dieser Arbeit

Die im vorhergehenden Abschnitt genannten Arbeiten beschränken sich zu einem Großteil darauf, eine Eingabe für einen SAT-Solver zu generieren und Versuche damit durchzuführen um die Laufzeiten zu ermitteln und zu vergleichen. Vegard Nossum geht noch einen Schritt weiter und versucht das Wissen, das ein SAT-Solver während eines Lösungsversuchs erwirbt, zu extrahieren und für weitere Lösungsversuche zu nutzen. Dieser Schritt erfolgt jedoch relativ kurz am Ende seiner Arbeit und führt zum Ergebnis, dass dieses Wissen in weiteren Lösungsversuchen keinen Vorteil mit sich bringt. An diesem Punkt setzt diese Masterarbeit an um die Analyse des erworbenen Wissens zu erweitern und möglicherweise doch einen Vorteil daraus zu ziehen. Dazu wird ein Hash-Algorithmus herangezogen und abschließend geprüft, ob die Berechnung schnell genug erfolgt, um praktisch relevante Angriffe durchzuführen.

1.4 Vorgehensweise

Als Hash-Algorithmus wird SHA-256 (siehe Kapitel 3) ausgewählt. Dieser wird bei Bitcoin eingesetzt und zeichnet sich dadurch aus, dass er einen Fingerabdruck in 64 ähnlichen Schritten berechnet. Wissen das über einen Schritt erworben wird, kann so unter Umständen für die anderen 63 Schritte übernommen werden. Außerdem haben Martin Maurer und Jonathan Heusser SHA-256 in ihren Arbeiten verwendet, so dass abschließend ein Vergleich der Ergebnisse erfolgen kann. Der von Vegard Nossum verwendete Hash-Algorithmus SHA1 berechnet einen Fingerabdruck in 80 Schritten, wobei vier unterschiedliche Schritte jeweils 20 Mal zum Einsatz kommen.

Verwendet wird in dieser Arbeit ausschließlich der SAT-Solver CryptoMiniSat. Dieser zeichnet sich dadurch aus, dass er nicht nur eine konjunktive Normalform als Eingabe akzeptiert, sondern auch in der Lage ist Klauseln im XOR-Format zu verarbeiten. Eine Klausel ist dabei eine Verknüpfung von Werten mit dem OR-Operator. Klauseln im XOR-Format verwenden den XOR-Operator. Dies ermöglicht eine kompaktere Repräsentation der Eingabe und beschleunigt bei richtiger Verwendung den Lösungsprozess des SAT-Solvers. Außerdem sammelt CryptoMiniSat nicht nur zusätzliches Wissen, sondern versucht auch die eingegebene konjunktive Normalform zu optimieren. Sowohl die optimierte Eingabe, als auch das zusätzliche Wissen lassen sich jederzeit einfach abrufen und analysieren.

Um eine Analyse der Daten durchzuführen, die CryptoMiniSat während eines Lösungsversuchs sammelt, reicht es nicht aus SHA-256 in eine konjunktive Normalform zu überführen und an CryptoMiniSat zu übergeben. Dieser Ansatz wird von Martin Maurer und Jonathan Heusser verfolgt. Jedoch ermöglicht er es nicht, von CryptoMiniSat erworbenes Wissen über die konjunktive Normalform den entsprechenden Teilen von SHA-256 zuzuordnen. Für diesen Prozess müssen während der Erzeugung der konjunktiven Normalform Daten darüber gesammelt werden, die eine Zuordnung ermöglichen. Zu diesem Zweck wird ein Framework erstellt, das je nach Bedarf die konjunktive Normalform erzeugt und gleichzeitig die dazugehörigen Daten

zur Verfügung stellt. Mit diesem Framework wird im Anschluss eine konjunktiven Normalform erzeugt, die mit denen von Martin Maurer und Jonathan Heusser verglichen wird, um diese Basis zu überprüfen und im Zweifelsfall noch weiter zu optimieren.

Mit der erstellten konjunktiven Normalform werden iterativ mehrere Lösungsversuche durchgeführt. Die dafür verwendeten Daten werden separat als „Annahmen" an CryptoMiniSat übergeben. Das ermöglicht eine Trennung zwischen dem Wissen, was auf den Daten basiert, und dem Wissen was ausschließlich auf der konjunktiven Normalform basiert. Nur Letzteres ist für diese Arbeit relevant, da dieses Wissen für SHA-256 im Allgemeinen gilt. Für die Auswertung dieses Wissens werden einige Hilfsprogramme erstellt, die potentiell interessante Klauseln filtern und, basierend auf den gesammelten Informationen, sortieren. Diese zusätzlichen Klauseln werden im Anschluss darauf geprüft, ob sie den Lösungsprozess beschleunigen.

Abschließend wird die konjunktive Normalform inklusive der zusätzlichen Klauseln mit der Performance der Implementierungen von Martin Maurer und Jonathan Heusser verglichen. Außerdem wird überprüft, ob die Berechnung schnell genug erfolgt um praktisch relevante Angriffe auf SHA-256 und Bitcoin durchzuführen.

1.5 Struktur

Im Anschluss an die Einleitung folgt in **Kapitel 2** eine kurze Erläuterung der Grundlagen und Hilfsmittel die in dieser Arbeit verwendet werden.

Es folgt in **Kapitel 3** eine Beschreibung des Hash-Algorithmus SHA-256, um die relevanten Komponenten für die Erzeugung der konjunktiven Normalform in **Kapitel 4** kennen zu lernen.

Kapitel 5 beschreibt die Analyse mit CryptoMiniSat und fasst die erhaltenen Klauseln zusammen die in **Kapitel 6** einer Bewertung unterzogen werden.

Kapitel 7 beinhaltet die Evaluation, in der das Ergebnis dieser Arbeit mit den Arbeiten von Martin Maurer und Jonathan Heusser verglichen wird, und eine Aussage darüber, ob praktisch relevante Angriffe möglich sind.

Kapitel 8 enthält die persönliche Meinung des Autors.

Kapitel 2

Grundlagen

In diesem Kapitel geht es darum, die wichtigsten Grundlagen vorzustellen. Dabei werden jeweils nur die Teile erläutert, die für diese Arbeit relevant sind. Zu den Grundlagen gehören neben Addierern und der konjunktiven Normalform auch die in dieser Arbeit genutzten Programme und die digitale Währung Bitcoin.

2.1 Addierer

Die modulare Addition ist eine surjektive Abbildung. Die Berechnung der Summe von zwei Zahlen ist eindeutig. Die Umkehrung ist jedoch nicht eindeutig, da sich die Summe einer n-Bit Zahl aus 2^n unterschiedlichen Kombinationen von Summanden bilden lässt. Wird die Assoziativität der Addition berücksichtigt, sind es nur $2^n/2$ Kombinationen, was für diese Arbeit jedoch keine Rolle spielt. Durch diese Eigenschaft eignet sich die Addition gut für die Verwendung in Verschlüsselungs- oder Hash-Algorithmen, da einem Angreifer mit Kenntnis der Summe keine Kenntnisse über die Summanden vorliegen.

Die modulare 32-Bit Addition ist die am häufigsten verwendete Operation in der Kompressionsfunktion von SHA-256 (Kapitel 3). Im Gegensatz zu den anderen verwendeten Operationen ist die Addition nicht direkt durch grundlegende binäre Operationen wie AND, OR und XOR definiert. Die Art der Realisierung mit Hilfe der binären Operationen bleibt dem Programmierer überlassen. Um eine Auswahl zu treffen, werden einige Konstruktionen kurz erläutert.

Der Carry-Ripple Addierer
: realisiert die Addition ähnlich der schriftlichen Addition. Jede Stelle der Binärzahl wird einzeln addiert wobei der Übertrag bei der nächsten Addition berücksichtigt wird. Durch dieses Vorgehen ist eine Parallelisierung der einzelnen Additionen nicht möglich, da jede Addition vom Übertrag der vorhergehenden Addition abhängig ist.

Der Conditional-Sum Addierer
: verwendet genau wie der Carry-Ripple Addierer Überträge. Jedoch werden alle Teilsummen zwei Mal berechnet. Einmal mit der Annahme, dass ein Übertrag vorliegt und einmal ohne. Dadurch können

die einzelnen Additionen parallel durchgeführt werden. Abschließend werden die Teilsummen mit den jeweils korrekten Annahmen zusammengeführt.

Der Carry-Lookahead Addierer

vermeidet die Abhängigkeit vom Übertrag indem die vorhergehenden Stellen direkt betrachtet werden. Liegt in jeder vorhergehenden Stelle der Summanden eine 1 vor (XOR) kann ein generierter Übertrag bis zur aktuellen Addition propagiert werden. Generiert wird ein Übertrag entweder durch eine direkte Eingabe oder das Vorliegen von zwei 1en (AND) an einer Stelle der Summanden. Die Betrachtung der vorhergehenden Stellen und die einzelnen Additionen können dabei parallel durchgeführt werden.

Interessant sind die unterschiedlichen Konstruktionen besonders für die Umsetzung in Hardware. Dort kann eine Beschleunigung der Berechnung durch Parallelisierung ausgenutzt werden. Während in der Hardware Parallelisierung bedeutet, eine Berechnung mehrfach auf dem Chip unterzubringen, müssen bei einer Realisierung in Software entsprechend viele Recheneinheiten vorhanden sein. Da es in dieser Arbeit darum geht, Additionen in Software zu berechnen, wird der Carry-Ripple Addierer verwendet. Durch die explizite Berechnung und Berücksichtigung der Überträge bietet dieser auch die Möglichkeit Aussagen über einzelne Überträge zu machen. Gelingt es den Wert 0 eines Übertrags festzulegen, lässt sich daraus folgern, dass die Bits der Summanden vor dem Übertrag keine Auswirkungen auf die Bits der Summe nach dem Übertrag mehr haben. Die Propagierung eines möglichen Übertrags ist unterbrochen. Das Problem, die Summanden einer Addition zu finden, wird dadurch in zwei kleinere Probleme zerlegt, bei der die Summanden von zwei Additionen kleinerer Bitbreite gefunden werden müssen.

Ein allgemeiner n-Bit Carry-Ripple Addierer setzt sich aus n Volladdierern zusammen, die jeweils wieder zwei Halbaddierer verwenden. Ein Halbaddierer summiert zwei Bits (a, b) und gibt die Summe (s) und den Übertrag (o) aus. Die Summe wird dabei durch den XOR-Operator berechnet während der Übertrag durch den AND-Operator berechnet wird. Der Halbaddierer ist in Abbildung 2.1 dargestellt.

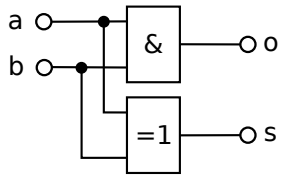


Abbildung 2.1 Halbaddierer[1]

Im Unterschied zum Halbaddierer berücksichtigt der Volladdierer einen eingehenden Übertrag (c). Es werden insgesamt drei Bits summiert, wobei in einem ersten Schritt die beiden Summanden (a, b) durch einen Halbaddierer addiert werden. Führt diese Addition schon zu einem Übertrag, ist dieser im Volladdierer ebenfalls geben (o). Die Summe des ersten Halbaddierers wird im Anschluss durch den zweiten Halbaddierer mit dem eingehenden Übertrag (c) addiert. Ist ein Übertrag gegeben und die Summe 1, liegt ebenfalls

[1] Auf Basis von MovGPo, CC BY-SA 2.0 de, *https://commons.wikimedia.org/w/index.php?curid=22912775*

ein Übertrag im Volladdierer vor (o). Die Summe des zweiten Halbaddierers bildet auch die Summe des Volladdierers (s). Der Volladdierer ist in Abbildung 2.2 dargestellt.

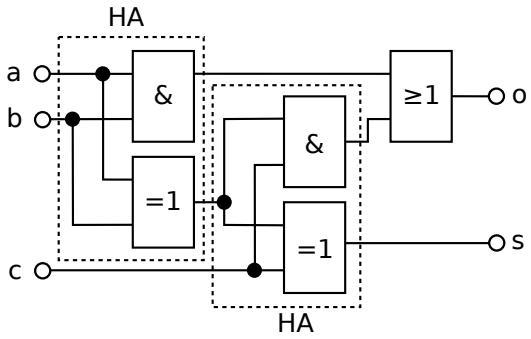


Abbildung 2.2 Volladdierer[2]

Abbildung 2.3 zeigt einen allgemeinen 4-Bit Carry-Ripple Addierer, der aus vier Volladdierern besteht. Dieser berücksichtigt einen eingehenden Übertrag (carry) und liefert auch einen ausgehenden Übertrag. Diese Überträge sind für die in dieser Arbeit benötigten modularen 32-Bit Addierer jedoch nicht notwendig.

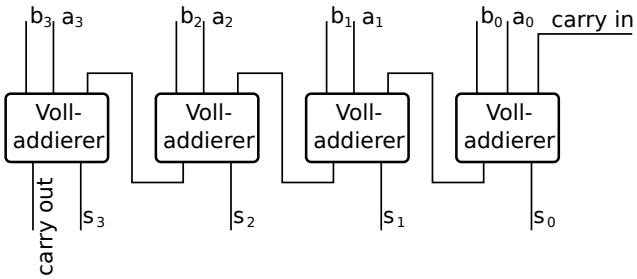


Abbildung 2.3 4-Bit Carry-Ripple Addierer[3]

Deshalb wird der erste Volladdierer durch einen Halbaddierer, und der letzte Volladdierer durch einen Mod2-Addierer ersetzt. Diese Konstruktion ist in in Abbildung 2.4 dargestellt.

[2] Auf Basis von MovGPo, CC BY-SA 2.0 de, *https://commons.wikimedia.org/w/index.php?curid=22912742*
[3] Von Mik81, CCo, *https://commons.wikimedia.org/w/index.php?curid=3393072*

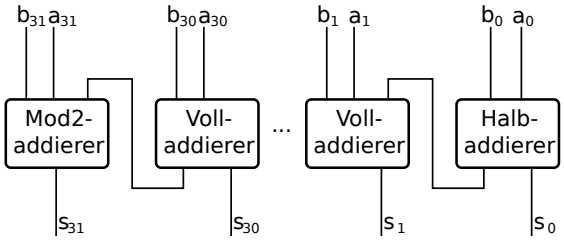


Abbildung 2.4 32-Bit Carry-Ripple Addierer[4]

Der Mod2-Addierer ist in Abbildung 2.5 dargestellt und ergibt sich aus einem Volladdierer, indem die für den Übertrag notwendigen Operatoren entfernt werden. Es verbleiben 2 XOR-Operatoren die die Summe (s) berechnen.

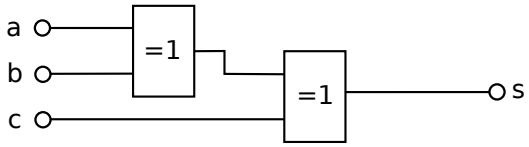


Abbildung 2.5 Mod-2 Addierer[5]

2.2 Konjunktive Normalform

In der boolesche Algebra liegt eine Formel in einer konjunktiven Normalform vor wenn es sich um eine Konjunktion von Klauseln handelt. Jede Klausel ist dabei eine Disjunktion von Literalen. Literale wiederum sind positive oder negative Boolesche Variablen. Diese Form ist in Abbildung 2.6 dargestellt.

$$\bigwedge_i \bigvee_j (\neg) x_{ij}$$

Abbildung 2.6 Formelle Darstellung einer konjunktiven Normalform

Ein Beispiel für eine konjunktive Normalform ist: $(\neg a \vee \neg b \vee \neg c \vee d) \wedge (\neg a \vee c)$. Jedes aussagenlogische Problem lässt sich in eine konjunktive Normalform überführen. Der einfachste Weg geht dabei über eine vollständige Wahrheitstabelle, aus der alle Einträge herangezogen werden, die keine gültige Lösung darstellen. Eine Klausel lässt sich somit als eine Reihe von Literalen betrachten, die in dieser Kombination keine

[4] Auf Basis von Mik81, CCo, *https://commons.wikimedia.org/w/index.php?curid=3393072*
[5] Auf Basis von MovGPo, CC BY-SA 2.0 de, *https://commons.wikimedia.org/w/index.php?curid=22912742*

gültige Belegung besitzen. Durch Konjunktion der Klauseln ist es somit möglich ein Problem zu definieren, in dem alle ungültigen Belegungen als Klauseln ergänzt werden. Um aus einer Gleichung eine Wahrheitstabelle zu erstellen, wird in dieser Arbeit das Programm „eqntott“ verwendet.

Dieses Vorgehen führt jedoch zu einer kanonisch konjunktiven Normalform die aus Klauseln besteht, in der jedes Literal genau einmal vorkommt. Bei komplexen Schaltkreisen mit vielen Ein- und Ausgängen wächst die Klauselmenge dadurch exponentiell. Um diesem entgegen zu wirken kann mit Hilfe der Tseitin-Transformation [Wik16] eine erfüllbarkeitsequivalente konjunktive Normalform erzeugt werden, indem zusätzliche Variablen eingeführt werden. Auf einen Schaltkreis angewandt folgt daraus, dass den Leitungen zwischen den Gattern Variablen zugewiesen werden. Mit Hilfe dieser Variablen kann jedes Gatter separat in eine konjunktive Normalform überführt werden. Diese werden dann schließlich aneinander gehängt und bilden die erfüllbarkeitsequivalente konjunktive Normalform.

Um eine konjunktive Normalform abzuspeichern sowie zur Lösung und Optimierung an weitere Programme zu übergeben, wird in dieser Arbeit das DIMACS-Dateiformat verwendet. Die im Beispiel genannte konjunktive Normalform ist im DIMACS-Format in Abbildung 2.7 dargestellt.

```
p cnf 4 2
-1 -2 -3 4 0
-1 3 0
```

Abbildung 2.7 Datei im DIMACS-Format

In der Kopfzeile wird zunächst die Anzahl der Variablen und Klauseln genannt. Jede Zeile repräsentiert eine Klausel. Jedes Literal bekommt eine Nummer, wobei Null das Ende einer Klausel kennzeichnet. Für die Negation wird ein Minus verwendet.

2.3 Espresso

Espresso ist ein heuristischer Logikminimierer der Berkeley Universität von 1989. Verwendet wird in dieser Arbeit eine moderne Version von 2012 [chm12]. Als Eingabe dienen Wahrheitstabellen wie sie unter anderem mit „eqntott“ (siehe Abschnitt 2.2) generiert werden. Die Ausgabe ist eine Wahrheitstabelle in minimierter Form.

Als Parameter werden „-Dexact“ und „-epos“ verwendet. -Dexact garantiert eine minimale Anzahl von Einträgen in der Wahrheitstabelle, und somit von Klauseln in der konjunktiven Normalform. Die Laufzeit ist dabei exponentiell im Bezug zur Anzahl der Variablen. Da Espresso in dieser Arbeit nur für Wahrheitstabellen mit bis zu 14 Variablen verwendet wird, sind Lösungen noch innerhalb eines Tages möglich. -epos ist notwendig, um die Menge der wahren (On-Set) und nicht wahren Zeilen (Off-Set) der Wahrheitstabelle zu invertieren. Ohne die Angabe des Parameters minimiert Espresso die Menge der wahren Zeilen. Benötigt wird für die konjunktive Normalform jedoch eine minimale Menge der nicht wahren Zeilen.

2.4 SAT-Solver

SAT-Solver dienen dazu, ein Erfüllbarkeitsproblem der Aussagenlogik zu lösen. Als Eingabe wird eine Formel in konjunktiver Normalform genutzt. Ist die Formel erfüllbar, liefert ein SAT-Solver dafür ein Beispiel. Da die Laufzeit im schlechtesten Fall exponentiell zur Anzahl der Variablen ist, gelingt es jedoch nicht immer eine Lösung zu finden oder die Unerfüllbarkeit zu belegen. Relevant für diese Arbeit ist, dass ein SAT-Solver bei einem Lösungsversuch Konfliktklauseln sammelt. Diese ergeben sich aus einem Versuch mehrere Variablen zu belegen der sich als ungültig herausstellt. Konfliktklauseln ergänzen die ursprüngliche konjunktive Normalform. Anstatt diese zu verwerfen, werden sie in dieser Arbeit exportiert, gefiltert und in weitere Lösungsversuche eingebracht um den Lösungsprozess zu beschleunigen.

Kurze Erläuterung der Funktionsweise

Beachtet werden muss dabei, dass die Konfliktklauseln sich auf eine spezifische Lösung beziehen. Soll ein SAT-Solver zum Beispiel zwei Summanden einer Summe finden, wird er mit den Konfliktklauseln eines erfolgreichen Lösungsversuchs immer wieder zum gleichen Ergebnis kommen, obwohl es unter Umständen viele weitere Lösungen gibt, da diese durch die Konfliktklauseln ausgeschlossen werden. Vermeiden lässt sich dieses Verhalten, wenn ausschließlich die Logik (in diesem Fall die Addition) als Eingabe dient. Spezifische Werte wie der Summand können als Annahmen (Assumptions) in den Lösungsprozess aufgenommen werden. Konfliktklauseln die durch diese Annahmen entstehen werden separat behandelt und bei Programmende verworfen.

In dieser Arbeit wird CryptoMiniSat in der Version 4.5.3 verwendet. CryptoMiniSat bietet den Vorteil auch Klauseln einlesen zu können, deren Literale durch den XOR-Operator verknüpft sind. Dies ermöglicht eine kompaktere Repräsentation der Eingabe, da eine XOR-Klausel mit n Literalen 2^{n-1} normale Klauseln repräsentiert. Ein weiterer Vorteil von CryptoMiniSat ist, dass die ursprüngliche konjunktive Normalform während eines Lösungsversuchs optimiert wird. Dadurch wird der Lösungsprozess beschleunigt und möglicherweise neue Klauseln generiert, die in dieser Arbeit analysiert werden sollen. Diese optimierte konjunktive Normalform kann als redundanzfreie Klauselmenge genau wie die Konfliktklauseln (redundante Klauselmenge) exportiert werden.

2.5 C Bounded Model Checking

C Bounded Model Checker (CBMC) ist ein Model Checker für C und C++ Programme. Verwendung findet CBMC in dieser Arbeit in der Evaluation in Kapitel 7. Um einen Beweis zu führen konvertiert CBMC Programmcode automatisch in eine konjunktive Normalform und übergibt diese an einen SAT-Solver. Relevant sind dafür in dieser Arbeit zwei Funktionen mit denen sich ein Beweis führen lässt. Mit nondet_uint() können ganzzahligen Variablen markiert werden, denen CBMC beliebige Werte zuweisen darf. Innerhalb von assert() können Bedingungen formuliert werden, die CBMC beweisen soll.

Bei dem Beispiel des Addierers (wie in Abschnitt 2.4) könnten die beiden Summanden beliebige Werte annehmen. Um zwei Summanden für die Summe 42 zu ermitteln, könnte die Bedingungen im assert() enthalten sein, dass die Summe immer ungleich 42 sein muss. Das veranlasst CBMC dazu, nach zwei Summanden zu suchen, die zusammen 42 ergeben. Gelingt dies, ist der Beweis fehlgeschlagen und CBMC gibt die gefundenen Summanden als Beispiel aus.

2.6 Bitcoin

Am 01. November 2008 veröffentlichte „Satoshi Nakamoto“ über eine Kryptographie-Mailingliste [Sato8b] ein Paper [Sato8a], in dem er Bitcoin als digitale Währung vorstellte. Obwohl bis heute nicht bekannt ist, wer sich hinter diesem Namen verbirgt, hat sich Bitcoin seit der Lösung des ersten „Blocks“ am 03. Januar 2009 [Bito9] weit verbreitet. Die Lösung eines Blocks ist der einzige Aspekt der für diese Arbeit eine Rolle spielt und wird im folgenden erläutert.

Bitcoin basiert auf einer Kette von Blöcken (Blockchain). Ein Block dient dazu, Transaktionen zwischen Benutzern zu bestätigen, die seit der Lösung des letzten Blocks durchgeführt wurden. Da auch der letzte Block in die Lösung mit einbezogen wird, wird die gesamte Kette erneut bestätigt. Für die Lösung eines Blocks wird die Hash-Funktion SHA-256 (siehe Kapitel 3) genutzt. Neben dem letzten Block, den aktuellen Transaktionen und einigen weiteren Daten fließt insbesondere eine Nonce als Eingabe in die Hash-Funktion ein die frei gewählt werden kann um den Block zu lösen. Gelöst ist ein Block dann, wenn durch die Wahl der Nonce und zweimaliger Anwendung von SHA-256 ein Hash erzeugt wird, der mit entsprechend vielen Nullen beginnt (siehe Abbildung 2.8). Je mehr Nullen gefordert sind, desto schwieriger wird diese Aufgabe.

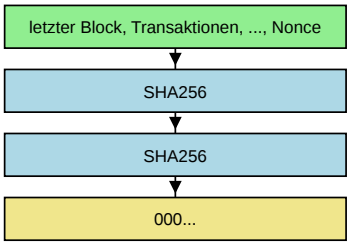


Abbildung 2.8 Schematische Darstellung einer Blockberechnung

Da eine Hash-Funktion eine Einwegfunktion ist, kann eine gültige Nonce nur durch ausprobieren aller möglichen Werte gefunden werden. Als Motivation für diesen Prozess Rechenleistung zu investieren, werden derzeit 12 Bitcoins an den Benutzer ausgeschüttet, der als erster eine gültige Nonce liefert.

Kapitel 3

SHA-256

Secure Hash Algorithm (SHA) steht als Oberbegriff für eine Reihe von Algorithmen die vom National Institute of Standards and Technology (NIST) seit 1993 standardisiert wurden. Diese Algorithmen dienen zur Erzeugung von Daten konstanter Länge aus Daten variabler Länge. Das Ergebnis kann als ein Fingerabdruck der Eingabedaten betrachtet werden und wird im folgenden als Hash bezeichnet. Sowohl die maximal mögliche Eingabelänge als auch die Länge des Fingerabdrucks variieren dabei von Algorithmus zu Algorithmus.

Ein Hash-Algorithmus sollte die drei folgenden Eigenschaften erfüllen um als sicher zu gelten [PP01].

Urbildresistenz

Die Urbildresistenz eines Hash-Algorithmus ist dann gegeben, wenn es sich um eine Einwegfunktion handelt. Zu einem gegebenen Hash soll es nahezu unmöglich sein, eine gültige Eingabe zu berechnen. Häufig wird diese Eigenschaft genutzt, um Passwörter in einer Benutzerdatenbank abzulegen. Bei einer Authentifikation des Benutzers kann der Hash durch Eingabe des Passworts neu berechnet und mit dem hinterlegten Hash verglichen werden, ohne dass ein Angreifer durch einen Datenbankzugriff das Passwort stehlen kann.

Schwache Kollisionsresistenz

Bei der schwachen Kollisionsresistenz soll es nahezu unmöglich sein, zu einer gegebenen Eingabe eine weitere Eingabe zu berechnen, die auf den selben Hash abgebildet wird. Im Gegensatz zur Urbildresistenz ist bei dieser Eigenschaft eine gültige Eingabe bekannt die einem Angreifer Informationen liefern könnte.

Starke Kollisionsresistenz

Während bei der Urbildresistenz ein Hash vorgegeben ist, kann dieser bei der starken Kollisionsresistenz frei gewählt werden. Die starke Kollisionsresistenz ist somit schwerer zu realisieren, da durch die freie Wahl des Hashs das Geburtstagsparadoxon angewendet werden kann. Dabei geht es darum, dass die Wahrscheinlichkeit zwei Personen mit einem gleichen Geburtstag in einem Raum zu finden, wesentlich höher ist, als zu einem gegebenen Geburtstag eine weitere Person zu finden. Konkret bedeutet das, dass bei einem sicheren Hash-Algorithmus mit einer Länge von 256 Bit nur ungefähr 2^{128} anstatt 2^{256} Eingaben geprüft werden müssen um eine Kollision zu finden.

In dieser Arbeit wird SHA-256 betrachtet. Nach SHA-0 und SHA-1 gehört dieser zur dritten Generation, zu der auch SHA-224, SHA-384 und SHA-512 gehören. Genau wie SHA-224 werden maximale Eingaben von $2^{64} - 1$ Bit unterstützt, was $2^{61} - 1$ Byte (vergleiche 1 Gibibyte (GiB) = 2^{30} Byte) entspricht. Der berechnete Fingerabdruck ist gemäß dem Namen 256 Bit lang.

Erstmal veröffentlicht wurde SHA-256 vom NIST im August 2002 [Com+02]. Die aktuellste Version wurde im August 2015 veröffentlicht [PMR15]. Begleitet wurden die Veröffentlichungen des NIST von der Internet Engineering Task Force (IETF). Mit den Request for Comments (RFC) 3174 [EJ01], 4634 [EH06] und 6234 [EH11] verfolgt die IETF das Ziel, den Quellcode zur Berechnung der Hash-Algorithmen der Internet-Gemeinde auf eine einfache Art und Weise zur Verfügung zu stellen.

Die Kompressionsfunktion von SHA-256 besteht aus mehreren Teilen die in den folgenden Abschnitten im Detail erläutert werden. Eine Übersicht ist in Abbildung 3.1 dargestellt. Die Abbildung zeigt die Berechnung durch die Kompressionsfunktion bei einer Eingabe bis zu 447 Bit. Am Anfang stehen acht 32 Bit breite Konstanten die durch die Kompressionsfunktion in 64 Runden (siehe Abschnitt 3.4) unter Berücksichtigung der Eingabe zum Hash entwickelt werden. Die Eingabe wird dafür mit Hilfe des Paddings (siehe Abschnitt 3.1) auf 512 Bit erweitert. 512 Bit entsprechen sechzehn 32 Bit Blöcken, die direkt in den ersten 16 Runden verwendet werden. Für die verbleibenden 48 Runden werden jeweils 4 vorhergehende 32 Bit Blöcke genutzt um die Eingabe zu erweitern (siehe Abschnitt 3.3). Um eine Cryptoanalyse zu erschweren, wird in jeder Runde eine andere Konstante auf die jeweilige Eingabe addiert, so dass der Einfluss der jeweiligen Eingabe auf die Entwicklung der Konstanten ein anderer ist. Abschließend werden die acht Konstanten noch einmal auf das Ergebnis addiert.

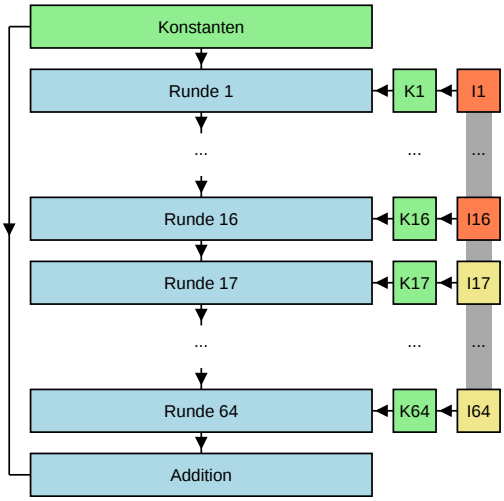


Abbildung 3.1 Schematische Darstellung einer einzelnen SHA-256-Berechnung

Ist die Eingabe länger als 447 Bit, wird sie mit Hilfe des Paddings auf ein Vielfaches von 512 Bit erweitert. Die Kompressionsfunktion wird dann entsprechend oft ausgeführt, wobei die acht 32 Bit breiten Konstanten

nur für den ersten Funktionsaufruf genutzt werden. Das Ergebnis des ersten Aufrufs dient als Eingabe für den nächsten Funktionsaufruf. Dieses Schema ist in Abbildung 3.2 dargestellt.

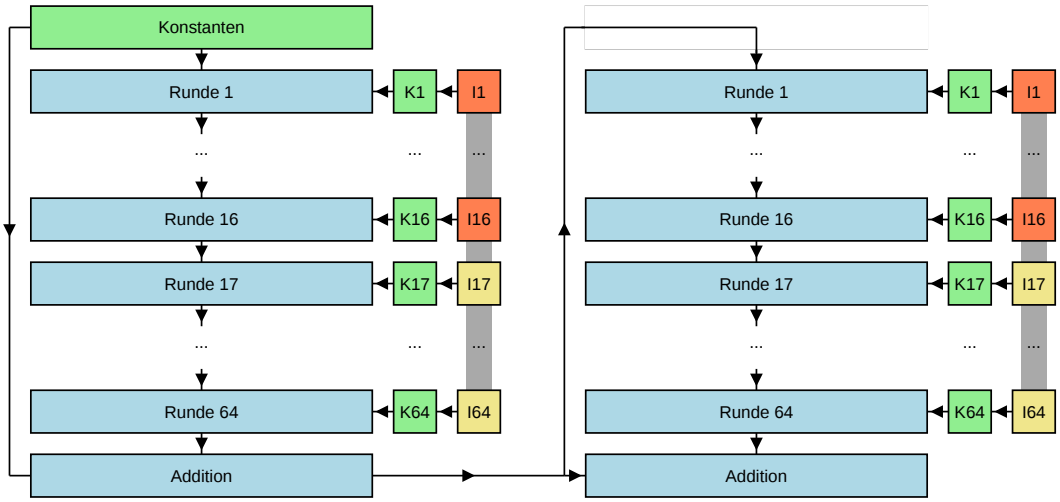


Abbildung 3.2 Schematische Darstellung mehrerer SHA-256-Berechnungen

3.1 Padding

Um die Eingabe auf ein Vielfaches von 512 Bit zu erweitern, wird das Padding benötigt. Dabei wird ein einzelnes Bit mit dem Wert „1“ an die Eingabe der Länge L gehängt. Dann folgen K „0“en und abschließend ein 64 Bit Block der die Länge L in binärer Repräsentation enthält. K ergibt sich dabei durch die Formel (L + K) mod 512 = 447. Hier zeigt sich, dass bei einem K von 0 die Eingabe 447 Bit lang werden kann, bevor die Kompressionsfunktion weiteres Mal ausgeführt werden muss, da ein zweiter 512 Bit Block erzeugt wird.

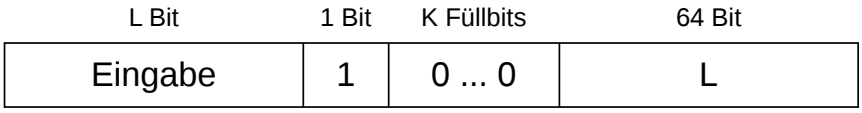


Abbildung 3.3 Schematische Darstellung des Paddings

3.2 Funktionen

Die Funktion, die in SHA-256 am häufigsten verwendet wird, ist die modulare 32-Bit Addition (siehe Abschnitt 2.1). Darüber hinaus werden sowohl bei der Erweiterung der Eingabe als auch bei der Rundenfunktion einige Funktionen verwendet, die im folgendem erläutert werden. Die Funktion „Small Sigma“ wird bei der Erweiterung der Eingabe verwendet, während die Funktionen „Choose“, „Majority“ und „Big Sigma“ in der Rundenfunktion verwendet werden. Alle genannten Funktionen operieren auf einer Datenwortbreite von 32 Bit.

3.2.1 Choose (CH)

Choose steht für Wählen und beschreibt das Verhalten eines Multiplexers. Dabei bestimmt der Wert von x, ob y oder z das Ergebnis der Funktion bildet. Durch die Datenwortbreite von 32 Bit, kommt es somit zu einer bitweisen Vermischung von y und z, die durch die Bits von x bestimmt wird. Dargestellt ist die Funktion in Abbildung 3.4. Gemäß Standard [PMR15, S. 10] führt das Ersetzen des XOR-Operators durch einen OR-Operator zu einem identischen Ergebnis.

$$\mathrm{CH}(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$$

Abbildung 3.4 Choose (CH)

3.2.2 Majority (MAJ)

Majority steht für Mehrheit und bezieht sich auf die Anzahl der „1“ und „0“ Bits. Genau wie bei der Funktion „Choose“ wird das Ergebnis aus drei Eingaben berechnet. Es ergibt sich ein Ergebnis von „1“, wenn mindestens zwei der Eingaben mit „1“ belegt sind. Auch hier führt das Ersetzen des XOR-Operators durch einen OR-Operator zu einem identischen Ergebnis.

$$\mathrm{MAJ}(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

Abbildung 3.5 Majority (MAJ)

3.2.3 Sigma (SIG)

Die Sigma-Funktionen werden in den RFCs der IETF mit SSIG und BSIG beschrieben, das vermutlich aus „Small Sigma“ und „Big Sigma“ abgeleitet wurde um Sonderzeichen zu vermeiden. Die Sigma-Funktionen werden auch als Substitutions-Boxen (S-Boxen) bezeichnet [A A09, S. 1]. Jedes Bit der Ausgabe wir dabei aus zwei bis drei Eingabebits berechnet. Da die Funktion 32 Bit auf 32 Bit abbildet, werden somit alle Eingabebits für zwei bis drei Ausgabebits verwendet. Dargestellt sind alle vier Sigma-Funktionen in Abbildung 3.6.

Innerhalb der Funktionen wird die Rotation nach rechts (ROTR) und die Verschiebung nach rechts (SHR) verwendet. Im Unterschied zur Rotation, bei der die Bits, die aus dem Register geschoben werden, auf der anderen Seite wieder eingefügt werden, werden die Bits bei der Verschiebung mit „0“en aufgefüllt.

Während bei den Σ-Funktionen konsequent drei Eingabebits mit dem XOR-Operator verknüpft werden um ein Ausgabebit zu berechnen, werden bei den σ-Funktionen einige Bits durch die Verschiebung verworfen, was einem Angreifer die Analyse erschwert.

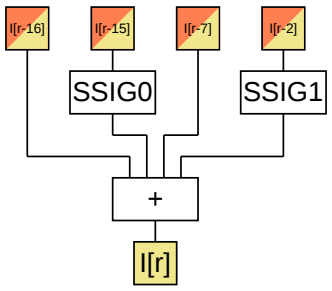
$$\sigma_0(x) = \text{ROTR}^7(x) \oplus \text{ROTR}^{18}(x) \oplus \text{SHR}^3(x)$$
$$\sigma_1(x) = \text{ROTR}^{17}(x) \oplus \text{ROTR}^{19}(x) \oplus \text{SHR}^{10}(x)$$

$$\Sigma_0(x) = \text{ROTR}^2(x) \oplus \text{ROTR}^{13}(x) \oplus \text{ROTR}^{22}(x)$$
$$\Sigma_1(x) = \text{ROTR}^6(x) \oplus \text{ROTR}^{11}(x) \oplus \text{ROTR}^{25}(x)$$

Abbildung 3.6 Sigma (SIG)

3.3 Erweiterung der Eingabe

Um für jede der 64 Runden einen 32 Bit Eingabewert berücksichtigen zu können, wird die Eingabe von 512 Bit auf 2048 Bit erweitert. Dabei kommen sowohl die modulare 32-Bit Addition als auch die beiden σ-Funktionen zum Einsatz. Für die ersten 16 Runden wird die Eingabe direkt übernommen. Für alle weiteren Runden werden vier vorhergehende Eingaben zur Berechnung gemäß Abbildung 3.7 herangezogen. Das führt dazu, dass die Eingabe einer Runde nicht nur bis zu vier nachfolgenden Runden beeinflusst, sondern auch Auswirkungen auf weitere Runden hat, da diese wiederum in nachfolgenden Runden verwendet wird.



$$I[r] = \sigma_1(I[r-2]) + I[r-7] + \sigma_0(I[r-15]) + I[r-16]$$

Abbildung 3.7 Schematische Darstellung der Erweiterung

3.4 Rundenfunktion

Die Rundenfunktion ist in Abbildung 3.8 dargestellt und ähnelt der eines doppelten Feistel-Netzwerks. Es kommen sowohl die modulare 32-Bit Addition, die Choose- und Majority-Funktion, wie auch die beiden Σ-Funktionen zum Einsatz. Bei einem Feistel-Netzwerk wird im allgemeinen ein Teil der Eingabe direkt in die Ausgabe kopiert während der andere Teil unter Verwendung des ersten Teils verschlüsselt wird [PP01, S. 311]. Direkt kopiert werden sowohl A bis C als auch E bis G. A' wird mit Hilfe von A bis C berechnet, während E' mit Hilfe von E bis G berechnet wird. Als Schlüssel fließen D, H und I mit ein.

Initialisiert werden A bis H in der ersten Runde der ersten Kompression mit 8 Konstanten. Für die Konstanten mit 32 Bit Breite wurden laut Standard [PMR15, S. 10] die Quadratwurzel der ersten 8 Primzahlen herangezogen. Die ersten 32 Bit der Dezimalstellen jeder Quadratwurzel ergeben die jeweilige Konstante. Mit Blick auf das Feistel-Netzwerk lassen sich diese Konstanten mit dem Klartext vergleichen, der mit der Eingabe als Schlüssel zum Geheimtext entwickelt wird.

Enthalten ist in Abbildung 3.8 auch die Addition der Rundenkonstante (K+). Für die Rundenkonstanten wurden laut Standard [PMR15, S. 10] die Kubikwurzel der ersten 64 Primzahlen herangezogen. Die ersten 32 Bit der Dezimalstellen jeder Kubikwurzel ergeben die jeweilige Konstante. Die Addition der Konstanten bewirkt, dass der Einfluss der Eingabe sich in jeder Runde unterschiedlich auswirkt.

3.5 Analyse

In dieser Analyse geht es um eine oberflächliche Betrachtung der Rundenfunktion und der Kompressionsfunktion. Ziel ist es ein Verständnis dafür zu entwickeln, was eine Umkehrung der Berechnung verhindert und welche Folgen es hätte, wenn die Berechnungen gelingt.

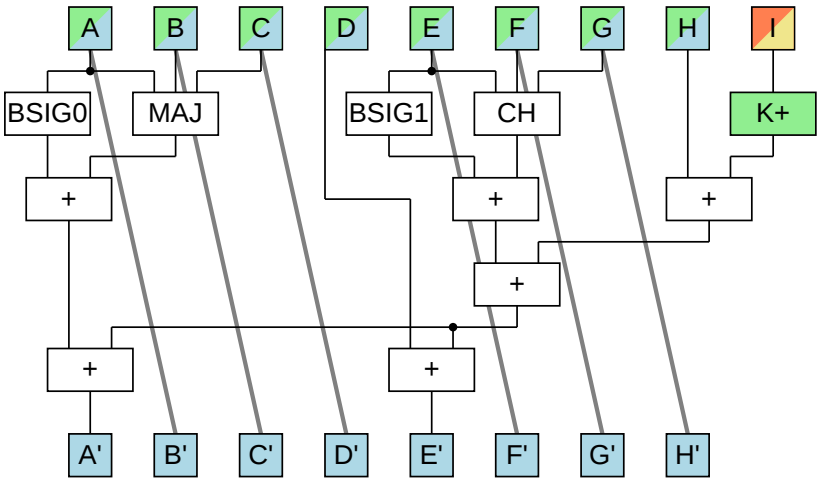


Abbildung 3.8 Schematische Darstellung einer SHA-256-Runde

3.5.1 Rundenfunktion

In Abbildung 3.9 ist noch einmal die Rundenfunktion mit einigen Markierungen dargestellt. Bei dem Versuch, die Funktion umzukehren, ist sofort ersichtlich, dass A bis C und E bis G direkt übernommen werden können.

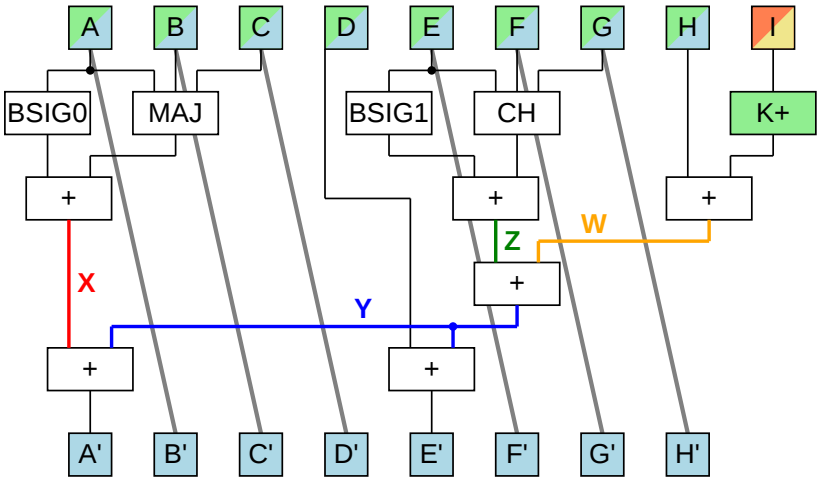


Abbildung 3.9 Analyse einer SHA-256-Runde

Dadurch lassen sich auch **X** und **Z** direkt berechnen. Mit Hilfe von **X** lässt sich schließlich **Y** berechnen, was zu D führt. Formell ist dieser Zusammenhang in Abbildung 3.10 dargestellt.

Schließlich lässt sich auch **W** berechnen (**Y** - **Z**). Ab diesem Punkt ist es jedoch nicht möglich weiter zu rechnen. Da weder H noch I bekannt sind, gibt es 2^{32} Möglichkeiten die Summe **W** zu bilden.

$$\begin{array}{lllll} A' & = & X + Y & \Rightarrow & Y = A' - X \\ E' & = & D + Y & \Rightarrow & Y = E' - D \end{array}$$

$$\begin{array}{lll} E' - D & = & A' - X \\ -D & = & A' - X - E' \\ D & = & X + E' - A' \end{array}$$

Abbildung 3.10 Berechnung von D

3.5.2 Urbildberechnung

Ausgehend davon, dass das Ergebnis von SHA-256 eine Länge von 256 Bit hat, und die Eingabe bei einer einzelnen Anwendung der Kompressionsfunktion bis zu 447 Bit lang werden kann, ist es wahrscheinlich, dass jeder Hash durch eine einzelne Anwendung der Kompressionsfunktion erzeugt werden kann. Um mit vollständigen Bytes zu arbeiten, wird in diesem Angriffsvektor versucht eine Eingabe mit 440 Bit zu finden. Im Mittel gibt es somit für jeden Hash $2^{440-256} = 2^{184}$ mögliche Eingaben die jeweils zu einer Kollisionen führen.

Bei einer einzelnen Anwendung der Kompressionsfunktion werden acht Konstanten zum Hash entwickelt. Da diese bekannt sind und der Hash ebenfalls vorgegeben ist, lässt sich die abschließende Addition berechnen (siehe Abbildung 3.1). Dadurch sind A' bis H' der letzten Runden bekannt. Wie im vorherigen Abschnitt beschrieben, müssen in jeder Runde H's und I's gefunden werden, die zu den vorgegebene Konstanten führen und eine gültige Eingabe bilden. Zu beachten ist, dass die letzten neun Byte durch das Padding (siehe Abschnitt 3.1) vorgegeben sind. Das führt in den Runden 15 und 16 dazu, dass das I vollständig bekannt ist, während es in der Runde 14 teilweise bekannt ist.

Sollte diese Berechnung gelingen, können auch Eingaben beliebiger Länge erzeugt werden. Dazu kann die Kompressionsfunktion beliebig oft mit einer selbst gewählten Eingabe ohne Padding ausgeführt werden. Der daraus resultierende Hash geht als Initialwert in die letzte Runde ein, deren Eingabe wie oben beschrieben berechnet werden könnte.

Eine Urbildberechnung würde es somit ermöglichen, zu hinterlegten Passwort-Hashes gültige Passwörter zu berechnen. Das Fälschen einer Signatur eines längeren Dokuments wäre jedoch noch schwierig, da der letzte Block Eingaben enthalten kann, die im gefälschten Dokument auffallen könnten. Um auch dieses Problem zu lösen, wird der nächste Ansatz betrachtet.

3.5.3 Initialwertberechnung

Wie im vorherigen Abschnitt zu sehen, kann die Urbildberechnung auf eine einzelne Ausführung von SHA-256 zurückgeführt werden. Problematisch ist jedoch, dass das Ende der erzeugten Eingabe nicht unbedingt sinnvoll ist. Um auch dieses Problem zu lösen, müssten die Werte A bis H berechnet werden, wobei der Hash und die Eingabe gegeben sind. I ist somit in jeder Runde bekannt. Da aber die Initalwerte von A bis H fehlen,

ist es nicht mehr möglich, die abschließende Addition zu berechnen, so dass A' bis H' in Runde 64 unbekannt sind. Lediglich die Differenz zwischen den Initialwerten der ersten Runde und A' bis H' der letzten Runde ist bekannt. Somit ist auch in diesem Fall eine einfache Berechnung nicht möglich.

Sollte diese Berechnung gelingen, wäre es möglich, ein Dokument zu erstellen, dass neben einem frei gewählten Inhalt zum Beispiel eine ausgeblendete Grafik oder einen sichtbaren QR-Code enthält. Die Hash-Berechnung könnte dann bis inklusive der einleitenden Bilddaten durchgeführt werden. Die abschließenden Bilddaten und das Dokumentende würden als Eingabe für die letzte Ausführung dienen, für die der Initialwert berechnet wird. Für die vorletzte Ausführung, die die Bilddaten verarbeitet, müsste so eine Urbildberechnung durchgeführt werden, für die keinerlei Anforderungen an die Eingabe gestellt werden braucht.

3.5.4 Kollisionsberechnung

Im Gegensatz zu den Berechnungen in den beiden vorigen Abschnitten hat die Kollisionsberechnung keine praktische Anwendung, da keine Anforderung an den Hash gestellt sind. Ziel ist es lediglich 2 beliebige Eingaben zu finden, die zu einem gleichen Hash führen. Mit Hilfe eines SAT-Solvers (siehe Abschnitt 2.4) lässt dich dieses Problem einfach formulieren. Dazu werden 2 Instanzen des Hash-Algorithmus in den SAT-Solver eingefügt und in einer Miter-Schaltung [D B93] verknüpft. In dieser Schaltung werden die Ausgänge (Hahes) gleichgesetzt und ein Unterschied in mindestens einem Eingabebit gefordert. So kann der SAT-Solver nach einer Lösung suchen ohne dass ein Hash vorgegeben wurde. Das Geburtstagsparadoxon findet somit praktische Anwendung.

Kapitel 4

Erzeugung der konjunktiven Normalform

Der einfachste Weg eine konjunktive Normalform zu erzeugen, die erfüllbarkeitsäquivalent zu SHA-256 ist, führt vermutlich über die Verwendung vom CBMC (siehe Abschnitt 2.5). CBMC erzeugt zur Verifikation von Aussagen über C-Programmcode eine konjunktive Normalform und übergibt diese direkt an einen SAT-Solver (siehe Abschnitt 2.4). Es besteht jedoch auch die Möglichkeit die generierte konjunktive Normalform im DIMACS-Format auszugeben, so dass ein C-Programm automatisch übersetzt werden kann. Dabei gehen jedoch jegliche Informationen über den Aufbau und die Bedeutung einzelner Literale verloren, so dass es nicht möglich ist, erworbenes Wissen über Literale einzelnen Berechnungen zuzuordnen. Ausgehend von der Addition, die in der Kompressionsfunktion von SHA-256 am häufigsten verwendet wird, ist es somit nicht möglich, erworbenes Wissen darüber zuzuordnen und auf alle weiteren Additionen zu übertragen. Außerdem ist es nicht möglich Einfluss auf die Anzahl und Verwendung der Literale zu nehmen. Für einen Addierer liegt die Entscheidung somit bei CBMC, wie dieser realisiert wird und ob und wie viele zusätzliche Literale verwendet werden. Das eine Extrem wäre die Verwendung eines Carry-Ripple-Addierers, dessen einzelne Volladdierer in Gatter zerlegt werden, die dann einzeln in die konjunktive Normalform überführt werden. Das führt zu vergleichsweise vielen Literalen mit wenig kurzen Klauseln. Das andere Extrem wäre der Versuch, eine konjunktive Normalform zu erzeugen, die ausschließlich Literale für die Summanden und die Summe erzeugt. Dabei entstehen jedoch vergleichsweise viele lange Klauseln.

Um sowohl Kontrolle über die Erzeugung der konjunktiven Normalform zu bekommen als auch Informationen zu sammeln um eine Analyse zu ermöglichen, wird ein Programm erstellt, dass das Entwurfsmuster „Besucher“ [Gam+96, S. 301] verwendet. Besucher sind dabei Instanzen von Klassen, die unterschiedlichste Aufgaben erfüllen können. Eine Aufgabe kann es dabei sein, die konjunktive Normalform zu erzeugen während eine andere Aufgabe das Zählen von Literalen und Klauseln sein kann. Besucht wird dabei eine Struktur, deren Objekte das Verhalten von SHA-256 beschreiben. Die Objekte der Struktur werden im Folgenden als Modul bezeichnet. Ein Modul kann sowohl die vollständige Kompressionsfunktion von SHA-256 sein, als auch ein kleiner Baustein wie ein Halbaddierer. Dabei kann ein Modul auch andere Module verwenden. Es zeigt sich, dass acht grundlegende Module ausreichen um SHA-256 vollständig zu beschreiben. Diese werden in den Abschnitten 4.2 bis 4.6 erläutert. Alle weiteren Module setzen sich aus diesen zusammen und werden in Abschnitt 4.7 erläutert.

Alle notwendigen allgemeinen Funktionen für ein Modul sind in einer Basisklasse hinterlegt, von der jedes konkrete Modul erben muss. Ein konkretes Modul wird dadurch realisiert, dass die bis dato rein virtuelle Funktion create(Collector* collector) implementiert wird. Über diese Funktion wird der Besucher in das Modul übergeben, dessen Basisklasse im Folgenden Collector genannt wird, da dieser Informationen über die besuchten Module sammelt. Der Collector stellt zwei virtuelle Funktionen bereit, die ein konkreter Collector implementieren kann.

newModul(unsigned level, const char* name, Modul* modul)
: muss von jedem Modul aufgerufen werden und dient der Registrierung im Collector. Übergeben wird dabei ein selbstgewähltes Level und ein eindeutiger Name. Das Level muss dabei immer höher sein, als alle Level der im Modul benutzen Module, so dass Module eines Levels keine gemeinsamen Literale verwenden. Der Name dient als Identifizierung um eine mehrfache Verwendung eines Moduls zu erkennen und gelerntes Wissen zuordnen zu können. Als weiterer Parameter wird noch ein Pointer auf das Modul selbst übergeben, so dass der Collector bei Bedarf weitere Informationen abrufen kann.

create(bool xOR, const std::vector<CMSat::Lit>& vars)
: kann von einem Modul mehrfach aufgerufen werden, um Klauseln zu generieren. Im Fall von SHA-256 wird diese Funktion nur von den acht grundlegenden Modulen aufgerufen. Alle weiteren Module setzen sich aus diesen zusammen und reichen den Collector lediglich weiter. Der erste Paramter gibt an, ob es sich um eine normale Klausel handelt, deren Literale durch ein OR verknüpft sind, oder ob es sich um eine Klausel handelt deren Literale durch ein XOR verknüpft sind. Ein SAT-Solver wie CryptoMiniSat kann XOR-Klauseln direkt verarbeiten. Für SAT-Solver ohne Unterstützung der XOR-Klauseln ist eine Transformation in OR-Klauseln einfach möglich. Eine Transformation in die andere Richtung würde das Sammeln und Analysieren der OR-Klauseln erfordern und wäre dadurch wesentlich aufwendiger.

Die für die Analyse notwendigen konkreten Collectoren werden in Kapitel 5 erläutert. Jedes Modul wird zunächst in seiner Normalform implementiert. Normalform bedeutet in diesem Fall, dass das Modul aufeinander folgende Literale beginnend mit 0 nutzt, um die konjunktive Normalform zu erzeugen. Dabei kommen zunächst die Eingänge, im Anschluss mögliche zusätzliche Literale und schließlich der Ausgang. Über Funktionen in der Basisklasse kann das Modul bei der Verwendung konfiguriert werden. Jeder Eingang setzt sich aus ein oder mehreren aufeinander folgenden Literalen zusammen. Für jeden Eingang kann das Literal mit dem der Eingang beginnt gesetzt werden. Für zusätzliche Literale und den Ausgang kann ebenfalls das Literal gesetzt werden mit dem begonnen wird.

Als Hilfsmittel für die Generierung der Klauseln wird die Klasse ClauseCreator genutzt. Die Verwendung ist in einem Beispiel in Abbildung 4.1 dargestellt. Initialisiert wird der ClauseCreator durch die Übergabe des Collectors, an den die Klauseln übergeben werden sollen. Im Anschluss werden die Nummern der Literale gesetzt. Dadurch ist es möglich mit diesen Literalen beliebig viele Klauseln zu erzeugen, wobei nur noch die Polarität angegeben werden muss. 0 führt zu einer Negation während das Makro CC_DC ein Don't-Care signalisiert und das Literal in der Klausel unterdrückt wird.

Um die Funktion aller Module sicherzustellen, werden diese mit MinUnit [D S15] getestet. MinUnit ist eine minimale Testumgebung und wurde ursprünglich für den Test von C-Programmen entwickelt. Da es

```
ClauseCreator cc(collector);
cc.setLiterals(4,     0,     1,     2,     3);
cc.printClause(4,     0,     1, CC_DC, CC_DC);
cc.printClause(4,     1,     0,     1,     0);
cc.printClause(4, CC_DC, CC_DC,     1, CC_DC);
```

$\Rightarrow$

$$(\overline{0} \vee 1) \wedge (0 \vee \overline{1} \vee 2 \vee \overline{3}) \wedge (2)$$

Abbildung 4.1 Verwendung des ClauseCreators

unter der MIT-Lizenz veröffentlicht wurde kann es ohne Einschränkungen direkt in den Programmcode eingebunden und angepasst werden. Ausgeliefert wird MinUnit in einer einzigen Header-Datei. Das führt zu einem Problem wenn der Linker mehrere Objekt-Dateien zusammenführt, da MinUnit mehrfach eingebunden wurde. Um dieses Problem zu vermeiden, wird der Programmcode von MinUnit in eine Header- und eine Quelltext-Datei aufgeteilt.

Für die Erzeugung einer potentiell minimalen konjunktiven Normalform wird im ersten Schritt das Programm eqntott verwendet. Mit Hilfe von eqntott ist es möglich, Boolesche Gleichungen in eine Wahrheitstabelle zu überführen. Da das Eingabeformat weder den XOR- noch den Gleichheits-Operator unterstützt, müssen diese beiden Operatoren in der Eingabedatei selbst definiert werden. Ein binärer XOR-Operator entspricht dabei einem negierten Gleichheits-Operator. Die Definitionen in Abbildung 4.2 werden bei allen Berechnungen in den nächsten Abschnitten verwendet.

```
#define xor(a,b) ((a)&!(b) | !(a)&(b))
#define eq(a,b) ((a)&(b) | !(a)&!(b))
```

Abbildung 4.2 Gattergleichungen - Kopf

Die ermittelte Wahrheitstabelle kann jedoch nicht direkt in eine konjunktive Normalform überführt werden. Sie enthält alle Einträge die die Boolesche Gleichung erfüllen und ist mit großer Wahrscheinlichkeit nicht minimal. Für die konjunktive Normalform werden alle Einträge benötigt, die die Boolesche Gleichung nicht erfüllen. Um diese zu generieren wäre es möglich, die Boolesche Gleichung entsprechend anzupassen. Am Beispiel eines 4-Bit-Addierers zeigt sich jedoch, dass das Ergebnis wesentlich größer sein kann. Ein 4-Bit-Addierer hat 2^4 mögliche Ergebnisse. Jedes dieser Ergebnisse lässt sich aus 2^4 unterschiedlichen Summanden gewinnen. Es gibt somit $(2^4)^2 = 256$ erfüllende Belegungen. Im Gegensatz dazu stehen bei jedem möglichen Ergebnis $2^4 \cdot (2^4 - 1)$ mögliche Kombinationen von Summanden die nicht zum Ergebnis führen. Die nicht erfüllenden Belegungen belaufen sich somit auf $(2^4)^2 \cdot ((2^4) - 1) = 3840$. Da diese Wahrheitstabelle nur als Zwischenergebnis und Eingabe für Espresso (siehe Abschnitt 2.3) verwendet werden soll, werden deshalb die erfüllenden Belegungen generiert.

Die Wahrheitstabelle wird im zweiten Schritt an Espresso übergeben. Nachdem Espresso die Wahrheitstabelle eingelesen hat führt der Parameter -epos dazu, dass die erfüllenden Belegungen in nicht-erfüllende überführt werden. Diese Belegungen werden von Espresso minimiert. Durch den Parameter -Dexact wird

eine minimale Anzahl an Belegungen garantiert. Das führt jedoch nur bei Problemen mit wenig Variablen zu einer Lösung, hat sich in den vorliegenden Fällen aber bewährt.

Die minimierten nicht-erfüllenden Belegungen gibt Espresso wieder in einer Wahrheitstabelle aus. Diese kann nun einfach in eine konjunktive Normalform überführt werden. Abbildung 4.3 zeigt dafür ein Beispiel. Ist zum Beispiel die Belegung b=1 und c=1 gegeben, führt dies dazu, dass die Klausel $(\overline{b} \vee \overline{c})$ zu 0 evaluiert und die Formel nicht mehr erfüllbar ist, was genau dem gewünschten Verhalten entspricht.

a	b	c	d		
1	0	-	0	$\Rightarrow$	$(\overline{a} \vee b \vee d) \wedge$
-	0	0	1	$\Rightarrow$	$(b \vee c \vee \overline{d}) \wedge$
0	0	0	0	$\Rightarrow$	$(a \vee b \vee c \vee d) \wedge$
-	1	1	-	$\Rightarrow$	$(\overline{b} \vee \overline{c})$

Abbildung 4.3 Wahrheitstabelle -> Konjunktive Normalform

Nachdem das allgemeine Vorgehen erläutert wurde, werden in den folgenden Abschnitten die konkreten Ergebnisse für die einzelnen Kompontenten von SHA-256 vorgestellt.

4.1 Gatter

Grundlage für die Tseitin-Transformation (siehe Abschnitt 2.2) sollen in dieser Arbeit die Operatoren AND, OR und XOR bilden. Verwendet werden sie jedoch nur, falls sie in einem Modul eine direkte Beziehung zwischen Eingangs- und Ausgang-Literalen definieren, da sonst zusätzliche Literale eingefügt werden müssen. Die Boolesche Gleichungen für die drei Operatoren sind in Abbildung 4.4 dargestellt. a und b werden als Literale für die Eingänge verwendet und r steht für den Ausgang (das Resultat).

```
NAME = AND;                      NAME = OR;                       NAME = XOR;
INORDER = r_out a_in b_in;       INORDER = r_out a_in b_in;       INORDER = r_out a_in b_in;
OUTORDER = z;                    OUTORDER = z;                    OUTORDER = z;

z = eq(r_out, (a_in & b_in));    z = eq(r_out, (a_in | b_in));    z = eq(r_out, xor(a_in, b_in));
```

Abbildung 4.4 Gatter - Gleichungen

Nach der Verwendung von eqntott und Espresso ergeben sich aus den Booleschen Gleichungen die konjunktiven Normalformen in Abbildung 4.5. Diese decken sich mit den Angaben auf Wikipedia [Wik16]. Für die Negation der Operatoren reicht es aus, das Ergebnisliteral r zu invertieren.

Während es bei den Operatoren AND und OR gelungen ist, Don't-Care Literale zu identifizieren und für eine Vereinfachung zu nutzen, ist dies bei dem XOR-Operator nicht möglich. Jede einzelne Änderung eines beliebigen Eingangsliterals führt zu einer Änderung des Ausgangssignals. Genau die Hälfte der möglichen

AND	OR	XOR
$(\overline{r} \vee a) \wedge$	$(r \vee \overline{a}) \wedge$	$(\overline{r} \vee \overline{a} \vee \overline{b}) \wedge$
$(\overline{r} \vee b) \wedge$	$(r \vee \overline{b}) \wedge$	$(r \vee a \vee \overline{b}) \wedge$
$(r \vee \overline{a} \vee \overline{b})$	$(\overline{r} \vee a \vee b)$	$(r \vee \overline{a} \vee b) \wedge$
		$(\overline{r} \vee a \vee b)$

Abbildung 4.5 Gatter - Konjunktive Normalformen

Belegungen ist somit nicht erfüllbar und muss in die konjunktive Normalform mit aufgenommen werden. Allgemein gilt, dass bei einem XOR mit n Eingängen 2^n Klauseln mit jeweils $(n + 1)$ Literalen notwendig sind. Die Klauselmenge wächst exponentiell im Bezug zur Anzahl der Eingänge.

CryptoMiniSat (siehe Abschnitt 2.4) ist in der Lage, neben normalen Klauseln (Disjunktion) auch Klauseln zu berücksichtigen, deren Literale mit dem XOR-Operator verknüpft sind. Genau wie normale Klauseln müssen sie für die Erfüllbarkeit zu 1 evaluiert werden. Damit lässt sich das exponentielle Wachstum der Klauselmenge umgehen. Für ein XOR mit n Eingängen ist nur noch eine Klausel mit $(n + 1)$ Literalen notwendig. Abbildung 4.6 zeigt die Klausel für den XOR-Operator mit zwei und drei Eingängen.

Zwei Eingänge:	$(\overline{r} \veebar a \veebar b)$
Drei Eingänge:	$(\overline{r} \veebar a \veebar b \veebar c)$

Abbildung 4.6 Gatter - XOR

Abschließend wird noch ein Operator für die Negation (NOT) benötigt. Um daraus einen Gleichheits-Operator (EQ) zu machen reicht es aus, das Ergebnisliteral r zu invertieren. Realisiert wird die Negation durch eine XOR-Klausel mit zwei Literalen (siehe Abbildung 4.7). Diese wird nur dann zu 1 evaluiert, wenn a und r unterschiedlich sind, was genau der Negation entspricht. Falls XOR-Klauseln nicht unterstützt werden, lässt sich die Negation mit zwei Klauseln à zwei Literalen realisieren, wie ebenfalls in der Abbildung dargestellt. Auch diese konjunktive Normalform deckt sich mit der Angabe auf Wikipedia [Wik16].

Ohne XOR	Mit XOR
$(r \vee a) \wedge$	$(r \veebar a)$
$(\overline{r} \vee \overline{a})$	

Abbildung 4.7 Gatter - NOT - Konjuktive Normalform

4.2 Addierer

Grundlage für die modulare 32-Bit Addition sind der Halbaddierer, der Volladdierer und der Mod-2 Addierer (siehe Abschnitt 2.1). Wie auch für die Gatter im Abschnitt vorher wird für den Halbaddierer eine Boolesche Gleichung erstellt (siehe Abbildung 4.8). Diese entspricht genau den Gattern in Abbildung 2.1.

```
NAME = HalfAdder;
INORDER = o_out s_out a_in b_in;
OUTORDER = z;

z = eq(s_out, xor(a_in, b_in)) & eq(o_out, a_in & b_in);
```

Abbildung 4.8 Halbaddierer - Gleichung

Nach der Verwendung von eqntott und Espresso ergibt sich aus der boolenschen Gleichung die konjunktive Normalform in Abbildung 4.9 (linke Seite). eqntott und Espresso unterstützen jedoch keine XOR-Klauseln, weshalb ein manueller Versuch unternommen wird, diesen Vorteil zu nutzen. Das Ergebnis ist in Abbildung 4.9 (rechte Seite) dargestellt und besteht aus dem AND- und dem XOR-Gatter. Die Klauselmenge kann so von sechs auf vier Klauseln reduziert werden. Diese Lösung bietet sich jedoch nur an, wenn die XOR-Klausel auch direkt verwendet werden kann. Wird sie in normale Klauseln umgewandelt, stehen im Ergebnis sieben Klauseln. Das ist eine mehr als in der Lösung von Espresso.

Ohne XOR	Mit XOR
$(\overline{s} \vee a \vee b) \wedge$	Übertrag - AND
$(\overline{o} \vee \overline{s}) \wedge$	$(\overline{o} \vee a) \wedge$
$(\overline{o} \vee b) \wedge$	$(\overline{o} \vee b) \wedge$
$(o \vee \overline{a} \vee \overline{b}) \wedge$	$(o \vee \overline{a} \vee \overline{b}) \wedge$
$(s \vee a \vee \overline{b}) \wedge$	Summe - XOR
$(s \vee \overline{a} \vee b)$	$(\overline{s} \veebar a \veebar b)$

Abbildung 4.9 Halbaddierer - Konjuktive Normalform

Im Volladdierer (siehe Abbildung 2.2) gibt es keine Gatter, die eine direkte Beziehung zwischen Eingangs- und Ausgangs-Literalen definieren. Für die direkte Realisierung mit Hilfe der Gatter müssten 3 weitere Literale eingefügt werden. Für zwei AND- und ein OR-Gatter ergeben sich neun Klauseln (jeweils drei). Bleiben noch die beiden XOR-Gatter die jeweils zu einer XOR-Klausel oder vier normalen Klauseln führen. Die Anzahl der Literale beträgt somit insgesamt 8 und die Anzahl der Klauseln 17 (oder 9 + 2 XOR).

Bei der Verwendung von eqntott und Espresso zeigt sich, dass es kompaktere Lösungen gibt, die ohne zusätzliche Literale weniger Klauseln benötigen. Die Boolesche Gleichung für den Volladdierer ist in Abbildung 4.10 dargestellt. Wie auch bei dem Halbaddierer werden zwei mögliche Lösungen generiert. Während in der ersten Lösung die vollständige Boolesche Gleichung genutzt wird, fallen in der zweiten Lösung die blau markierten Teile weg, um die XOR-Operationen manuell zu berücksichtigen.

```
NAME = FullAdder;
INORDER = o_out s_out a_in b_in c_in;
OUTORDER = z;

z = eq(s_out, xor(xor(a_in, b_in), c_in)) & eq(o_out, (a_in & b_in) | (xor(a_in, b_in) & c_in));
```

Abbildung 4.10 Volladdierer - Gleichung

Es ergeben sich die konjunktiven Normalformen in Abbildung 4.11. Der eingehende Übertrag wird als c bezeichnet während der berechnete Übertrag o genannt wird. Die Anzahl der Klauseln beläuft sich auf zehn, wobei durch die XOR-Unterstützung eine Reduktion auf 7 Klauseln möglich ist. Die Umwandlung der XOR-Klausel in normale Klauseln würde zu einer Menge von 14 Klauseln führen.

Ohne XOR	Mit XOR
$(s \vee \overline{a} \vee \overline{b} \vee \overline{c}) \wedge$	Übertrag
$(\overline{s} \vee a \vee b \vee c) \wedge$	$(o \vee \overline{a} \vee \overline{b}) \wedge$
$(o \vee \overline{a} \vee \overline{b}) \wedge$	$(\overline{o} \vee a \vee b) \wedge$
$(\overline{o} \vee a \vee b) \wedge$	$(o \vee \overline{a} \vee \overline{c}) \wedge$
$(o \vee s \vee \overline{c}) \wedge$	$(o \vee \overline{b} \vee \overline{c}) \wedge$
$(\overline{o} \vee \overline{s} \vee c) \wedge$	$(\overline{o} \vee a \vee c) \wedge$
$(\overline{s} \vee a \vee \overline{b} \vee \overline{c}) \wedge$	$(\overline{o} \vee b \vee c) \wedge$
$(\overline{s} \vee \overline{a} \vee b \vee \overline{c}) \wedge$	Summe - XOR
$(s \vee a \vee \overline{b} \vee c) \wedge$	$(\overline{s} \veebar a \veebar b \veebar c)$
$(s \vee \overline{a} \vee b \vee c)$	

Abbildung 4.11 Volladdierer - Konjuktive Normalform

Der Mod-2 Addierer braucht keiner weiteren Analyse unterzogen werden, da er ausschließlich aus XOR-Gattern besteht. Die konjuktive Normalform ist in Abbildung 4.12 dargestellt. Es werden entweder acht normale Klauseln oder eine XOR-Klausel benötigt. Eine Lösung mit zwei einzelnen XOR-Gattern würde keine Vorteile bringen, da sie ein zusätzliches Literal benötigt und die Klauselanzahl nicht verringert.

Ohne XOR	Mit XOR
$(\overline{s} \vee a \vee b \vee c)$	$(\overline{s} \veebar a \veebar b \veebar c)$
$(\overline{s} \vee a \vee \overline{b} \vee \overline{c})$	
$(\overline{s} \vee \overline{a} \vee b \vee \overline{c})$	
$(\overline{s} \vee \overline{a} \vee \overline{b} \vee c)$	
$(s \vee \overline{a} \vee \overline{b} \vee \overline{c})$	
$(s \vee \overline{a} \vee b \vee c)$	
$(s \vee a \vee \overline{b} \vee c)$	
$(s \vee a \vee b \vee \overline{c})$	

Abbildung 4.12 Mod-2 Addierer - Konjuktive Normalform

In Tabelle 4.1 sind noch mal alle Ergebnisse aufgelistet. Die erste Zahl steht jeweils für die Anzahl der Literale und die zweite Zahl für die Anzahl der Klauseln. In der letzten Spalte sind die Werte für einen modularen 32-Bit Addierer aufgelistet. Die Anzahl der Klauseln ergibt sich aus einem Halbaddierer, 30 Volladdierern und

einem Mod-2 Addierer. Die Anzahl der Literale wird zunächst auf die gleiche Weise berechnet. Es müssen jedoch noch 31 Literale abgezogen werden, da die Literale für die Überträge doppelt berechnet wurden.

Lösung	Halbaddierer	Volladdierer	Mod-2 Addierer	Gesamt
Gatter	4 - 7	8 - 17	4 - 8	217 - 525
Gatter mit XOR	4 - 4	8 - 11	4 - 1	217 - 335
eqntott & Espresso	4 - 6	5 - 10	4 - 8	127 - 314
XOR	4 - 4	5 - 7	4 - 1	127 - 215
XOR ohne Unterstützung	4 - 7	5 - 14	4 - 8	127 - 435

Tabelle 4.1 Addierer - Literale und Klauseln

Es zeigt sich, dass die Verwendung von eqntott und Espresso, im Vergleich zur Verwendung der Gatter, zu einer kompakteren konjunktiven Normalform führt. Durch die Verwendung von XOR-Klauseln, lässt sich die Klauselmenge weiter reduzieren. Dies sollte jedoch nicht die Basis für eine Nutzung ohne XOR-Unterstützung sein, da in diesem Fall die Klauselmenge größer ist, als die Klauselmenge der von eqntott und Espresso berechneten Version. Es werden daher beide Versionen implementiert, so dass sich je nach Anwendungsfall die bessere Version nutzen lässt.

Ein Anwendungsfall ohne XOR-Unterstützung ist die Generierung von Wahrheitstabellen. Die Komponenten des modularen 32-Bit Addierers wurden bisher einzeln erzeugt und zusammengefügt. Die kompakte Repräsentation ermöglicht es, den 32-Bit Addierer vollständig in einer Wahrheitstabelle darzustellen. Die Wahrheitstabelle hat dabei 127 Spalten mit 314 Zeilen. Diese Wahrheitstabelle dient wieder als Eingabe für Espresso, um weitere Optimierungen zu ermöglichen, die sich aus dem Zusammenhang der einzelnen Komponenten ergeben könnten. Es stellt sich aber heraus, dass Espresso mit 127 Literalen überfordert ist, und auch nach mehr als 24 Stunden zu keiner Lösung kommt.

4.3 Addierer (Konstante)

Für die Addition der Rundenkonstanten wird eine Addition konstruiert, die Literale für die Konstanten vermeidet. Da die Konstanten bekannt sind, können der Halbaddierer, der Volladdierer und der Mod-2 Addierer entsprechend reduziert werden. Abbildung 4.13 zeigt die reduzierten Varianten. In der ersten Zeile der Tabelle sind die Varianten dargestellt, die genutzt werden, falls das eingehende Bit (b) der Konstante den Wert 0 hat. Entsprechend sind in der zweiten Zeile die Varianten für ein Bit (b) mit dem Wert 1 dargestellt.

Ein Halbaddierer der zwei Bits addiert und am Eingang (b) den Wert 0 erhält, gibt den Wert des Eingangs (a) direkt an die Summe (s) weiter. Der Übertrag (o) kann dabei niemals 1 werden. Hat der Eingang (b) den Wert 1, liegt immer genau dann ein Übertrag (o) vor, wenn am Eingang (a) der Wert 1 anliegt. Die Summe (s) hat jedoch nur den Wert 1, falls der Wert 0 am Eingang (a) anliegt. Der Eingang (a) muss invertiert werden.

[1] Jeweils auf Basis von MovGPo, CC BY-SA 2.0 de, *https://commons.wikimedia.org/w/index.php?curid=22912775*

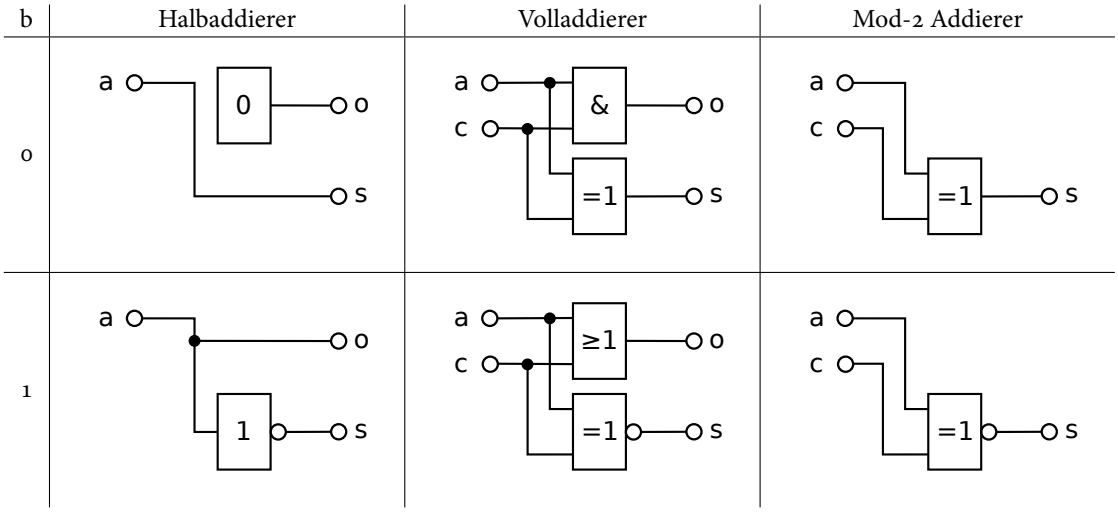


Abbildung 4.13 Reduzierte Addierer[1]

Der Volladdierer addiert drei Bits. Liegt dabei an einem beliebigen Eingang ein Bit mit dem Wert 0 an, brauchen nur noch zwei Bits addiert werden. Der Volladdierer kann auf einen Halbaddierer reduziert werden. Liegt am Eingang (b) der Wert 1 an, ist für einen Übertrag nur noch Vorraussetzung, das Eingang (a) oder Übertrag (c) den Wert 1 haben. Für die Summe (s) liegt durch den Eingang (b) schon ein 1-Bit vor. Die Summe (s) ist nur dann 1, wenn der Eingang(a) und der Übertrag (c) gleich sind, was sich durch den inversen XOR-Operator beschreiben lässt.

Die Mod-2 Addierer werden, wie schon in Abschnitt 2.1 durchgeführt, aus den Volladdierern erzeugt, indem die für den Übertrag notwendigen Operatoren entfernt werden.

Nach der Erstellung der reduzierten Addierer, werden die konjunktiven Normalformen erstellt. Wie auch im vorhergehenden Abschnitt werden zwei mögliche Lösungen generiert, um eine optimale Lösung sowohl mit, als auch ohne XOR-Unterstützung bereit zu stellen. eqntott und Espresso werden dabei nur mit einbezogen, falls der Übertrag und die Summe in Abhängigkeit von den selben Literalen berechnet werden. Im anderen Fall lässt sich die optimale Lösung direkt durch die Verwendung der Gatter generieren.

Die Summe und der Übertrag bei einem Halbaddierer mit eingehendem Bit (b) = 1 werden beide in Abhängigkeit des Eingangs (a) berechnet. Die Boolesche Gleichung in Abbildung 4.14 wird deshalb genutzt um eine optimale konjunktive Normaform zu berechnen. Für den anderen Fall ergibt sich die optimale Lösung direkt aus der Anwendung der Gatter.

```
NAME = HalfAdderConst1;
INORDER = c_out s_out a_in;
OUTORDER = z;

z = eq(c_out, a_in) & eq(s_out, !a_in);
```

Abbildung 4.14 Halbaddierer (1) - Gleichung

Abbildung 4.15 zeigt die ermittelten konjunktiven Normalformen. Unabhängig von dem eigehenden Bit (b) sind bei der Unterstützung von XOR-Klauseln zwei Klauseln notwendig, während es im anderen Fall drei Klauseln sind. Bei Verwendung der Gatter sind die Klauseln entsprechend beschriftet. Eine Umwandlung der XOR-Klauseln in normale Klauseln führt für den Fall des eigenenden Bits (b) = 0 ebenfalls zu drei Klauseln während es im anderen Fall vier Klauseln sind.

b = 0		b = 1	
Ohne XOR	Mit XOR	Ohne XOR	Mit XOR
Übertrag - 0 $(\overline{o}) \wedge$ Summe - EQ $(\overline{s} \vee a) \wedge$ $(s \vee \overline{a})$	Übertrag - 0 $(\overline{o}) \wedge$ Summe - EQ $(\overline{s} \veebar a)$	$(o \vee s) \wedge$ $(\overline{s} \vee \overline{a}) \wedge$ $(\overline{o} \vee a)$	Übertrag - EQ $(\overline{o} \veebar a) \wedge$ Summe - NOT $(s \veebar a)$

Abbildung 4.15 Reduzierter Halbaddierer - Konjuktive Normalform

Ein Volladdierer mit eingehendem Bit (b) = 0 entspricht exakt einem Halbaddierer. Die konjunktive Normalform wird für diesen Fall direkt übernommen. Im anderen Fall dient die Boolesche Gleichung in Abbildung 4.16 zur Ermittlung der konjunktiven Normalform.

```
NAME = FullAdderConst1;
INORDER = c_out s_out a_in c_in;
OUTORDER = z;

z = eq(c_out, a_in | c_in) & eq(!s_out, xor(c_in, a_in));
```

Abbildung 4.16 Volladdierer (1) - Gleichung

Die konjunktiven Normalformen für den Volladdierer sind in Abbildung 4.15 aufgeführt. Wie auch bei dem Halbaddierer ist die Anzahl der Klauseln unabhängig von dem eigehenden Bit (b). Bei einer Unterstützung von XOR-Klauseln sind vier Klauseln notwendig während es ohne Unterstützung sechs Klauseln sind. Eine Umwandlung der XOR-Klauseln führt in beiden Fällen zu einer Klauselanzahl von sieben.

Für den Mod-2 Addierer brauchen eqntott und Espresso nicht herangezogen werden. Dieser berechnet die Summe direkt durch den XOR-Operator. Abbildung 4.18 zeigt die konjunktiven Normalformen für den Mod-2 Addierer. Auch hier hat das eingehende Bit (b) keinen Einfluss auf die Anzahl der Klauseln. Mit

b = 0

Ohne XOR	Mit XOR
$(\overline{s} \vee a \vee c) \wedge$	Übertrag - AND
$(\overline{o} \vee \overline{s}) \wedge$	$(\overline{o} \vee a) \wedge$
$(\overline{o} \vee c) \wedge$	$(\overline{o} \vee c) \wedge$
$(o \vee \overline{a} \vee \overline{c}) \wedge$	$(o \vee \overline{a} \vee \overline{c}) \wedge$
$(s \vee a \vee \overline{c}) \wedge$	Summe - XOR
$(s \vee \overline{a} \vee c)$	$(\overline{s} \veebar a \veebar c)$

b = 1

Ohne XOR	Mit XOR
$(s \vee \overline{a} \vee \overline{c}) \wedge$	Übertrag - OR
$(o \vee s) \wedge$	$(o \vee \overline{a}) \wedge$
$(o \vee \overline{a}) \wedge$	$(o \vee \overline{c}) \wedge$
$(\overline{s} \vee a \vee \overline{c}) \wedge$	$(\overline{o} \vee a \vee c) \wedge$
$(\overline{o} \vee \overline{s} \vee c) \wedge$	Summe - XNOR
$(\overline{o} \vee a \vee c)$	$(s \veebar a \veebar c)$

Abbildung 4.17 Reduzierter Volladdierer - Konjuktive Normalform

Unterstützung von XOR-Klauseln ist eine einzelne Klausel ausreichend während ohne Unterstützung vier Klauseln benötigt werden.

b = 0

Ohne XOR	Mit XOR
Summe - XOR	Summe - XOR
$(\overline{s} \vee \overline{a} \vee \overline{c}) \wedge$	$(\overline{s} \veebar a \veebar c)$
$(s \vee a \vee \overline{c}) \wedge$	
$(s \vee \overline{a} \vee c) \wedge$	
$(\overline{s} \vee a \vee c)$	

b = 1

Ohne XOR	Mit XOR
Summe - XNOR	Summe - XNOR
$(s \vee \overline{a} \vee \overline{c}) \wedge$	$(s \veebar a \veebar c)$
$(\overline{s} \vee a \vee \overline{c}) \wedge$	
$(\overline{s} \vee \overline{a} \vee c) \wedge$	
$(s \vee a \vee c)$	

Abbildung 4.18 Reduzierter Mod-2 Addierer - Konjuktive Normalform

Wie auch für den Addierer im vorhergehenden Abschnitt sind in Tabelle 4.2 noch mal alle Ergebnisse aufgelistet. Die Daten der gatterbasierten Lösungen entsprechen in diesem Fall den Daten der XOR-basierten Lösungen mit und ohne Unterstützung von XOR-Klauseln, da sich alle Überträge und Summen mit Hilfe einer Operation direkt aus den Eingangswerten ergeben. Zusätzliche Literale sind nicht notwendig.

Lösung	**Halbaddierer**	**Volladdierer**	**Mod-2 Addierer**	**Gesamt**
Gatter	3 - 4	4 - 7	3 - 4	95 - 218
Gatter mit XOR	3 - 2	4 - 4	3 - 1	95 - 123
eqntott & Espresso	3 - 3	4 - 6	3 - 4	95 - 187
XOR	3 - 2	4 - 4	3 - 1	95 - 123
XOR ohne Unterstützung	3 - 4	4 - 7	3 - 4	95 - 218

Tabelle 4.2 Reduzierter Addierer - Literale und Klauseln

Für die Addition von Konstanten ist es ebenfalls sinnvoll zwei Lösungen zu implementieren. Bei der Unterstützung von XOR-Klauseln wird die gatterbasierte Lösung verwendet während im anderen Fall die Lösung von eqntott und Espresso verwendet wird.

4.4 Choose (CH)

Die Choose-Funktion kann durch zwei AND-Gatter und ein XOR-Gatter realisiert werden. Als Alternative für das XOR-Gatter wird auch ein OR-Gatter berücksichtigt. Bei Verwendung der Gatter müssen 2 zusätzliche Literale eingefügt werden, die das Ergebnis der beiden AND-Gatter enthalten, so dass insgesamt 6 Literale benötigt werden. Für die AND-Gatter werden jeweils drei Klauseln benötigt. Das XOR-Gatter benötigt eine XOR-Klausel oder vier normale Klauseln während das OR-Gatter drei Klauseln benötigt. Je nach Realisierung ergeben sich damit zehn, neun oder sieben Klauseln.

Nach den Erfolgen in den beiden vorhergehenden Abschnitten wird für die Choose-Funktion ebenfalls eine Boolesche Gleichung erstellt (siehe Abbildung 4.19). Im Gegensatz zu den Addierern ist das Ergebnis des XOR-Gatters nicht direkt von Eingaben abhängig sondern von anderen Gattern. Damit lässt sich das XOR-Gatter nicht ausgliedern und separat betrachten, so dass nur die vollständige Boolesche Gleichung genutzt wird.

```
NAME = CH;
INORDER = r_out a_in b_in c_in;
OUTORDER = z;

z = eq(r_out, xor(a_in & b_in, !a_in & c_in));
```

Abbildung 4.19 Choose - Gleichung

Aus der Berechnung von eqntott und Espresso ergibt sich die konjunktive Normalform in Abbildung 4.20. Vier Klauseln reichen aus um die Choose-Funktion zu beschreiben.

$$
\begin{aligned}
&(r \vee \overline{a} \vee \overline{b}) \wedge \\
&(\overline{r} \vee \overline{a} \vee b) \wedge \\
&(r \vee a \vee \overline{c}) \wedge \\
&(\overline{r} \vee a \vee c)
\end{aligned}
$$

Abbildung 4.20 Choose - Konjuktive Normalform

In Tabelle 4.3 sind die unterschiedlichen Lösungen gegenübergestellt. Die Choose-Funktion wird bitweise auf einer 32 Bit breiten Binärzahl angewendet. Jedes Bit der Eingabe wird dabei nur einmal genutzt. Die einzelnen Berechnungen sind somit unabhängig wodurch sich die Gesamtanzahl der Literale und Klauseln aus der Multiplikation mit 32 ergibt.

Das Ergebnis zeigt, dass es nicht notwendig ist, unterschiedliche Lösungen bereitzustellen. Egal ob eine XOR-Unterstützung gegeben ist oder nicht, ist die von eqntott und Espresso ermittelte Lösung optimal und wird implementiert.

Lösung	Choose	Gesamt
Gatter	6 - 10	192 - 320
Gatter mit OR	6 - 9	192 - 288
Gatter mit XOR	6 - 7	192 - 224
eqntott & Espresso	4 - 4	128 - 128

Tabelle 4.3 Choose - Literale und Klauseln

4.5 Majority (MAJ)

Die Majority-Funktion ist ähnlich aufgebaut wie die Choose-Funktion und enthält ein AND-Gatter mehr. Für die auf den Gattern basierte Lösung sind somit drei zusätzliche Literale notwendig. Anstatt vier werden sieben Literale benötigt. Für die AND-Gatter werden jeweils drei Klauseln benötigt. Für die XOR- bzw. OR-Funktion wird jeweils ein Gatter mit drei Eingängen herangezogen um zumindest hier ein zusätzliches Literals zu vermeiden. Das XOR-Gatter benötigt eine XOR-Klausel oder acht normale Klauseln während das OR-Gatter vier Klauseln benötigt. Je nach Realisierung ergeben sich damit siebzehn, dreizehn oder zehn Klauseln.

Gegenübergestellt wird die Lösung von eqntott und Espresso die sich aus der Booleschen Gleichung in Abbildung 4.21 ergibt.

```
NAME = MAJ;
INORDER = r_out a_in b_in c_in;
OUTORDER = z;

z = eq(r_out, xor(xor(a_in & b_in, a_in & c_in), b_in & c_in));
```

Abbildung 4.21 Majority - Gleichung

Die ermittelte konjunktive Normalform ist in Abbildung 4.22 aufgeführt. Sechs Klauseln reichen aus um die Majority-Funktion zu beschreiben.

$$
\begin{aligned}
&(r \vee \overline{a} \vee \overline{b}) \wedge \\
&(\overline{r} \vee a \vee b) \wedge \\
&(r \vee \overline{a} \vee \overline{c}) \wedge \\
&(r \vee \overline{b} \vee \overline{c}) \wedge \\
&(\overline{r} \vee a \vee c) \wedge \\
&(\overline{r} \vee b \vee c)
\end{aligned}
$$

Abbildung 4.22 Majority - Konjuktive Normalform

In Tabelle 4.4 sind die unterschiedlichen Lösungen gegenübergestellt. Genau wie die Choose-Funktion wird die Majority-Funktion bitweise auf einer 32 Bit breiten Binärzahl angewendet. Jedes Bit der Eingabe wird dabei nur einmal genutzt. Die einzelnen Berechnungen sind somit unabhängig wodurch sich die Gesamtanzahl der Literale und Klauseln aus der Multiplikation mit 32 ergibt.

Lösung	**Majority**	**Gesamt**
Gatter	7 - 17	224 - 544
Gatter mit OR	7 - 13	224 - 416
Gatter mit XOR	7 - 10	224 - 320
eqntott & Espresso	4 - 6	128 - 192

Tabelle 4.4 Majority - Literale und Klauseln

Auch dieses Ergebnis zeigt, dass es nicht notwendig ist, unterschiedliche Lösungen bereitzustellen. Egal ob eine XOR-Unterstützung gegeben ist oder nicht, ist die von eqntott und Espresso ermittelte Lösung optimal und wird implementiert.

4.6 Sigma-Familie (SIG)

Im Gegensatz zu den Funktionen in den vorhergehenden Abschnitten wird bei den Sigma-Funktionen ausschließlich das XOR-Gatter verwendet. Während bisher mehrere Eingaben verrechnet wurden, wird bei den Sigma-Funktionen nur eine einzelne Eingabe verwendet, deren einzelne Bits Verwendung in bis zu drei unterschiedlichen Ausgabebits finden. Da jedes Ausgabebit direkt durch ein XOR-Gatter aus den Eingabebits berechnet wird, sind keine zusätzlichen Literale notwendig. Aus 32 Eingabebits und 32 Ausgabebits ergeben sich 64 Literale.

Bei Verwendung der XOR-Klauseln reicht eine Klausel für jedes Ausgabebit aus, unabhängig davon ob das Ausgabebit aus zwei oder drei Eingangsbits berechnet wird. Anders ist es bei fehlender XOR-Unterstützung. Ein XOR mit drei Eingängen benötigt acht Klauseln während ein XOR mit zwei Eingängen vier Klauseln benötigt. Bei den Σ-Funktionen kommt dieser Unterschied nicht zum tragen, da generell drei Eingabebits für jedes Ausgabebit herangezogen werden. Es ergeben sich 256 Klauseln. Anders verhält es sich bei den σ-Funktionen. Durch die Verschiebung nach rechts werden 3 bzw. 10 Eingangsbits nur zwei mal berücksichtigt. Die entstehende 0 hat keinen Einfluss auf das Ergebnis einer XOR-Operation und kann ignoriert werden. Es ergeben sich deshalb nur 244 bzw. 216 Klauseln. Eine Übersicht ist in Tabelle 4.5 dargestellt.

	Funktion			
Lösung	σ_0	σ_1	Σ_0	Σ_1
Gatter ohne XOR	64 - 244	64 - 216	64 - 256	64 - 256
Gatter mit XOR	64 - 32	64 - 32	64 - 32	64 - 32

Tabelle 4.5 Sigma - Literale und Klauseln

Wie auch für den Addierer, führt der Versuch die vier Funktionen als Wahrheitstabelle auszugeben und mit Espresso zu optimieren zu keinem Ergebnis.

4.7 Übergeordnete Module

Die acht grundlegenden Module aus den vorhergehenden Abschnitten werden in diesem Abschnitt genutzt, um hierarchisch die vollständige Kompressionsfunktion von SHA-256 zu erzeugen. Eine einfache Lösung wäre es, jeweils ein Modul für die Rundenfunktion und für die Erweiterung der Eingabe zu Erstellen. Diese werden im Modul für die Kompressionsfunktion dann 64 bzw. 48 mal eingebunden. Das löst zwar die Generierung der konjunktiven Normalform, ist jedoch für eine Analyse unter Umständen zu grob. Sollte es gelingen, Wissen mit CryptoMiniSat zu erwerben, und einzelnen Modulen zuzuordnen, müsste eine weitere Zuordnung innerhalb des Moduls per Hand erfolgen.

Um dieses Problem zu vermeiden, werden zunächst kleinere Module definiert. Abbildung 4.23 zeigt auf der linken Seite zwei Module die Bestandteil der Rundenfunktion sind und jeweils aus drei anderen Modulen bestehen. Interessant ist dabei, dass A und E jeweils Eingang in zwei Module finden, deren Ergebnisse durch eine Addition wieder zusammengeführt werden. Die Analyse dieser Rekonvergenzen ist besonders interessant und wird durch die beiden Module vereinfacht.

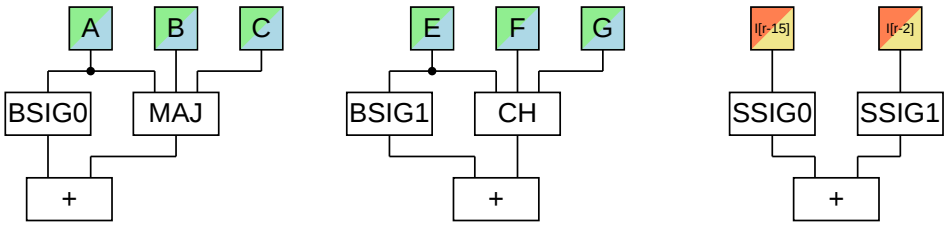


Abbildung 4.23 Erweiterte Module von SHA-256

Ein weiteres Modul ist Bestandteil der Erweiterung der Eingabe und in Abbildung 4.23 auf der rechten Seite dargestellt. Bei der Erweiterung der Eingabe werden vier Werte addiert, wobei zwei davon zunächst durch die σ-Funktionen transformiert werden. Da es keine Rolle spielt, in welcher Reihenfolge die Addition durchgeführt wird, werden diese beiden zunächst in einem eigenen Modul addiert. Da jedes Eingabebit der σ-Funktionen Einfluss auf bis zu drei Ausgabebits hat, könnten sich möglicherweise Rückschlüsse auf die jeweils anderen Eingabebits ziehen lassen. Mit Hilfe dieses Moduls und weiteren Additionen, wird schließlich ein Modul (Prepare) für die vollständige Erweiterung der Eingabe (siehe Abbildung 3.7) erstellt. Die verbleibenden beiden Werte werden zunächst addiert, um das Ergebnis dann mit dem Ergebnis des Moduls zu addieren.

Die beiden erstgenannten Module werden ebenfalls zusammen mit weiteren Additionen genutzt, um ein Modul (ShaCore) für die vollständige Rundenfunktion (siehe Abbildung 3.8) zu realisieren. Außen vor bleibt dabei die Addition der Konstante. Dadurch kann erworbenes Wissen auf alle 64 Runden angewendet werden, da es sich jedes Mal um das gleiche Modul handelt. Eingefügt wird in dieses Modul auch eine weitere Addition und eine Subtraktion um das Wissen aus Abschnitt 3.5.1 mit einzubringen. Da dafür zusätzliche Literale notwendig sind, wird dieses Wissen zunächst außen vor gelassen und nur in Kapitel 6 einer Bewertung unterzogen. Dadurch ist ein besserer Vergleich mit anderen Implementierungen möglich und die Analyse

in Kapitel 5 bezieht sich auf die ursprünglichen Literale und Klauseln, so dass dieses zusätzliche Wissen je nach Bedarf eingebracht oder außen vor gelassen werden kann.

In einem weiteren Modul (ShaCoreEx) wird schließlich die Rundenfunktion mit der Addition der Konstante zusammengeführt. Erworbenes Wissen, was auf der Konstante basiert, kann dadurch in diesem Modul gesammelt werden, falls es sich in einem zweiten Schritt als allgemeingültig erweist.

Abschließend werden die genannten Module in einem Modul (Sha256) für die vollständige Rundenfunktion genutzt. Zusätzlich erfolgt in diesem Modul auch die Addition der anfänglich genutzten Konstanten auf das Ergebnis (siehe Abbildung 3.1). Abbildung 4.24 zeigt den hierarchischen Aufbaue der Module.

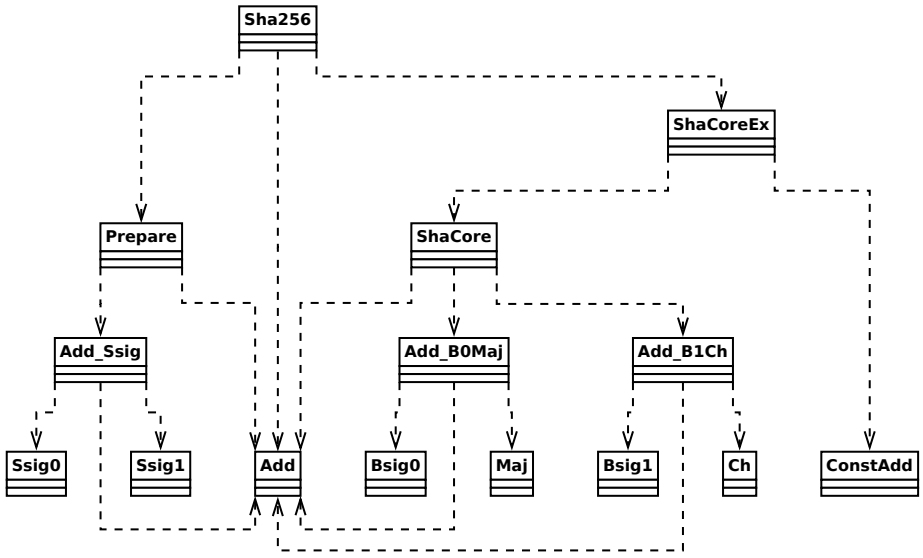


Abbildung 4.24 Modulhierarchie von SHA-256

4.8 Vergleich mit anderen Implementierungen

Für die vollständige Kompressionsfunktion werden bei diesem Vorgehen 49.832 Literale benötigt. Je nach Unterstützung von XOR-Klauseln werden 255.600 bzw. 150.760 Klauseln benötigt, um die Kompressionsfunktion auf diesen Literalen zu definieren. Nicht mit eingerechnet sind dabei Klauseln, die für eine Wertebelegung benötigt werden. Soll ein Bitcoin-Block berechnet werden, muss die Kompressionsfunktion zwei Mal angewendet werden (siehe Abschnitt 2.6). Das führt im Idealfall zu einer Verdopplung der Klauseln und Literale, wobei 256 Literale wegfallen. Diese Literale stehen für das Ergebnis der ersten Anwendung der Kompressionsfunktion und dienen als Eingabe für die zweite Anwendung der Kompressionsfunktion.

Jonathan Heusser hat CBMC genutzt, um die konjunktive Normalform aus einem C-Programm zu generieren. Sein Fokus lag dabei auf der Berechnung eines Bitcoin-Blocks. Da CBMC keine XOR-Klauseln unterstützt, können ausschließlich normale Klauseln zum Vergleich herangezogen werden. Aus seiner Datei satcoin.c [J H13b] wurde eine konjunktive Normalform mit 132.615 Literalen und 648.233 Klauseln generiert. Für die Berechnung einer einzelnen Kompressionsfunktion wird das C-Programm entsprechend angepasst (siehe Anhang B). Daraus ergeben sich 69.356 Literale und 347.128 Klauseln. Hier zeigt sich, dass CBMC für die Berechnung eines Bitcoin-Blocks weniger als das Doppelte einer Kompressionsfunktion notwendig ist. Das liegt daran, dass CBMC die Operationen, deren Eingaben bekannt sind, vor berechnet und nur die unbekannten Teile in die konjunktive Normalform überführt. Anzumerken ist auch, dass diese Klauselmengen bereits Belegungen für konkret Werte enthalten, was jedoch zu vernachlässigen ist.

Martin Maurer hat ein ähnliches Konzept wie das aus dieser Arbeit verwendet und die konjunktive Normalform selbst generiert. Neben der Unterstützung für XOR-Klauseln erzeugt sein Programm [M M13] zwei verschiedene Version der konjunktiven Normalform. Die erste nutzt die Tseitin-Transformation auf Ebene der Gatter, während die zweite Espresso mit einbezieht um zusätzliche Literale in den Addierern zu vermeiden. Die Zahlen zu dieser Lösung sind in Tabelle 4.6 neben den beiden anderen Lösungen dargestellt. Wie auch bei der Realisierung in dieser Arbeit lässt sich die Verdopplung der Literale und Klauseln erkennen, wenn ein Bitcoin-Block berechnet werden soll.

Problem	Realisierung	Literale	Klauseln	Klauseln (XOR)
SHA-256	Diese Arbeit	49832	255600 (5,13 / 3,58)	150760 (3,03 / 3,12)
	CBMC	69356	347128 (5,01 / 3,01)	—
	Maurer - Tseitin	130209	449929 (3,46 / 2,77)	261777 (2,10 / 2,46)
	Maurer - Espresso	60161	665345 (11,06 / 5,60)	590593 (9,82 / 5,87)
Bitcoin	Diese Arbeit	99408	511200 (5,14 / 3,58)	301520 (3,03 / 3,12)
	CBMC	132615	648233 (4,89 / 3,01)	—
	Maurer - Tseitin	260673	901137 (3,46 / 2,77)	524321 (2,01 / 2,46)
	Maurer - Espresso	120577	1331969 (11,05 / 5,60)	1181953 (9,80 / 5,87)

Tabelle 4.6 Vergleich der Anzahl von Literalen und Klauseln

Der Vergleich zeigt, dass das Vorgehen aus dieser Arbeit zu einer kompakteren konjunktiven Normalform führt. Sowohl die Anzahl der Literale als auch der Klauseln sind im Bezug zu den anderen Realisierungen minimal.

Auffällig ist, dass Martin Maurer mit seiner Espresso Variante ebenfalls wenig Literale benötigt, jedoch die Anzahl der Klauseln am höchsten von allen Realisierungen ist. Auf seiner GitHub-Seite [M M13] schreibt er: „In my tests with CMS 3.x the version with TSEITIN ADDERS were faster than version with ESPRESSO.“. Das legt die Vermutung nahe, dass die Anzahl der Klauseln ein wesentlicher Faktor für die Geschwindigkeit ist.

Aufgeführt sind in Tabelle 4.6 für jede Klauselmenge in Klammern zwei weitere Werte. Der erste Wert beschreibt das Verhältnis von Klauseln zu Literalen. In dieser Arbeit werden ohne XOR-Unterstützung bspw.

5,14 Klauseln pro Literal benötigt. Der zweite Wert beschreibt die durchschnittliche Klausellänge. Im genannten Beispiel bezieht sich eine Klausel durchschnittlich auf 3,58 Literale.

Bei der Betrachtung dieser Werte sticht die Tseitin-Variante von Martin Maurer hervor. Die Gatter AND und OR mit zwei Eingängen lassen sich jeweils durch drei Klauseln beschreiben. Zwei Klauseln beziehen sich dabei auf jeweils zwei Literale und eine Klausel auf drei Literale. Ein XOR-Gatter lässt sich mit vier Klauseln à drei Literale beschreiben. Die durchschnittliche Klausellänge liegt damit zwischen zwei und drei und die durchschnittliche Klauselmenge pro Literal zwischen drei und vier. Durch eine XOR-Unterstützung lässt sich die durchschnittliche Klauselmenge auf eins bis drei reduzieren. Diese Werte passen zu den ermittelten Werten in der Tabelle. Im Gegensatz zu seiner Espresso-Variante ist die Realisierung der Tseitin-Variante gelungen.

Kapitel 5

Analyse mit CryptoMiniSat

Die im letzten Kapitel erstellte konjunktive Normalform wird in diesem Kapitel für eine Analyse mit CryptoMiniSat verwendet. Analyse bedeutet in diesem Fall, Lösungsversuche mit CryptoMiniSat durchzuführen. CryptoMiniSat sammelt dabei zusätzliches Wissen in Form von Konfliktklauseln. Außerdem können durch die Optimierung der ursprünglichen konjunktiven Normalform weitere Klauseln generiert werden. Sowohl die Konfliktklauseln als auch die aus der Optimierung entstandenen Klauseln sollen in diesem Kapitel analysiert werden um die Module aus dem vorhergehenden Kapitel mit weiterem Wissen anzureichern und weitere Lösungsversuche zu beschleunigen. Das Vorgehen ist dabei iterativ. Nach einem Lösungsversuch werden mutmaßlich wertvolle Klauseln identifiziert und in die Module aufgenommen, um mit diesem zusätzlichen Wissen einen weiteren Lösungsversuch zu starten.

Als Lösungsversuch wird die Urbildberechnung (siehe Abschnitt 3.5.2) herangezogen. Die Initialwertberechnung (siehe Abschnitt 3.5.3) eignet sich nicht, da die Eingabe bekannt ist und somit die Erweiterung der Eingabe vollständig berechnet werden kann. Dadurch ist dieser Teil bereits gelöst und wird im Lösungsversuch nicht berücksichtigt. Das führt dazu, dass kein Wissen über die Erweiterung der Eingabe erworben werden kann. Für die Urbildberechnung wird der Hash aus Abbildung 5.1 herangezogen. Dieser wurde aus der Eingabe „Das ist eine Eingabe aus der ein Hash erstellt wird.“ mit Hilfe der interaktiven Rubykonsole berechnet und dient unter anderem auch als Testfall für die Kompressionsfunktion. Ziel des Lösungsversuchs ist es, eine Eingabe mit 52 Byte Länge zu finden. Dies könnte auch die Eingabe sein, die benutzt wurde um den Hash zu berechnen.

```
require 'digest'
Digest::SHA256.hexdigest 'Das ist eine Eingabe aus der ein Hash erstellt wird.'
                      => "27931f0e7e53670ddbec1a1ce23e21b4663c63c0d17117ee1a934bc0c294dbe9"
```

Abbildung 5.1 Ruby - SHA-256

Beachtet werden muss auch, dass der Hash nicht direkt mit in die konjunktive Normalform kodiert wird. Gleiches gilt für die Initialwerte und das Padding, was vorgegeben wird um die Eingabelänge zu definieren. Eine direkt Kodierung würde dazu führen, dass erworbenes Wissen sich auf diese Werte bezieht und

somit nur für diesen Fall gültig ist. Um das zu vermeiden werden alle Werte als „Annahmen“ (Assumptions) an CryptoMiniSat übergeben. Das führt dazu, dass die Werte zwar für diesen Lösungsversuch gelten, jedoch nicht als allgemeingültig betrachtet werden. Klauseln die auf diesem Wissen basieren, werden separat behandelt und nach dem Lösungsversuch verworfen (in Anlehnung an „Incremental Satisfiability“ [Sht01, Kapitel 6]).

Ebenfalls berücksichtigt werden muss die Extraktion der gelernten Klauseln aus CryptoMiniSat. Die internen Datenstrukturen sind nicht dafür geeignet, die Klauseln während eines laufenden Lösungsversuchs zu extrahieren. Eine Extraktion kann nur erfolgen, wenn CryptoMiniSat eine Lösung gefunden hat wobei die Lösung auch sein kann, dass es keine Lösung gibt. Wie zu erwarten ist, kommt CryptoMiniSat aber bei vollständiger Vorgabe des Hash zeitnah zu keiner Lösung. Deshalb wird ein iterativer Ansatz verfolgt. Die Initialwerte und das Padding werden immer vollständig vorgeben, während die Vorgabe des Hash bitweise erweitert wird. Nach jeder Lösung werden dabei die Klauseln extrahiert. Es werden so lange Bits des Hash ergänzt, bis keine Lösung mehr gefunden wird. Im Allgemeinen zeigt sich dies dadurch, dass CryptoMiniSat aufgrund eines vollen Hauptspeichers abstürzt oder das Programm selbst die Menge der Klauseln nicht mehr handhaben kann.

Extrahiert werden in jedem Lösungsversuch die irredundanten und die redundanten Klauseln. Die irredundanten Klauseln enthalten zunächst die eingegebene konjunktive Normalform, die jedoch während der Lösungsversuche optimiert wird. Die redundanten Klauseln enthalten das gelernte Wissen, das zusätzlich genutzt wird und wegfallen könnte, ohne die Lösungsmenge zu beeinflussen.

In den folgenden Abschnitten wird die Analyse der extrahierten Klauseln im Detail beschrieben. Zunächst werden in Abschnitt 5.1 alle Klauseln gesammelt und schon bekannte Klauseln aussortiert. In Abschnitt 5.2 wird versucht, die Klauseln einzelnen Modulen zuzuordnen und diese zu normalisieren. Gelingt die Zuordnung zu einem Modul nicht, wird in Abschnitt 5.3 eine Distanzmetrik genutzt, um die Klauseln zu bewerten und mutmaßlich wertvolle Klauseln zu identifizieren. Die Verallgemeinerung von Klauseln über die Breite von 32 Bit und die 64 Runden von SHA-256 wird in Abschnitt 5.4 betrachtet. Durch die Iterationen, die jeweils gelerntes Wissen mit einbeziehen, kann es dazu kommen, dass Don't-Care Literale identifiziert werden und zur Verkürzung von Klauseln führen. Diese Entwicklung wird in Abschnitt 5.5 betrachtet. Abschließend wird das gelernte Wissen in Abschnitt 5.6 zusammengefasst.

5.1 Aussortieren bekannter und doppelter Klauseln

Da die Entscheidung, ob eine XOR-Unterstützung vorliegt, beim Kompilieren getroffen werden muss, werden die bekannten Klauseln mit dem Programm „dimacsprinter“ nach jedem Lösungsversuch und der darauf folgenden Analyse in zwei Dateien im DIMACS-Format ausgegeben. Die eine Datei enthält dabei die Klauselmenge ohne XOR-Unterstützung während die andere Datei die Klauselmenge für eine XOR-Unterstützung enthält. Das Programm „clausecollector“ liest beide Dateien beim Start ein und führt die

Klauselmengen zusammen. Da CryptoMiniSat XOR-Klauseln akzeptiert, diese jedoch intern in normale Klauseln umrechnet, enthalten auch die extrahierten Klauseln keine XOR-Klauseln. Der Parser für das DIMACS-Format konvertiert deshalb ebenfalls XOR-Klauseln in normale Klauseln.

Abgelegt werden die bekannten Klauseln in einem „set“. Diese Datenstruktur ermöglicht die Suche nach einer Klausel in logarithmischer Zeit im Bezug zur Anzahl der darin enthaltenen Klauseln. So kann effizient geprüft werden, ob eine extrahierte Klausel schon bekannt ist. Neue Klauseln aus der irredundanten und der redundanten Klauselmenge werden ebenfalls jeweils in einem „set“ gesammelt wodurch die Klauselmenge automatisch sortiert wird und doppelte Klauseln wegfallen. Abschließend werden die neuen Klauseln aus der irredundanten und der redundanten Klauselmenge jeweils in eine DIMACS-Datei zur weiteren Analyse geschrieben.

5.2 Erkennen und normalisieren modulspezifischer Klauseln

Nach dem Aussortieren der schon bekannten Klauseln werden die neuen Klauseln den einzelnen Modulen zugeordnet. Durch die hierarchische Anordnung muss beachtet werden, dass eine Klausel mehreren Modulen zugeordnet werden kann. Sinnvoll ist es jedoch, die Klausel nur dem Modul zuzuordnen das in der Hierarchie möglichst weit unten steht. Wird eine Klausel für einen Addierer gefunden, kann diese Klausel so zukünftig für alle Addierer genutzt werden, während sie im Modul für die vollständige Kompressionsfunktion nur für den einen speziellen Addierer gelten würde. Um diese Zuordnung zu realisieren erhält jedes Modul ein Level. Im hierarchischen Aufbau muss das Level eines Moduls dabei höher sein, als das höchste Level der verwendeten Module. Die vergebenen Level sind in Abbildung 5.2 dargestellt.

Die Level werden nicht fortlaufend nummeriert, um bei Bedarf weitere Module/Level einfügen zu können, ohne die Level der vorhandenen Module anpassen zu müssen.

Anhand der Level können die Module sortiert werden, so dass die Prüfung, ob eine Klausel zum Modul gehört, bei den Modulen mit dem niedrigsten Level beginnen kann. Eine Klausel gehört dann zu einem Modul, wenn alle Literale der Klausel im Modul verwendet werden. Dabei kann es sich um Eingänge, zusätzliche Literale oder Ausgänge handeln. Ein Sonderfall ist eine Klausel die ausschließlich aus Eingangsliteralen oder ausschließlich aus Ausgangsliteralen besteht. Die Ausgangsliterale eines Moduls sind im allgemeinen die Eingangsliterale eines anderen Moduls wodurch die Zuordnung bei gleichem Level unklar ist. Klauseln dieser Art wurden bei der Analyse jedoch nicht gefunden.

Für die Zuordnung der Klauseln wird zunächst ein Collector mit dem Namen „ModulDB“ implementiert. Die ModulDB überschreibt nur die Methode newModul des Collectors, um die Registrierung der Module zu erfassen. Generierte Klauseln, die an die Methode create übergeben werden, werden dadurch ignoriert. Erfasst werden neben dem Modulnamen und dem Level die verwendeten Literale in der Reihenfolge: Eingänge, zusätzliche Literale und Ausgänge. ModulDB stellt außerdem eine Funktion bereit, der eine beliebige

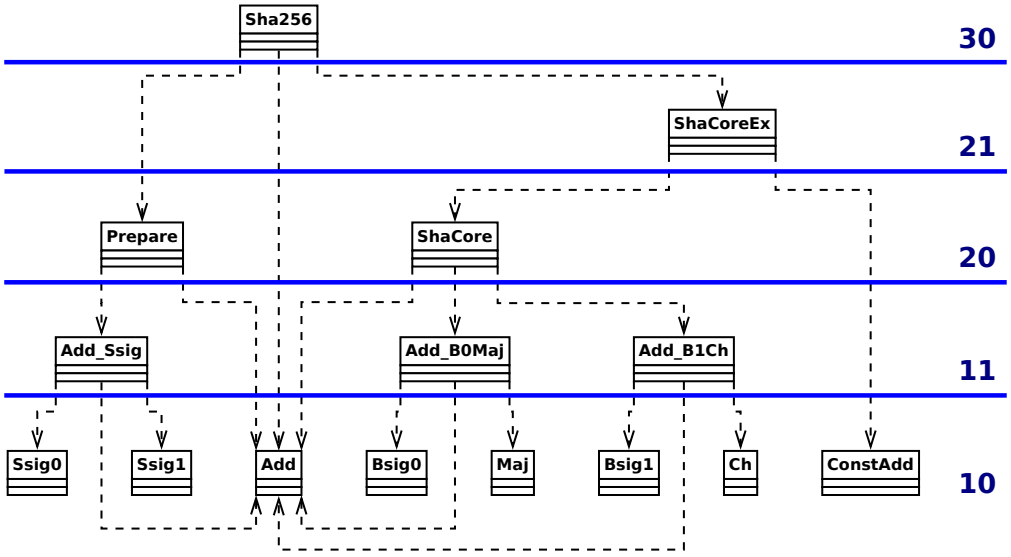


Abbildung 5.2 Modullevel von SHA-256

Klausel zur Zuordnung übergeben werden kann. Wird ein passendes Modul gefunden, wird ein Zeiger auf die Modulinformationen zurückgeliefert.

Das Programm „modulchecker“ nutzt schließlich die ModulDB für die Zuordnung. Zunächst wird ein Objekt der Kompressionsfunktion erzeugt, in das die ModulDB als Besucher übergeben wird. Danach werden die im vorigen Abschnitt erstellten DIMACS-Dateien eingelesen und jede Klausel einem Modul zugeordnet. Die Zuordnung wird in diesem Fall immer gelingen, da spätestens die Kompressionsfunktion alle Literale umfasst.

Da die Module in ihrer Normalform implementiert werden (siehe Kapitel 4) kann eine zugeordnete Klausel noch nicht direkt genutzt werden. Die verwendeten Literale in der Kompressionsfunktion sind andere als die des Moduls in der Normalform. Wird eine Klausel zugeordnet, erfolgt deshalb direkt im Anschluss die Normalisierung der Klausel. Diese wird auf Basis der im Modul verwendeten Literale durchgeführt. Da diese in der richtigen Reihenfolge vorliegen, können in der Klausel vorkommende Literale darin erkannt und anhand des Index in das Literal der Normalform überführt werden.

Nach der Normalisierung werden die Klauseln für jedes Modul in einem eigenen „set“ gesammelt. Wie in Abschnitt 5.1 erfolgt dadurch eine automatische Aussortierung doppelter Klauseln. Diese ist erneut notwendig, da Module mehrfach an verschiedenen Stellen verwendet werden. Klauseln die vorher unterschiedlich sind werden durch die Normalisierung zu gleichen Klauseln, wenn das gleiche Wissen über ein Modul an mehreren Stellen erworben wurde.

Abschließend werden die neuen Klauseln jedes Moduls in eine eigene DIMACS-Datei geschrieben. Bevor diese Klauseln in die Module integriert werden können ist es notwendig die Gültigkeit zu überprüfen. Ungültig kann eine Klausel dann sein, wenn sie sich aus dem Kontext, in dem das Modul verwendet wurde,

ergibt. In diesem Fall ist es notwendig, die Klausel auf ein Modul höheren Levels zu übertragen. Die Gültigkeitsprüfung wird mit dem Programm „clausechecker“ durchgeführt. Dabei werden die bekannten Klauseln des Moduls an CryptoMiniSat übergeben. Jede (neue) Klausel lässt sich als nicht erfüllbare Belegung interpretieren und wird deshalb als Annahme bei einem Lösungsversuch übergeben. Gelingt es CryptoMiniSat nicht, eine Lösung zu finden, ist die Gültigkeit der Klausel bestätigt. In dieser Arbeit haben sich jedoch alle überprüften Klauseln als gültig herausgestellt und brauchen nicht auf ein Modul höheren Levels übertragen werden.

Nur ein kleiner Teil (< 5%) der neuen Klauseln lässt sich einem Modul unterhalb des Levels 30 zuordnen. Der Großteil ergibt sich aus dem Zusammenspiel der Erweiterung der Eingabe und der Rundenfunktion und wird somit dem Modul für die Kompressionsfunktion zugeordnet. Der Versuch alle diese Klauseln in einen weiteren Lösungsversuch einzubringen hat gezeigt, dass dadurch der Lösungsprozess deutlich verlangsamt wird. Daraus ergibt sich die Notwendigkeit, diese Klauseln einer weiteren Analyse zu unterziehen, die im folgenden Abschnitt erläutert wird.

5.3 Distanzermittlung von Klauseln in der Kompressionsfunktion

Die Analyse der neuen Klauseln aus der Kompressionsfunktion erfolgt auf Basis einer Distanzmetrik. Je höher die Distanz einer Klausel, desto höher ist ihr angenommener Wert. Eine hohe Distanz bedeutet, dass die Klausel Literale aus Modulen in Zusammenhang bringt, die bisher in keiner direkten Beziehung stehen.

Für die Distanzberechnung wird ein ungerichteter Graph aus den Modulen mit Level zehn erstellt. Ein Modul definiert sich dabei aus zusätzliche Literalen und den Ausgangsliteralen. Im Gegensatz zum letzten Abschnitt fallen die Eingangsliterale weg, um eine eindeutige Zuordnung eines Literals zu einem Modul durchführen zu können. Realisiert wird der Graph durch einen Collector mit dem Namen „ModulGraph“. Genau wie die ModulDB aus dem letzten Abschnitt überschreibt der ModulGraph nur die Methode newModul des Collectors, um die Registrierung der Module zu erfassen. Um die Zuordnung der Literale zu einem Modul schnellstmöglich durchführen zu können, wird eine Lookup-Tabelle erstellt, in der zu jedem Literal das zugehörige Modul hinterlegt ist. Registriert sich ein Modul im ModulGraph, erfolgt zunächst die Aufnahme in die Lookup-Tabelle. Direkt im Anschluss wird das Modul mit den Modulen verknüpft, die die Eingangswerte liefern, und so der ungerichtete Graph erstellt.

Der ModulGraph stellt eine Funktion „printGraph“ bereit, mit der der ungerichtete Graph visualisiert werden kann. Die Ausgabe erfolgt in der Sprache DOT [Gra16] mit der Graphen einfach beschrieben werden können. Mit Hilfe eines Renderers kann die Beschreibung des Graphen in ein Grafikformat überführt werden. Optional kann als Parameter eine Klausel übergeben werden um Module, deren Literale in der Klausel vorkommen, farblich hervorzuheben. Abbildung 5.3 zeigt einen Ausschnitt aus dem vollständigen Graphen der Kompressionsfunktion. Abgebildet ist die dritte Anwendung der Rundenfunktion. Neben den Modulnamen werden die verwendeten Literale ausgegeben. Da der Addierer zusätzliche Literale benötigt (Carrybits),

sind bei diesem zwei Bereiche aufgeführt. Die anderen Module kommen ohne zusätzliche Literale aus und enthalten somit nur die Ausgangsliterale. Anzumerken sind die fünf ausgehenden Kanten der beiden finalen Addierer der Rundenfunktion. Die Ergebnisse werden jeweils in den vier folgenden Runden in fünf weiteren Modulen verwendet.

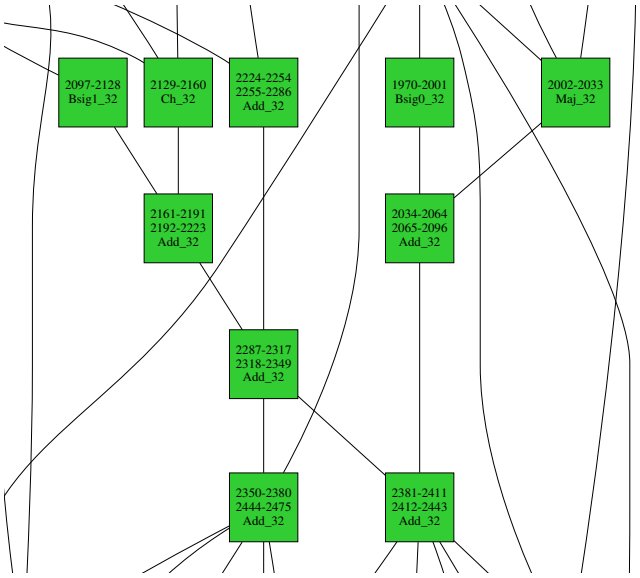


Abbildung 5.3 Ausschnitt des ungerichteten Graphen von SHA-256

Nach der Erstellung des Graphs werden die Distanzen der Module untereinander berechnet. Für jedes Modul wird dafür der Dijkstra-Algorithmus [Dij59] angewandt, um die Distanzen zu allen anderen Modulen zu berechnen. Jede Kante bekommt dabei das Gewicht eins. Genau wie für die Zuordnung der Literale zu den Modulen wird für die Distanzermittlung eine Lookup-Tabelle verwendet. In jedem Knoten werden so die Distanzen zu allen anderen Modulen abgelegt. Für die vollständige Kompressionsfunktion von SHA-256 stellt sich heraus, dass die größte Distanz zwischen zwei Modulen sechzehn beträgt; Es liegen somit maximal fünfzehn Module zwischen zwei anderen Modulen. Berechnet werden durch den Dijkstra-Algorithmus auch die Wege für die die Distanz gilt. Die Wege haben jedoch für diese Arbeit keine Relevanz und werden verworfen. Durch die Lookup-Tabellen kann die Berechnung der Distanzen einmalig bei Programmstart erledigt werden und große Klauselmengen können effizient bewertet werden.

Nachdem die Distanzen der einzelnen Module bekannt sind, kann damit die Distanz einer Klausel berechnet werden. Da eine Klausel mehr als zwei Module betreffen kann, werden die Distanzen aller betroffenen Module überprüft und die größte Distanz ausgewählt. Ermittelt wird außerdem die Anzahl der betroffenen Module. Ein Klausel mit sechs Literalen kann zum Beispiel zwei Module betreffen, wenn je drei Literale aus einem Modul stammen. Die Distanz einer Klausel ergibt sich aus der Formel in Abbildung 5.4.

Die Formel soll anhand eines Beispiels erläutert werden bei dem eine Klausel mit vier Literalen herangezogen wird. Die Zuordnung der Literale zu den Modulen ergibt drei unterschiedliche Module. Bei dem Vergleich

$$Klauseldistanz = größte\ Moduldistanz - Modulanzahl + 1$$

Abbildung 5.4 Formel zur Berechnung der Distanz einer Klausel

der Distanzen zwischen den Modulen wird eine maximale Distanz von drei ermittelt. Mit der Formel aus Abbildung 5.4 ergibt sich eine Klauseldistanz von eins.

Abbildung 5.5 zeigt zwei mögliche Fälle, die dem Beispiel entsprechen. In den gezeigten Graphen sind die Module, denen mindestens ein Literal zugeordnet wurde rot eingefärbt. Weitere Module denen kein Literal der Klausel zugeordnet werden konnte, sind in grün dargestellt.

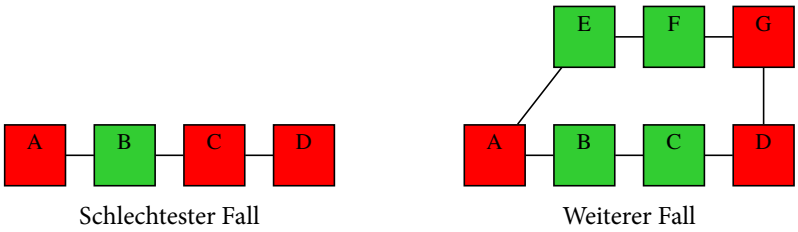


Abbildung 5.5 Beispiel für die Distanz einer Klausel

Im schlechtesten Fall ergibt sich die maximale Distanz zwischen den Modulen A und D, wobei das dritte Modul C auf dem Pfad zwischen den Modulen A und D liegt. Genau ein Modul B wird dabei übersprungen, was genau dem Wert der Klauseldistanz entspricht. Die Klauseldistanz sagt damit aus, wie viele Module im schlechtesten Fall übersprungen werden.

Ein weiterer Fall zeigt, dass die Klauseldistanz auch höher sein kann, wenn weitere Pfade von Modul A nach Modul D führen. Ein weiterer Pfad ist jedoch mindestens genau so lang wie der erste Pfad. Angenommen der zusätzliche Pfad hat die Länge vier und das dritte betroffenen Modul (in diesem Fall G) liegt auf diesem längeren Pfad, werden zwei Module übersprungen.

Um die Distanz für die neuen Klauseln aus der Kompressionsfunktion zu ermitteln, wird das Programm „distancechecker“ erstellt. Dieses Programm liest alle Klauseln ein und verteilt diese je nach Klauseldistanz auf eine eigene DIMACS-Datei. Vernachlässigt wird bei diesem Vorgehen die Länge der Klausel. Die Anzahl der Literale wird nur indirekt durch die Anzahl der betroffenen Module berücksichtigt. Dies wird aber vernachlässigt, da Klauseln mit großer Länge und Distanz durchaus interessant sein könnten, auch wenn es eher unwahrscheinlich ist, dass sie den Lösungsprozess in CryptoMiniSat beschleunigen.

Bei der Analyse zeigt sich, dass die Distanz der Klauseln breit gestreut ist. Die Distanz reicht von 5 bis unter -250. Wird zum Vergleich eine Klausel betrachtet, die einem einzelnen Modul mit Level zehn zugeordnet ist, ergibt sich dafür eine Klauseldistanz von 0 $(= 0 - 1 + 1)$. Darauf basierend werden im weiteren Verlauf nur Klauseln berücksichtigt, deren Distanz größer als null ist, in der Erwartung, dass sie zusätzliches Wissen darstellen und den Lösungsprozess beschleunigen.

5.4 Verallgemeinerung von Klauseln

Bisher erhaltene Klauseln werden in diesem Abschnitt darauf untersucht, ob eine Verallgemeinerung möglich ist. Verallgemeinerung bedeutet dabei, den Sinn der Klausel zu erkennen und auf weitere Stellen anzuwenden. Auf Modulebene erfolgt dieser Prozess bereits automatisch, da diese Klauseln überall dort angewendet werden, wo das Modul zum Einsatz kommt. Eine weitere Möglichkeit ist es, weitere Stellen innerhalb des Moduls zu identifizieren, an denen die Klausel ebenfalls gültig ist. Besonders gut funktioniert diese Verallgemeinerung bei dem Addierer. Durch die Breite von 32 Bit können viele Klauseln bitweise nach vorne oder hinten verschoben werden und sind an diesen Stellen ebenfalls gültig. Aufgrund der Vielzahl der erhaltenen Klauseln ist jedoch eine weitere Unterteilung notwendig. Zunächst werden die in Abschnitt 4.2 erläuterten Komponenten des Addierers (Halb-, Voll- und Mod-2 Addierer) in eigene Module mit dem Level eins ausgelagert. Dadurch können viele Klauseln direkt den Halb-, Voll und Mod-2 Addiereren zugeordnet werden. Ein Großteil der erworbenen Klauseln verbleibt aber nach wie vor im Modul des 32-Bit Addierers. Bei der Analyse der verbleibenden Klauseln zeigt sich, dass sich diese Klauseln in Halb-, Voll und Mod-X Addiereren höherer Ebene organisieren lassen. Als Addierer höherer Ebene wird ein Addierer bezeichnet, der bei Bedarf ein eingehendes oder ausgehendes Carrybit berücksichtigt, intern jedoch keine Carrybits verwendet. Diese Addierer entsprechen dem Carry-Lookahead Addierer (siehe Abschnitt 2.1) in unterschiedlichen Bitbreiten. Ergänzt werden Module für Addierer bis vier Bit Breite. Abbildung 5.6 zeigt die Erweiterung der Modulhierarchie.

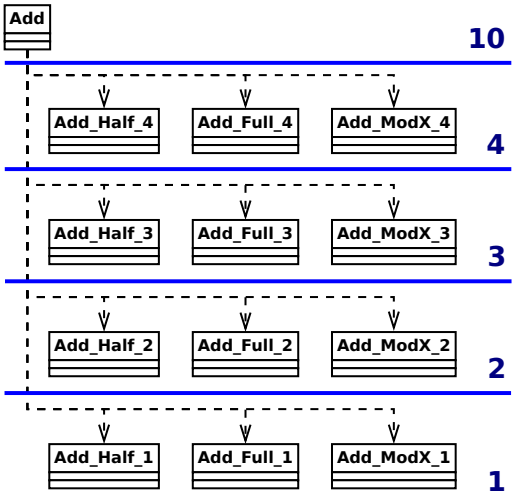


Abbildung 5.6 Ergänzte Module von SHA-256

Module für Addierer mit mehr als vier Bit Breite werden nicht ergänzt, da der SAT-Solver keine Klauseln ermittelt hat, die diesen Modulen zugeordnet werden konnten. Notwendig für die Funktion des Addierers sind nur die Module mit dem Level eins. Die Module mit dem Level zwei bis vier dienen als zusätzliche

Information auf den selben Literalen. Durch dieses Vorgehen erfolgt die Verallgemeinerung von Klauseln innerhalb des Addierers ebenfalls automatisch.

Bei den anderen Modulen zeigt sich kein Muster dieser Art und die Anzahl der erworbenen Klauseln ist vergleichsweise gering. Deshalb werden die erworbene Klauseln darin direkt untergebracht und durch Verwendung einer Schleife verallgemeinert in sofern dies möglich ist. Eine Ausnahme bildet das Modul der vollständigen Kompressionsfunktion. Neben der Verallgemeinerung auf Bitbreite bietet sich in diesem Modul die Möglichkeit, die Klauseln auch auf weitere Runden anzuwenden. Klauseln die sich nur auf eine Runde beziehen wurden schon dem Modul für die Rundenfunktion zugeordnet, so dass nur noch Klauseln betrachtet werden die sich über zwei oder mehr Runden erstrecken. Jedes Literal der jeweiligen Klausel wird um eine Runde nach vorne oder hinten verschoben.

Beachtet werden muss, dass jede der 64 Runden eine andere Rundenkonstante verwendet und die Erweiterung der Eingabe bei den ersten 16 Runden keine Anwendung findet. Während bei den anderen Modulen im Allgemeinen eine Verallgemeinerung entweder funktioniert oder nicht, muss dadurch jede durch die Verallgemeinerung generierte Klausel einzeln überprüft werden. Wie auch bei der Überprüfung der modulspezifischen Klauseln in Abschnitt 5.2 wird das Programm „clausechecker“ für die Überprüfung genutzt. Nur wenige der ermittelten Klauseln lassen sich ohne Einschränkungen auf weitere Bits oder Runden übertragen. Der clausechecker generiert deshalb ein zweidimensionales Array, in dem die gültigen Klauseln markiert werden. Dieses Array wird im Modul der Kompressionsfunktion genutzt, um die gültigen Klauseln in die konjunktive Normalform zu integrieren. Es zeigt sich, dass die Werte des Arrays in einigen Fällen die Rundenkonstante widerspiegeln. Im diesem Fall wird auf das Array verzichtet und die Bits der Rundenkonstante verwendet.

Es stellt sich heraus, dass dieses Vorgehen sehr erfolgreich ist weitere gültige Klauseln zu generieren. Auch Klauseln mit einer großen Distanz in der Kompressionsfunktion lassen sich oft auf 400 bis 700 weitere Stellen übertragen, wobei CryptoMiniSat davon nur wenige gefunden hat. In der iterativen Anwendung zeigt sich, dass die generierten Klauseln im nächsten Durchlauf dabei helfen, weitere Klauseln mit positiver Distanz zu finden. Ob sie auch dabei helfen den Lösungsprozess zu beschleunigen wird Kapitel 6 zeigen.

5.5 Subklauselprüfung

Durch die mehrfache iterative Anwendung der Methodik aus den letzten Abschnitten, kommt es zu einer kontinuierlichen Erweiterung der Klauselmenge. Dazu zählt auch, dass Don't-Care Literale in Klauseln identifiziert und entfernt werden. Wenn die neuen kürzeren Versionen dieser Klauseln durch die Analyse in die Klauselmenge aufgenommen werden, sind die alten langen Klauseln überflüssig. Um diese zu erkennen wird das Programm „clausesorter“ genutzt. Dieses bekommt als Eingabe alle bekannten Klauseln und überprüft diese auf kürzere Klauseln. Die naive Variante hat eine quadratische Laufzeit im Bezug zur Eingabegröße, da zu jeder Klausel alle weiteren überprüft werden müssen. Um die Laufzeit zu reduzieren wird eine Lookup-

Tabelle genutzt, in der zu jedem Literal die Klauseln hinterlegt sind, die das Literal enthalten. So werden zu jeder Klausel nur noch die Klauseln überprüft, die mindestens ein gemeinsames Literal haben. Dieses Vorgehen entspricht dem Watchlist-Prinzip, das auch im SAT-Solver „zChaff“ [Zha+01] Verwendung findet. In der Praxis reduziert sich die Laufzeit bei zwei Millionen Klausel dadurch von einer Woche auf fünf Minuten.

Die als überflüssig erkannten Klauseln werden in eine separate Datei ausgegeben und genau wie neue Klauseln einem Modul zugeordnet. Dadurch können diese Klauseln in den Modulen erkannt und entfernt werden. Zu Dokumentationszwecken werden die Klauseln nur auskommentiert. Eine Ausnahme bildet die Kompressionsfunktion in der die Klauseln mit positiver Distanz enthalten sind. Der Aufwand die überflüssige Klausel im Programmcode zu identifizieren ist bei geringem Nutzen vergleichsweise hoch. Das Modul, und somit jede enthaltene Klausel, wird nur einmal verwendet. Die Anzahl der überflüssigen Klauseln ist damit am Ende verhältnismäßig gering und sollte CryptoMiniSat nicht bremsen.

5.6 Erhaltene Klauseln

Nach der mehrfachen iterativen Anwendung der Methodik aus den letzten Abschnitten, wird die Analyse beendet. Bei den letzten Iterationen konnten keine weiteren Klauseln sinnvoller Länge für die Module gefunden werden. Anders ist die Situation bei Klauseln aus der Kompressionsfunktion mit positiver Distanz. Nach wie vor werden Klauseln generiert, die bei gleicher Distanz tendenziell kürzer werden. Da aber bisher nicht bekannt ist, ob diese Klauseln den Lösungsprozess tatsächlich beschleunigen, wird die Analyse beendet. Basierend auf der Bewertung der Klauseln in Kapitel 6 wird sich zeigen, ob die Fortsetzung der Suche nach Klauseln mit großer Distanz sinnvoll ist.

Tabelle 5.1 zeigt zunächst die erhaltenen Klauseln für die zusätzlichen Komponenten des Addierers. Die Halb- und Mod-X-Addierer sind jeweils einmal enthalten, während die Volladdierer mit zwei bis vier Bit 29 bis 27 Mal enthalten sind. Dadurch ergeben sich für den Addierer selbst insgesamt die zusätzlichen Klauseln in der unteren Zeile. Erkennen lässt sich, dass die Länge der Klauseln mit der Ebene der Komponenten zunehmen. Gleichzeitig nimmt die Anzahl der erhaltenen Klauseln ab. Das lässt sich dadurch erklären, dass bspw. ein 3-Bit Volladdierer auf den Literalen arbeitet, auf denen bereits zwei 2-Bit Volladdierer und drei Volladdierer vorhanden sind. Damit ist ein Großteil des Verhaltens bereits definiert und es werden nur zusätzliche Klauseln benötigt, die bisher fehlendes Verhalten definieren.

Tabelle 5.2 zeigt, wie viele Klauseln mit welcher Anzahl Literale in den essentiellen Modulen gefunden werden konnten. Dazu gehören sowohl normale Klauseln als auch XOR-Klauseln. Sind in einer Zelle zwei Werte aufgeführt, gibt der erste Wert die Anzahl der zusätzlichen Klauseln bei einer XOR-Unterstützung an. Der zweite Werte gibt die Anzahl der zusätzlichen Klauseln ohne XOR-Unterstützung wieder. Der Mod-2 Addierer verwendet, genau wie die Σ-Funktionen, ausschließlich den XOR-Operator. Weitere Klauseln waren darin deshalb nicht zu erwarten und wurden auch nicht gefunden. Bei den σ-Funktionen konnten weitere Klauseln gefunden werden da bei diesen, im Gegensatz zu den Σ-Funktionen, der Shift-Operator verwen-

	Klauselläng				
Modul	**3**	**4**	**5**	**6**	**7**
Add_Half_2	9	16	9		
Add_Full_2		22	38		
Add_ModX_2			24		
Add_Half_3		4	9	6	
Add_Full_3			17	33	
Add_ModX_3				27	
Add_Half_4			5	2	4
Add_Full_4					
Add_ModX_4					16
Add	9	658	1625	959	20

Tabelle 5.1 Erworbene Klauseln im Addierer

	Klauselläng					
Modul	**2**	**3**	**4**	**5**	**6**	**7**
Add_Half_1	1	2 - 3				
Add_Full_1		6 - 8	0 - 2			
Add_ModX_1						
Add		9	658	1625	959	20
Ssig0 (σ_0)				3 - 48		
Ssig1 (σ_1)				7 - 112		
Bsig0 (Σ_0)						
Bsig1 (Σ_1)						
Choose		64				
Majority						
Add_Ssig			8			
Add_B0Maj						
Add_B1Ch			244	122		
Prepare	2	8	68			
ShaCore	3	32	81 - 95	1		

Tabelle 5.2 Erworbene Klauseln in den Modulen

det wird, durch den einige Bits verworfen werden. Das zeigt sich auch bei der Addition der Ergebnisse der σ-Funktionen, wo ebenfalls einige Klauseln gefunden werden konnten.

Im Modul der Choose-Funktion wurden zwei weitere Klauseln gefunden, die sich auf die Breite von 32 Bit anwenden lassen. Dadurch ergeben sich 64 zusätzliche Klauseln. Im Gegensatz dazu konnten bei der Majority-Funktion keine weiteren Klauseln gefunden werden. Dieses Muster findet sich auch in den Modulen wieder, in denen das Ergebnis der Choose- und Majority-Funktion mit den Ergebnissen der Σ-Funktionen addiert wird. Nur in dem Modul, in dem die Choose-Funktion verwendet wird (Add_B1Ch), konnten weitere Klauseln gefunden werden.

Erfolgreicher ist die Ausbeute bei der Erweiterung der Eingabe (Prepare) und der Rundenfunktion (ShaCore). In diesen Modulen konnten einige Klauseln gefunden werden, die sich aus dem Zusammenspiel zwischen Modulen kleineren Levels ergeben.

In der Kompressionsfunktion wurden die Klauseln mit positiver Distanz aus Tabelle 5.3 ermittelt. Basis für die insgesamt 29703 Klauseln bildeten 1107 Klauseln aus CryptoMiniSat. 28596 Klauseln wurden durch die Verallgemeinerung auf Bitbreite und Runden ermittelt.

	Klauselläge					
Distanz	**2**	**3**	**4**	**5**	**6**	$\sum$
1	428	412				840
2	50	5287				5337
3		159	9560	1430		11149
4		318	1829	2068	5787	10002
5		28	188	1469	718	2403
$\sum$	478	6204	11577	4967	6505	29731

Tabelle 5.3 Erworbene Klauseln in der Kompressionsfunktion

Wie in Abschnitt 5.5 erwähnt sind von diesen Klauseln einige überflüssig, weil bereits Don't-Care Literale identifiziert wurden und kürzere Klauseln vorhanden sind. Tabelle 5.4 zeigt die Anzahl der sinnvollen Klauseln. 8425 Klauseln haben sich als überflüssig herausgestellt womit 21306 sinnvolle Klauseln verbleiben.

	Klauselläge					
Distanz	**2**	**3**	**4**	**5**	**6**	$\sum$
1	336	390				726
2	50	3812				3862
3		41	6849	954		7844
4			778	2004	3896	6678
5			33	1452	711	2196
$\sum$	386	4243	7660	4410	4607	21306

Tabelle 5.4 Erworbene Klauseln in der Kompressionsfunktion nach Bereinigung

Anzumerken ist, dass die Ermittlung von Klauseln mit positiver Distanz nicht abgeschlossen ist. Die bisher erhaltenen Klauseln werden in Kapitel 6 genutzt um zu prüfen ob sie den Lösungsprozess beschleunigen. Dabei wird sich zeigen ob es sinnvoll ist die Suche nach weiteren Klauseln zukünftig voranzutreiben.

Kapitel 6

Bewertung der zusätzlichen Klauseln

Die erhaltenen Klauseln aus Kapitel 5 werden in diesem Abschnitt darauf überprüft, ob und wie sehr sie den Lösungsprozess beschleunigen oder ausbremsen. Die Berechnungen wird auf einem Server mit folgenden Eigenschaften durchgeführt:

- Fedora 22
- 2 x Intel(R) Xeon(R) CPU E3-1240 V2 (3.40GHz)
- 2 x 4 Kerne (8 Threads)
- 32Gb RAM
- 16Gb Auslagerungsdatei

Wie auch für die Ermittlung der Klauseln in Kapitel 5 wird eine Urbildberechnung (siehe Abschnitt 3.5.2) für die Bewertung herangezogen. Eine annähernd sinnvolle Zeitmessung ist erst möglich, wenn mindestens neun Bits des Hashs vorgegeben werden. Die obere Grenze liegt bei 22 Bits, bei denen die genutzte Hardware noch zuverlässig zu einer Lösung kommt. Da CryptoMiniSat in der Standardkonfiguration mit acht Threads während des Lösungsprozesses Werte rät, kann es zufällig zu schnellen Lösungen kommen. Um ein aussagekräftiges Ergebnis zu bekommen, werden für neun bis 22 Bit jeweils fünf Versuche durchgeführt. Betrachtet wird auch jeweils eine Version mit und ohne XOR-Klauseln.

Um die Ergebnisse kompakt darstellen zu können, werden diese in Diagrammen dargestellt die mit der Legende in Abbildung 6.1 kurz erläutert werden sollen. Jede Spalte in der X-Achse stellt 5 Versuche dar, die mit der jeweils festgelegten Anzahl der Bits im Hash durchgeführt wurden. Die linke Y-Achse repräsentiert die Dauer in Sekunden. Da die Dauert im Bezug zu der Anzahl der vorgegebenen Bits exponentiell wächst, ist die Y-Achse logarithmisch skaliert. Die grünen Balken zeigen, in welchem Bereich die Ergebnisse der fünf Versuche liegen, wobei der blaue Punkt das arithmetische Mittel darstellt. Die rechte Y-Achse stellt die Anzahl der unterschiedlichen Ergebnisse dar, die durch die roten Punkte markiert werden. Dieser gibt somit einen Hinweis, wie viel Raten und somit Zufall bei der Lösung im Spiel war. Kommt CryptoMiniSat bei allen fünf Versuchen zum selben Ergebnis, haben die Heuristiken gut funktioniert und es musste wenig geraten werden. Das spiegelt sich auch darin wieder, dass die Dauer der fünf Versuche sich wenig unterscheidet.

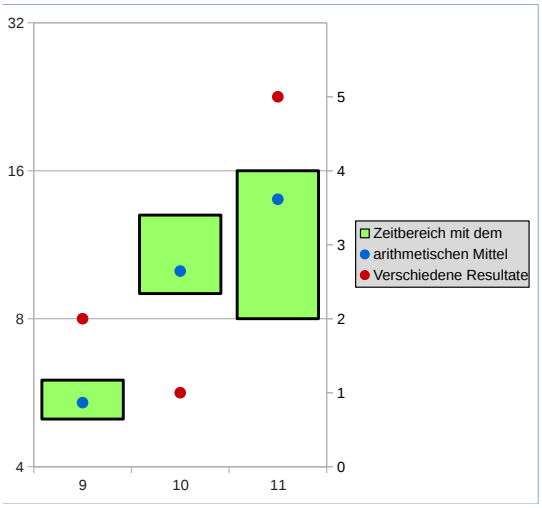
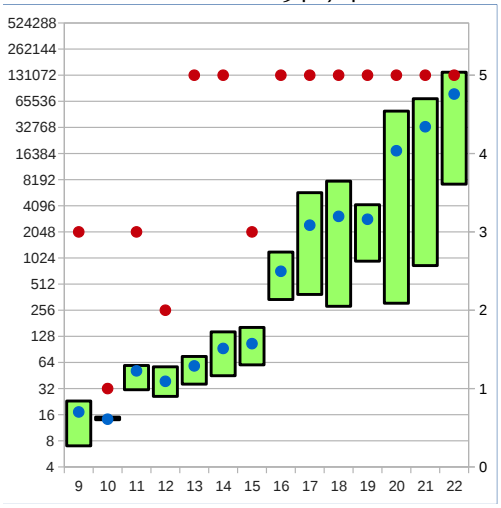


Abbildung 6.1 Legende

Als Vergleichsbasis dient die konjunktive Normalform aus Kapitel 4 ohne zusätzliche Klauseln. Die Ergebnisse sind in Abbildung 6.2 dargestellt. Es zeigt sich, dass die XOR-Variante insgesamt ein wenig schneller ist. Auffällig ist, dass die Zeit für die Variante ohne XOR sich im Bereich von 11 bis 15 Bit nur verdoppelt und bei 16 Bit ein Sprung erfolgt. In diesem Bereich hat die XOR-Variante einen Nachteil. Bezüglich der Anzahl der unterschiedlichen Ergebnisse gibt es keine relevanten Unterschiede.

Gesamtdauer ohne XOR: 194:27:41



Gesamtdauer mit XOR: 169:24:12

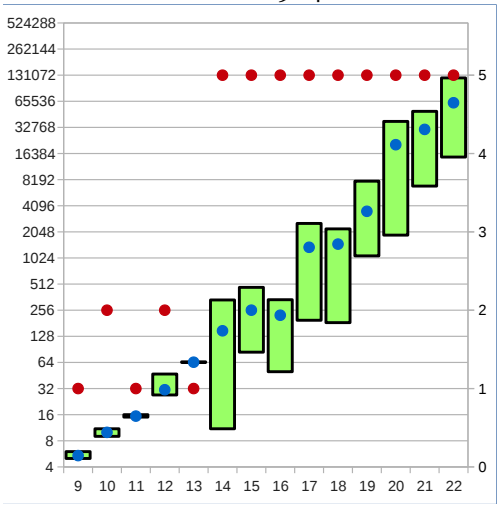
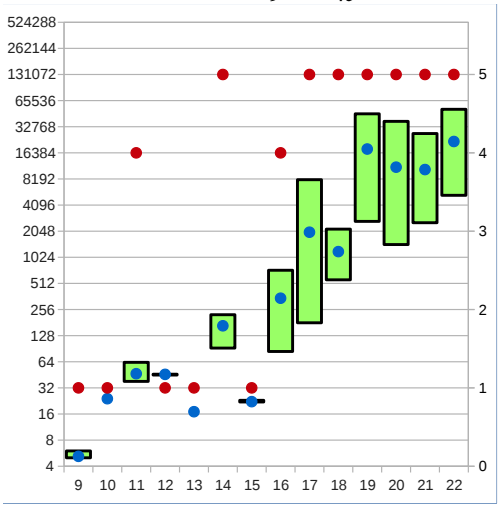


Abbildung 6.2 Vergleichsbasis

6.1 Test mit modulspezifischen Klauseln

Dieser Test bezieht zusätzlich zu den Basisklauseln die Klauseln aus Tabelle 5.2 mit ein. Ausgenommen sind davon jedoch die zusätzlichen Klauseln für den Addierer, die in Abschnitt 6.3 separat getestet werden. Die Ergebnisse sind in Abbildung 6.3 dargestellt. Insgesamt reduziert sich die Testlaufzeit um mehr als 50% wobei die XOR-Variante nach wie vor schneller ist. Ein weiterer Vorteil der XOR-Variante in diesem Test ist die Anzahl der unterschiedlichen Ergebnisse. Durch die zusätzlichen Klauseln konnte CryptoMiniSat mehr Berechnungen durchführen und es vermeiden Belegungen zu raten.

Gesamtdauer ohne XOR: 91:28:43



Gesamtdauer mit XOR: 69:07:57

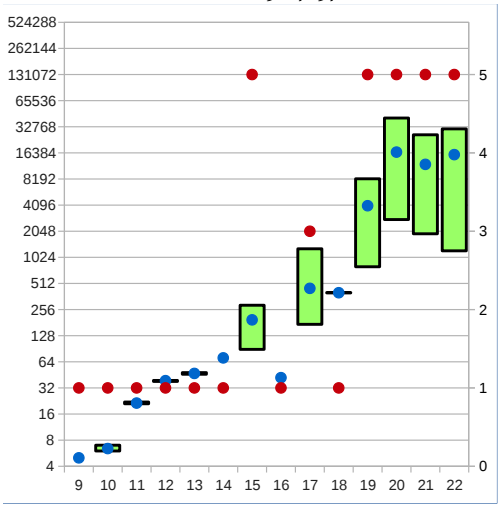
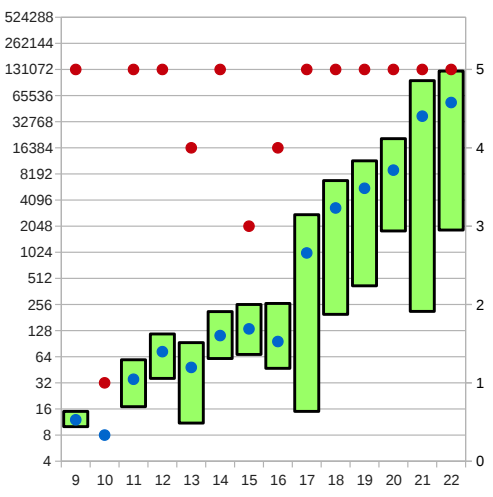


Abbildung 6.3 Ergebnisse mit modulspezifischen Klauseln

6.2 Test mit Distanzklauseln

In diesem Test werden zusätzlich zu den Basisklauseln die Klauseln aus Tabelle 5.4 mit einbezogen. Die Ergebnisse sind in Abbildung 6.4 dargestellt. Auch hier reduziert sich die Testlaufzeit. Im Vergleich zu den modulspezifischen Klauseln reduziert sich die Laufzeit nur um 20% anstatt 50%. Genau entgegen gesetzt ist jedoch das Verhalten bezüglich der Anzahl der unterschiedlichen Lösungen. Diese Klauseln führen dazu, dass mehr geraten und weniger berechnet wird. Wie auch bei den Basisklauseln lässt sich in der Variante ohne XOR-Klauseln ein Sprung der Lautzeit nach oben erkennen. Dieser erfolgt bei diesem Test jedoch erst bei 17 Bit.

Gesamtdauer ohne XOR: 155:31:37



Gesamtdauer mit XOR: 141:34:35

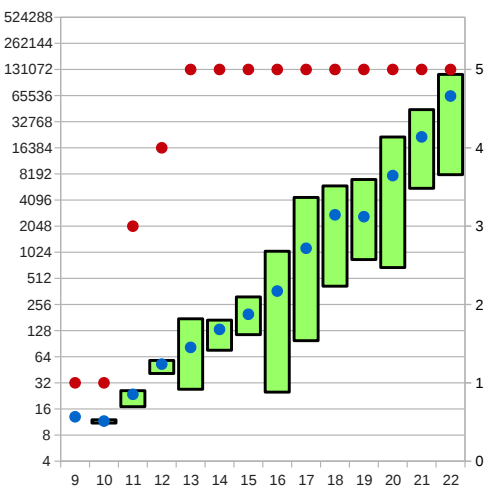
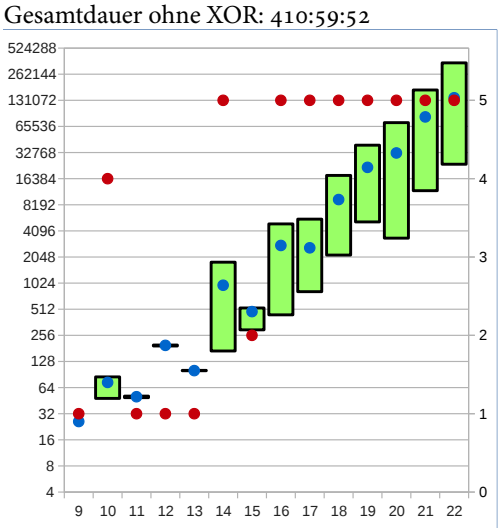


Abbildung 6.4 Ergebnisse mit Distanzklauseln

6.3 Test mit zusätzlichen Addiererklauseln

Im Test aus Abschnitt 6.1 wurden die modulspezifischen Klauseln getestet, jedoch zusätzliche Klauseln für den Addierer weggelassen. Das betrifft die Klauseln aus Tabelle 5.1 die in diesem Test separat getestet werden. Die Ergebnisse sind in Abbildung 6.5 dargestellt. Die Testlaufzeit beider Varianten sind mehr als doppelt so hoch und bei der Anzahl der unterschiedlichen Lösungen lässt sich keine Veränderung erkennen.

Gesamtdauer ohne XOR: 410:59:52



Gesamtdauer mit XOR: 491:53:48

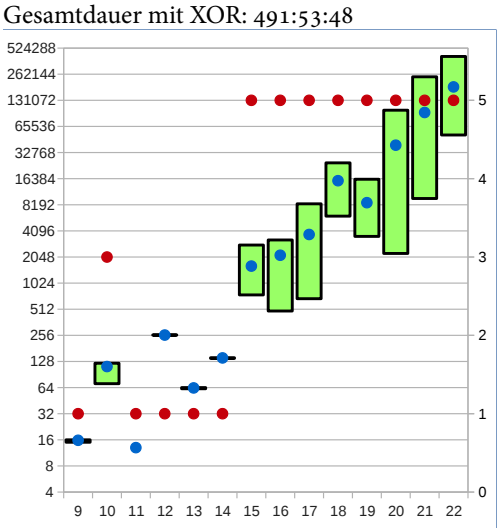


Abbildung 6.5 Ergebnisse mit zusätzlichen Addiererklauseln

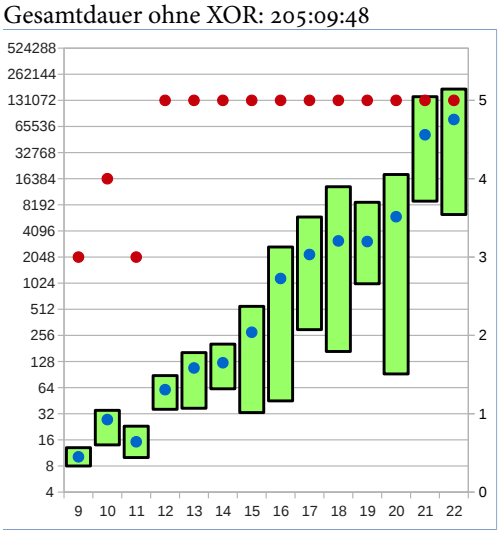
6.4 Test mit zusätzlichem Wissen

In Abschnitt 3.5.1 (Abbildung 3.10) wurde zusätzliches Wissen über die Rundenfunktion ermittelt/beschrieben. Für einen SAT-Solver mag dieses Wissen redundant sein und überflüssig erscheinen, da ein SAT-Solver nicht zwischen Vorwärts- und Rückwärts-Rechnen unterscheidet. Jedoch werden durch dieses Wissen bestehende Module noch einmal anders verknüpft, was einem SAT-Solver die Möglichkeit bieten könnte, einen alternativen Pfad zu nutzen und weitere Zusammenhänge zwischen Modulen zu erkennen. Nach dem selben Prinzip wäre es auch möglich, die Addition von mehreren Summanden in unterschiedlicher Reihenfolge zusätzlich einzufügen. Der Fokus liegt jedoch darauf, nur festzustellen ob sich das zusätzliche Wissen überhaupt auswirkt, weshalb auf die zusätzlichen Addierer verzichtet wird.

Ein Unterschied zu den vorhergehenden Tests ist, dass bei diesem Test nicht nur zusätzliche Klauseln notwendig sind sondern auch weitere Literale benötigt werden. Außerdem wird ein zusätzliches Modul für die Subtraktion genutzt, welches bisher nicht näher beschrieben wurde, da es nach dem selben Prinzip wie ein Addierer aufgebaut ist.

Die Ergebnisse sind in Abbildung 6.6 dargestellt. Während sich bei den vorhergehenden Tests die Laufzeit generell verkürzt oder verlängert hat, verhalten sich die beiden Varianten in diesem Tests entgegengesetzt. Während sich die Laufzeit bei der Variante ohne XOR-Klauseln leicht verlängert hat, hat sie sich bei der Variante mit XOR-Klauseln um 40% verkürzt. Bei beiden Varianten ist jedoch ersichtlich, dass sich die Anzahl der unterschiedlichen Lösungen speziell im Bereich mit wenig vorgegeben Bits erhöht hat. Das deutet darauf hin, dass CryptoMiniSat wieder mehr rät. Bei der Variante mit XOR-Klauseln hat das jedoch gut geklappt. Fraglich ist aber, ob das dem Zufall geschuldet ist.

Gesamtdauer ohne XOR: 205:09:48



Gesamtdauer mit XOR: 100:41:09

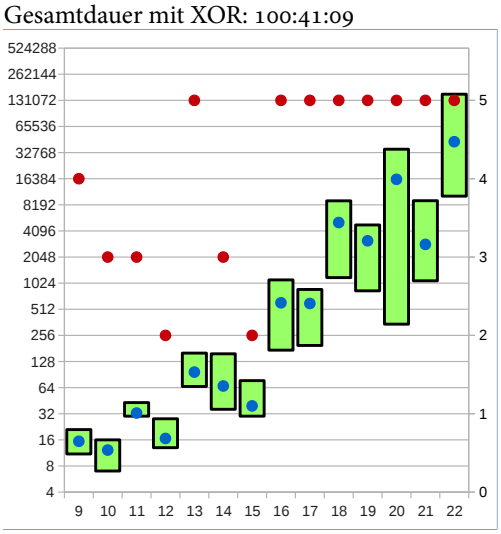
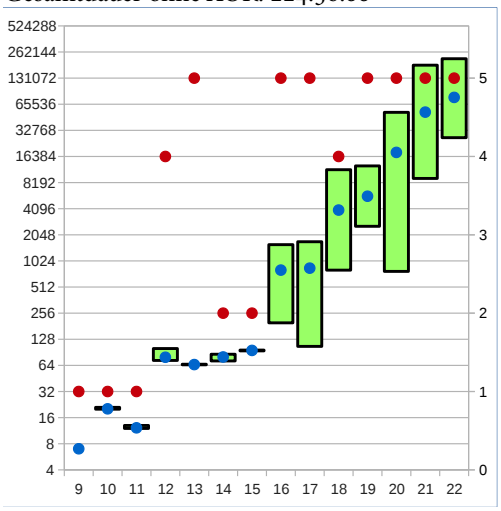


Abbildung 6.6 Ergebnisse mit zusätzlichem Wissen

6.5 Test mit allen vorteilhaften Klauseln

Nachdem die Klauselmengen in den vorhergehenden Abschnitten einzeln getestet wurden, werden sie in diesem Test gemeinsam genutzt. Ausgenommen davon sind die Addiererklauseln aus Abschnitt 6.3, da sie die Laufzeit negativ beeinflusst haben. Vernachlässigt wird dabei das negative Ergebnis der Klauseln aus Abschnitt 6.4 in der Variante ohne XOR-Klauseln. Die resultierende Laufzeit war in diesem Fall nur minimal höher und wird zur besseren Vergleichbarkeit mit der XOR-Variante trotzdem mit einbezogen. Die Ergebnisse sind in Abbildung 6.7 dargestellt. Wie auch bei den vorhergehenden Tests ist die XOR-Variante schneller. Die Laufzeit reduziert sich um 20% gegenüber der der Vergleichsbasis während die Laufzeit der Variante ohne XOR-Klauseln um 15% ansteigt. Die Anzahl der unterschiedlichen Lösungen hat sich im Bezug zur Vergleichsbasis wenig geändert.

Gesamtdauer ohne XOR: 224:36:00



Gesamtdauer mit XOR: 136:37:46

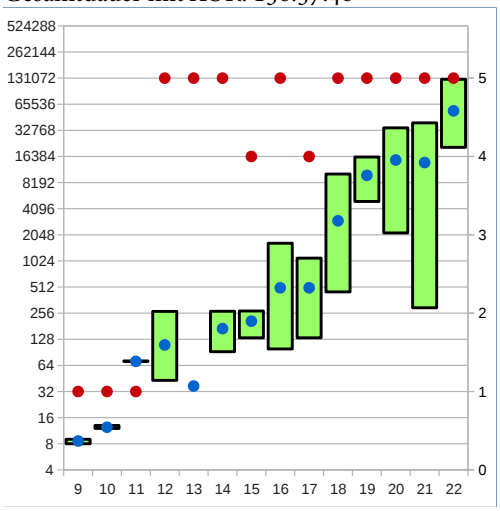


Abbildung 6.7 Ergebnisse mit allen vorteilhaften Klauseln

6.6 Vergleich der Resultate

In allen Tests hat die XOR-Variante besser abgeschnitten als die Variante ohne XOR-Klauseln. Im Folgenden wird deshalb nur noch die XOR-Variante betrachtet. Das beste Ergebnis haben die modulspezifischen Klauseln mit einer Reduktion der Testlaufzeit um fast 60% erzielt. Auch die Distanzklauseln und das zusätzliche Wissen haben die Testlaufzeit um 20% bzw. 40% reduziert. Negativ haben sich nur die zusätzlichen Addiererklauseln ausgewirkt. Die Vereinigung der positiv getestet Klauseln hat nur eine Reduktion der Testlaufzeit um 20% ergeben und konnte damit nicht an die 60% der modulspezifischen Klauseln anknüpfen. Das ist deshalb ein Problem, weil die modulspezifischen Klauseln mehr oder weniger „abgeschlossen“ sind.

Selbst wenn es in dieser Arbeit nicht gelungen ist, die optimale „vollständige“ Klauselmenge für die Module zu finden, ist diese doch begrenzt. Anders ist dies insbesondere bei den Distanzklauseln, da bei diesen noch die Möglichkeit besteht viele weitere Klauseln zu ermitteln. Aus diesem Grund wird die Klauselmenge aus Abschnitt 6.5 für die Evaluation in Kapitel 7 herangezogen. Gelingt es dieser Klauselmenge gegen andere Implementierungen zu bestehen, ist eine weitere Suche nach Distanzklauseln möglicherweise sinnvoll.

Kapitel 7

Evaluation

Nachdem im letzten Kapitel die unterschiedlichen Klauselmengen bewertet wurden, wird die Klauselmenge aus Abschnitt 6.5 in diesem Kapitel für die Evaluation herangezogen. Bisher wurden Klauseln mit Hilfe von acht Threads in CryptoMiniSat ermittelt und bewertet. Da mit dieser Konfiguration auch Werte geraten werden, hat das zu unterschiedlichen Ergebnissen und Laufzeiten geführt. Um dieses Verhalten zu vermeiden wird die Evaluation mit nur einem Thread durchgeführt, dessen Standardkonfiguration ein deterministisches Verhalten bewirkt und somit in annähernd gleicher Zeit immer zum selben Ergebnis führt.

Während die Implementierung dieser Arbeit im folgenden Abschnitt zunächst mit den Implementierungen von CBMC und Martin Maurer verglichen wird, erfolgt im Anschluss eine Bewertung, ob praktische relevante Angriffe auf SHA-256 und Bitcoin möglich sind.

7.1 Vergleich

Um einen fairen Vergleich zu ermöglichen, werden alle Implementierungen unter gleichen Bedingungen mit CryptoMiniSat getestet. Genau wie in Kapitel 6 wird eine Urbildberechnung (siehe Abschnitt 3.5.2) für den Vergleich herangezogen. Die Tests erfolgen jedoch nur von neun bis 21 Bit da die Laufzeit mit einem Thread länger ist und eine Berechnung für 22 Bit nicht in angemessener Zeit abgeschlossen werden konnte. Außerdem ist pro Bit nur ein Versuch notwendig, da das Verhalten deterministisch ist und die Laufzeit somit sehr ähnlich. Abbildung 7.1 zeigt das Ergebnis des Vergleichs. Zur besseren Übersicht sind die einzelnen Punkte im Diagramm durch Linien verbunden auch wenn es dazwischen praktisch keine weiteren Messungen gibt.

Oberflächlich betrachtet hat die in dieser Arbeit erstellte Implementierung mit einer Gesamtdauer von 105 Stunden am besten abgeschnitten. Die Implementierung von CBMC benötigte 344 Stunden. Dieser große Unterschied ergibt sich jedoch ausschließlich aus dem letzten Test mit 21 Bit. Wird der gesamte Verlauf mit einbezogen, weisen beide Implementierungen ein ähnliches Verhalten auf. Die Implementierung dieser Arbeit kann an drei Stellen (13, 17 und 20 Bit) vergleichsweise schnell eine Lösung finden während CBMC dies an zwei Stellen (10 und 20 Bit) gelingt.

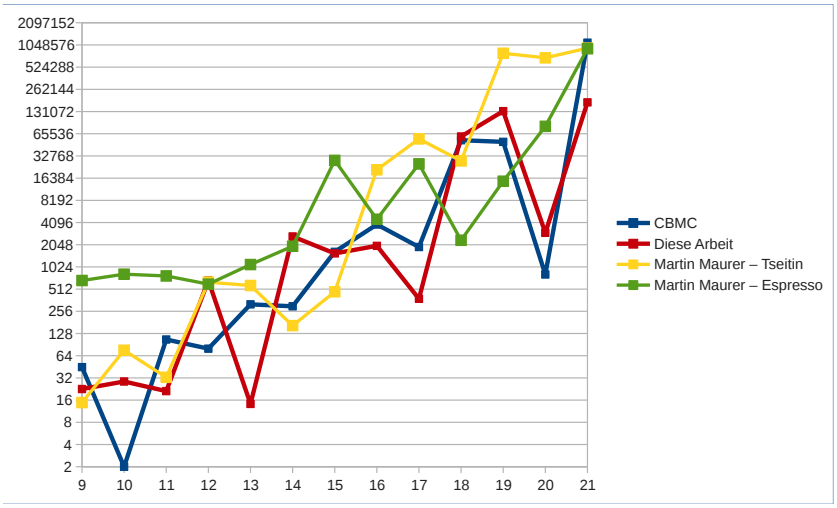


Abbildung 7.1 Eigene Implementierung

Die Tseitin-Variante von Martin Maurer verhält sich bis 15 Bit ebenfalls ähnlich, benötigt danach aber wesentlich mehr Rechenzeit. Inklusive des Versuchs mit 19 Bit benötigt die Tseitin-Variante bereits über 250 Stunden und die Versuche mit 20 und 21 Bit wurden abgebrochen. Anders verhält sich die Espresso-Variante. Diese benötigt zunächst länger, verhält sich aber ab 16 Bit wieder ähnlich zur Implementierung dieser Arbeit und CBMC. Damit bestätigt sich die Aussage von Martin Maurer, dass in seinen Tests die Tseitin-Variante schneller war, da er nur Versuche mit einem wenig eingeschränkten Lösungsraum durchgeführt hat. Für die schwierigen Probleme setzt sich jedoch die Espresso-Variante durch. Für die Tests benötigte diese insgesamt 300 Stunden.

7.2 Bedeutung für SHA-256

Die Versuche aus Kapitel 6 und Abschnitt 7.1 zeigen, dass sich innerhalb eines Tages ein Urbild für einen Hash berechnen lässt, der je nach Implementierung mit 18 bis 20 festgelegten Bits beginnt. Das ist jedoch kein Vergleich zum Ergebnis, das sich durch reines Probieren erzielen lässt. In der Testumgebung können 720.000 Hashes pro Sekunde berechnet werden. Damit ist es möglich innerhalb eines Tages ein Urbild für einen Hash mit bis zu 35 festgelegten Bits zu finden.

7.2.1 Initialwertberechnung

Abschnitt 3.5.3 beschreibt den Sinn einer Initialwertberechnung. Diese wird im folgenden separat betrachtet, da die Erweiterung der Eingabe (siehe Abschnitt 3.3) vollständig berechnet werden kann und somit durch

einen SAT-Solver nicht mehr betrachtet werden braucht. Das Problem reduziert sich damit auf die Berechnung der 64 Runden. Abbildung 7.2 zeigt das Ergebnis dieses Versuchs im Vergleich zur Urbildberechnung.

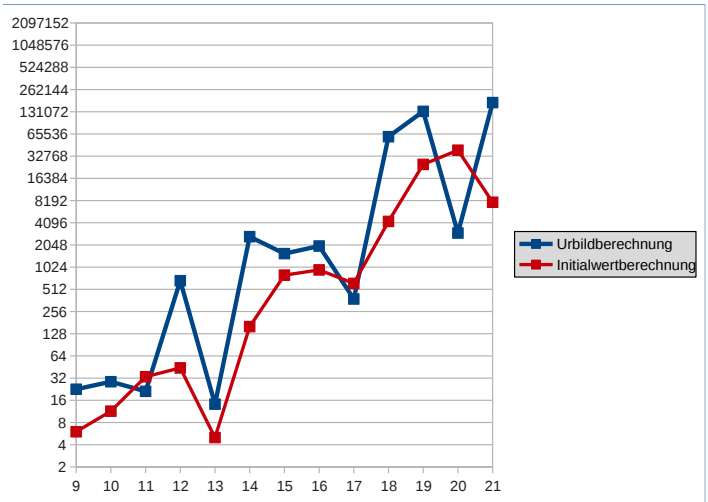


Abbildung 7.2 Evaluation - Initialwertberechnung

Während die Urbildberechnung insgesamt 105 Stunden benötigt hat, wurde die Initialwertberechnung in 22 Stunden abgeschlossen. Mit drei Ausnahmen war die Initialwertberechnung in allen Fällen schneller als die Urbildberechnung.

7.2.2 Kollisionsberechnung

Abschließend wird die in Abschnitt 3.5.4 beschriebene Kollisionsberechnung durchgeführt. Abbildung 7.3 zeigt das Ergebnis im Vergleich zur Urbildberechnung.

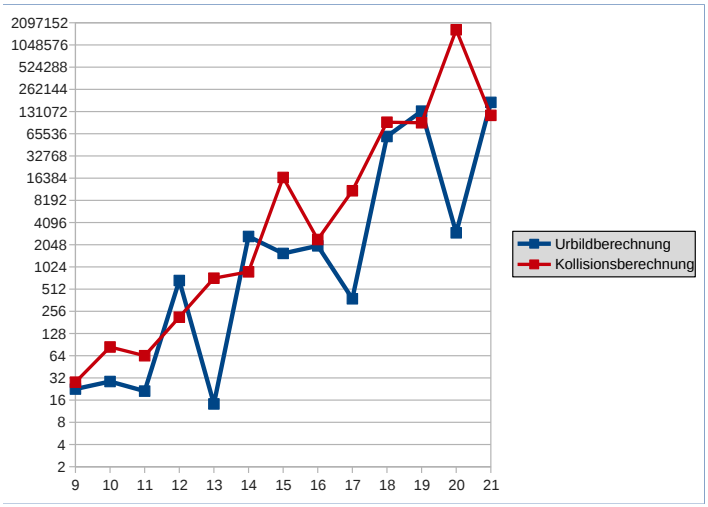


Abbildung 7.3 Evaluation - Kollisionsberechnung

Es zeigt sich, dass die freie Wahl eines Hash für CryptoMiniSat keinen Vorteil darstellt. In den meisten Fällen dauert eine Kollisionsberechnung länger als eine Urbildberechnung.

7.3 Bedeutung für das Bitcoin-Mining

Jonathan Heusser führt in seinem Artikel [J H13a] zwei Versuche mit unterschiedlichen SAT-Solvern durch. Diese ließen sich für einen direkten Vergleich leider nicht reproduzieren, da die vom ihm genutzte konjunktive Normalform unterschiedlich zu der ist, die sich mit CBMC aus seinem C-Programmcode erstellen lässt. Außerdem gibt er in seinen Versuchen eine feste Anzahl führender Nullen für den Hash vor und variiert die Anzahl der Stellen der Nonce die noch berechnet werden müssen um die Schwierigkeit zu verändern. Das eignet sich zwar für den Vergleich unterschiedlicher SAT-Solver, ermöglicht jedoch keine direkte Aussage dazu, ob sich dieses Vorgehen dafür eignet aktuelle Bitcoin-Blöcke zu lösen.

In der Praxis ist die Nonce vollständig unbekannt und die Schwierigkeit wird dadurch angepasst, dass die Anzahl der geforderten führenden Nullen sich ändert. Ziel von Bitcoin ist es, die Schwierigkeit immer so anzupassen, dass im Mittel alle zehn Minuten ein Block gelöst wird. Damit stellt sich die Frage, bei wie vielen führenden Nullen der SAT-Solver innerhalb von 10 Minuten zu einer Lösung kommt. Tabelle 7.1 zeigt die Ergebnisse der mit CryptoMiniSat durchgeführten Versuche.

Für jede Anzahl der geforderten Nullen wurden jeweils fünf Versuche durchgeführt, die durch eine Zeile in Tabelle 7.1 repräsentiert werden. Es hat sich gezeigt, dass CryptoMiniSat innerhalb jeder Versuchsreihe jeweils die selbe Nonce ermittelt hat und die Dauer nur wenig variierte. Die angegebene Dauer ist der arithmetische Mittelwert. Die letzte Spalte enthält die Anzahl der führenden Nullen die tatsächlich erreicht wurden. Da ein SAT-Solver die nicht festgelegten Stellen frei belegen kann, ist es möglich, dass das Ergebnis

Geforderte Nullen	Dauer in Minuten	Erhaltene Nullen
9	0:43	9
10	1:55	10
11	8:32	11
12	5:29	14
13	10:39	16

Tabelle 7.1 Erworbene Klauseln in der Kompressionsfunktion nach Bereinigung

mehr führende Nullen hat als gefordert. Dies ist jedoch zufällig und nur zur Vollständigkeit mit aufgeführt. In annähernd zehn Minuten ist es somit möglich eine Nonce für einen Hash mit bis zu 13 führenden Nullen zu ermitteln.

Da die aktuellen Blöcke einen Hash mit mindestens 68 führenden Nullen erfordern, zeigt dieser Versuch, dass SAT-Solving sich derzeit nicht für den Einsatz in Bitcoin-Minern eignet. Betrachtet werden muss auch, dass in der Testumgebung 720.000 Hashes pro Sekunde berechnet werden können. Damit kann alleine durch Probieren in zehn Minuten eine gültige Nonce für einen Hash mit 27 führenden Nullen ermittelt werden.

Kapitel 8

Fazit

In Kapitel 4 konnte im Vergleich zu CBMC und der Arbeit von Martin Maurer eine deutlich kompaktere konjunktive Normalform erzeugt werden. Das dafür erstellte Framework wurde frühzeitig veröffentlicht und von Jens Trillmann genutzt um den Verschlüsselungs-Algorithmus DES in eine konjunktive Normalform zu überführen [J T16]. Wird ein Klartext (64 Bit) vorgegeben und werden vom Geheimtext 48 Bit festgelegt, ist CryptoMiniSat in der Lage innerhalb von 24 Stunden einen gültigen Schlüssel für DES zu finden. Das Framework und die konjunktive Normalform dürfen deshalb als Erfolg betrachtet werden.

Basierend auf den Lösungsversuchen mit der konjunktiven Normalform aus Kapitel 4 konnten in Kapitel 5 zusätzliche Klauseln unterschiedlichster Art generiert werden. Auch hier hat sich das Framework bewährt und eine Zuordnung der Klauseln ermöglicht. Besonders einfach und effektiv war die Zuordnung der Klauseln zu einzelnen Modulen. Wissen was an einer Stelle über ein Modul erworben wurde, konnte so automatisch an allen anderen Stellen, an denen das Modul verwendet wird, genutzt werden. Kapitel 6 hat gezeigt, dass sich gerade diese modulspezifischen Klauseln besonders positiv auf die Dauer des Lösungsprozesses auswirken. Eine Ausnahme bilden dabei zusätzliche Klauseln für den 32 Bit Addierer. Diese bremsen den Lösungsprozess im Gegensatz zu zusätzlichen Klauseln für die Halb- und Volladdierer eher aus. Vom 32 Bit Addierer abgesehen scheint somit nicht die minimale Klauselmenge am schnellsten zum Ziel zu führen, sondern die Menge aller Klauseln ohne Don't-Care Literale. Das ermöglicht einem SAT-Solver an jeder Stelle eine direkte Propagation von Werten ohne den Umweg über weitere Klauseln. Beachtet werden muss jedoch auch, dass der Lösungsprozess gerade bei einem Hash-Algorithmus wie SHA-256 eher auf Probieren basiert, da zumindest bei dem aktuellen Stand vergleichsweise wenig Wissen erworben und genutzt werden kann. Das macht die effiziente Propagation von Werten um so wichtiger. Die modulspezifischen Klauseln beschleunigen am Ende zwar den Lösungsprozess, lassen aber keine weiteren Optimierungen zu, da diese Klauselmenge mehr oder weniger endlich ist.

Diesbezüglich sind modulübergreifende Klauseln wesentlich interessanter, da die Suche danach in dieser Arbeit nicht abgeschlossen wurde und noch Potenzial vorhanden ist. Die Distanzmetrik hat sich bewährt, da die Dauer des Lösungsprozesses mit vergleichsweise wenig zusätzlichen Klauseln um 20% reduziert werden konnte. Damit ist es zumindest geglückt, einen Teil der Klauseln herauszufiltern, die den Lösungsprozess tatsächlich beschleunigen. Zur Ermittlung der Distanz wurde in dieser Arbeit ein Graph herangezogen, dessen

Knoten auf den grundlegenden Module basieren. Dafür wäre ein allgemeinerer Ansatz denkbar, der einen Graphen anhand der Literale und Klauseln erzeugt, um darauf eine Distanz zu ermitteln. Dieser Graph wird jedoch wesentlich umfangreicher und hätte es in dieser Arbeit erschwert die gefundenen Klauseln zu verallgemeinern, da der Zusammenhang zwischen den einzelnen Bits eines 32 Bit breiten Zwischenergebnisses verloren geht.

Diese Verallgemeinerung von modulübergreifenden Klauseln hat sich in dieser Arbeit ebenfalls bewährt. Vielfach konnten Klauseln mit positiver Distanz auf bis zu 700 weitere Stellen übertragen werden, wobei CryptoMiniSat davon nur 10 bis 50 selbst gefunden hat. Das zeigt, dass es durchaus sinnvoll sein kann einen SAT-Solver mit externem Wissen zu unterstützen. Während in dieser Arbeit gerade bei der Verallgemeinerung noch viele Schritte von Hand durchgeführt wurden, kann es zukünftig interessant sein diesen Prozess vollständig zu automatisieren. Eine erworbene Klausel mit positiver Distanz könnte so direkt verallgemeinert und in den Lösungsprozess integriert werden. Beachtet werden muss dabei jedoch, dass alle durch Verallgemeinerung erzeugten Klauseln noch einer Gültigkeitsprüfung unterzogen werden müssen, was nebenläufig in einem weiteren Thread erledigt werden kann.

Ein weiterer Aspekt bezüglich der modulübergreifenden Klauseln ist, dass viele dieser Klauseln Carry-Bits der 32 Bit Addierer mit einbeziehen. Deshalb ist es besonders interessant, auch tatsächlich einen Carry-Ripple Addierer zu nutzen und nicht durch die Aneinanderreihung von 2 Bit Addierern ohne internes Carry-Bit Literale einzusparen. Nur dadurch kann die Verallgemeinerung ihr maximales Potential entfalten.

Zusätzliche Klauseln wurden des Weiteren nicht nur durch CryptoMiniSat sondern auch durch zusätzliches Wissen und weitere Module manuell erzeugt. Ein Unterschied ist hierbei, dass neben weiteren Klauseln auch weitere Literale für zusätzliche Zwischenergebnisse benötigt werden. Diese können unter Umständen helfen, weitere Zusammenhänge zu erkennen. Die Bewertung war jedoch nicht eindeutig, und der Nutzen ist zweifelhaft. Nur bei der Verwendung von XOR-Klauseln konnte der Lösungsprozess damit beschleunigt werden.

Die Evaluation in Kapitel 7 hat gezeigt, dass es in dieser Arbeit gelungen eine konjunktive Normalform zu erzeugen, die schneller ist als die von Martin Maurer und die mit CBMC mithalten kann. Das Ergebnis ist jedoch kein Vergleich zu dem, was sich durch reines Probieren realisieren lässt. Interessant bleibt dieser Ansatz trotzdem, da weitere Klauseln den Lösungsprozess noch weiter beschleunigen können.

Im Gegensatz zu spezifischen Angriff auf SHA-256 lässt sich durch diesen Ansatz jedes neue Wissen problemlos integrieren und nutzen. Die Suche nach weiteren Klauseln könnte auch durch eine große Gemeinschaft erfolgen. Zukünftig wäre ein Internetauftritt denkbar, durch den alle Klauseln gesammelt und der Allgemeinheit zur Verfügung gestellt werden.

Akronyme

CBMC C Bounded Model Checker, S. 10, 11, 23, 39, 61, 62, 64, 67, 68, 80

DES Data Encryption Standard, S. 4, 67

GiB Gibibyte, S. 14

IETF Internet Engineering Task Force, S. 14, 17

NIST National Institute of Standards and Technology, S. 13, 14

RFC Request for Comments, S. 14, 17

Glossar

Geburtstagsparadoxon

Die Wahrscheinlichkeit in einem Raum zwei Personen mit einem gleichen Geburtstag zu finden ist wesentlich höher, als zu einem gegebenen Geburtstag eine weitere Person zu finden.

S. 13

Hash

Ein Fingerabdruck konstanter Länge von Daten beliebiger Länge.

S. 1, 11, 13, 14, 20, 21, 41, 42, 53, 62, 64, 65

Initialwert

Im Fall von SHA-256 eine 256 Bit lange Zeichenfolge mit der der Hash-Algorithmus initialisiert wird.

S. 20, 21, 41, 42

Padding

Fülldaten, mit denen vorhandene Daten verlängert werden.

S. 20

SHA-256

Hash-Algorithmus der einen 256 Bit langen Fingerabdruck (Hash) von Daten beliebiger Länge erzeugt.

S. 2–5, 11, 14, 16, 19, 20, 23, 24, 26, 37–39, 41, 42, 44, 46, 48, 61, 67, 68, 75, 76

Literaturverzeichnis

[A A09] A. Arul Lawernce Selvakumar and Dr. R. S. Ratastogi. *Study the function of building blocks in SHA Family*. Jan. 2009. URL: *http://arxiv.org/pdf/1402.1314.pdf*.

[Bit09] Bitcoin. *Block #0*. 2009. URL: *https://blockexplorer.com/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f*. (Abgerufen am 03. April 2017).

[chm12] chmod755. *Espresso Heuristic Logic Minimizer*. 2012. URL: *http://chmod755.tumblr.com/post/31417234230/espresso-heuristic-logic-minimizer*. (Abgerufen am 03. April 2017).

[Com+02] U.S. Department of Commerce, National Institute of Standards, Technology und Information Technology Laboratory. *Secure Hash Standard*. FIPS PUB 180-2. National Institute of Standards und Technology, Aug. 2002. URL: *http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf*.

[D B93] D. Brand. *Verification of Large Synthesized Designs*. 1993. URL: *https://www.computing.ece.vt.edu/~mhsiao/ece5506/papers/iccad93_brand.pdf*.

[D S15] D. S. Pastor. *MinUnit*. 2015. URL: *https://github.com/siu/minunit*.

[Dij59] E. W. Dijkstra. »A note on two problems in connexion with graphs«. In: *Numerische Mathematik* 1 (1959), S. 269–271.

[EH06] D. Eastlake und T. Hansen. *US Secure Hash Algorithms (SHA and HMAC-SHA)*. RFC 4634 (Proposed Standard). Internet Engineering Task Force, Juli 2006. URL: *https://tools.ietf.org/html/rfc4634*.

[EH11] D. Eastlake und T. Hansen. *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*. RFC 6234 (Proposed Standard). Internet Engineering Task Force, Mai 2011. URL: *https://tools.ietf.org/html/rfc6234*.

[EJ01] D. Eastlake und P. Jones. *US Secure Hash Algorithm 1 (SHA1)*. RFC 3174 (Proposed Standard). Internet Engineering Task Force, Sep. 2001. URL: *https://tools.ietf.org/html/rfc3174*.

[Gam+96] E. Gamma, R. Helm, R. Johnson und J. Vlissides. *Entwurfsmuster*. 5. Auflage. Addison-Wesley, 1996. ISBN: 3-8273-1862-9.

[Gra16] Graphviz. *The DOT Language*. 2016. URL: *http://www.graphviz.org/doc/info/lang.html*.

[J H13a] J. Heusser. *SAT solving - An alternative to brute force bitcoin mining*. Feb. 2013. URL: *http://jheusser.github.io/2013/02/03/satcoin.html*. (Abgerufen am 03. April 2017).

[J H13b] J. Heusser. *satcoin*. Dez. 2013. URL: *https://github.com/jheusser/satcoin*. (Abgerufen am 03. April 2017).

[J T16] J. Trillmann. *DEScrack*. März 2016. URL: *https://github.com/jtrillma/DEScrack*. (Abgerufen am 03. April 2017).

[M M13] M. Maurer. *sha256-sat-bitcoin – SAT instance generator for SHA-256 and BITCOIN*. Dez. 2013. URL: *https://github.com/capiman/sha256-sat-bitcoin*. (Abgerufen am 03. April 2017).

[PMR15] P. Pritzker, W. E. May und C. H. Romine. *Secure Hash Standard (SHS)*. FIPS PUB 180-4. National Institute of Standards und Technology, Aug. 2015. URL: *http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf*.

[PP01] C. Paar und J. Pelzl. *Understanding Cryptography*. Springer, 2001. ISBN: 978-3-642-04100-6.

[Sato8a] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008. URL: *https://bitcoin.org/bitcoin.pdf*. (Abgerufen am 03. April 2017).

[Sato8b] Satoshi Nakamoto. *Bitcoin P2P e-cash paper*. 2008. URL: *http://www.mail-archive.com/cryptography@metzdowd.com/msg09959.html*. (Abgerufen am 03. April 2017).

[Shto1] O. Shtrichman. »Pruning techniques for the SAT-based bounded model checking problem«. In: *Proceedings of the correct hardware design and verification methods* (2001). Volume 2144 of lecture notes in computer science, S. 58–70.

[V N12] V. Nossum. *SAT-based preimage attacks on SHA-1*. Nov. 2012. URL: *https://www.duo.uio.no/handle/10852/34912*. (Abgerufen am 03. April 2017).

[V N13] V. Nossum. *sha1-sat – SAT instance generator for SHA-1*. März 2013. URL: *https://github.com/vegard/sha1-sat*. (Abgerufen am 03. April 2017).

[Wik16] Wikipedia. *Tseytin transformation — Wikipedia, The Free Encyclopedia*. Feb. 2016. URL: *https://en.wikipedia.org/wiki/Tseytin_transformation*. (Abgerufen am 03. April 2017).

[Zha+01] L. Zhang, C. F. Madigan, M. W. Moskewicz und S. Malik. »Efficient conflict driven learning on a Boolean satisfiability solver«. In: *Proceedings of the international conference on computer-aided design* (2001), S. 279–285.

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang A

CD und Inhalt

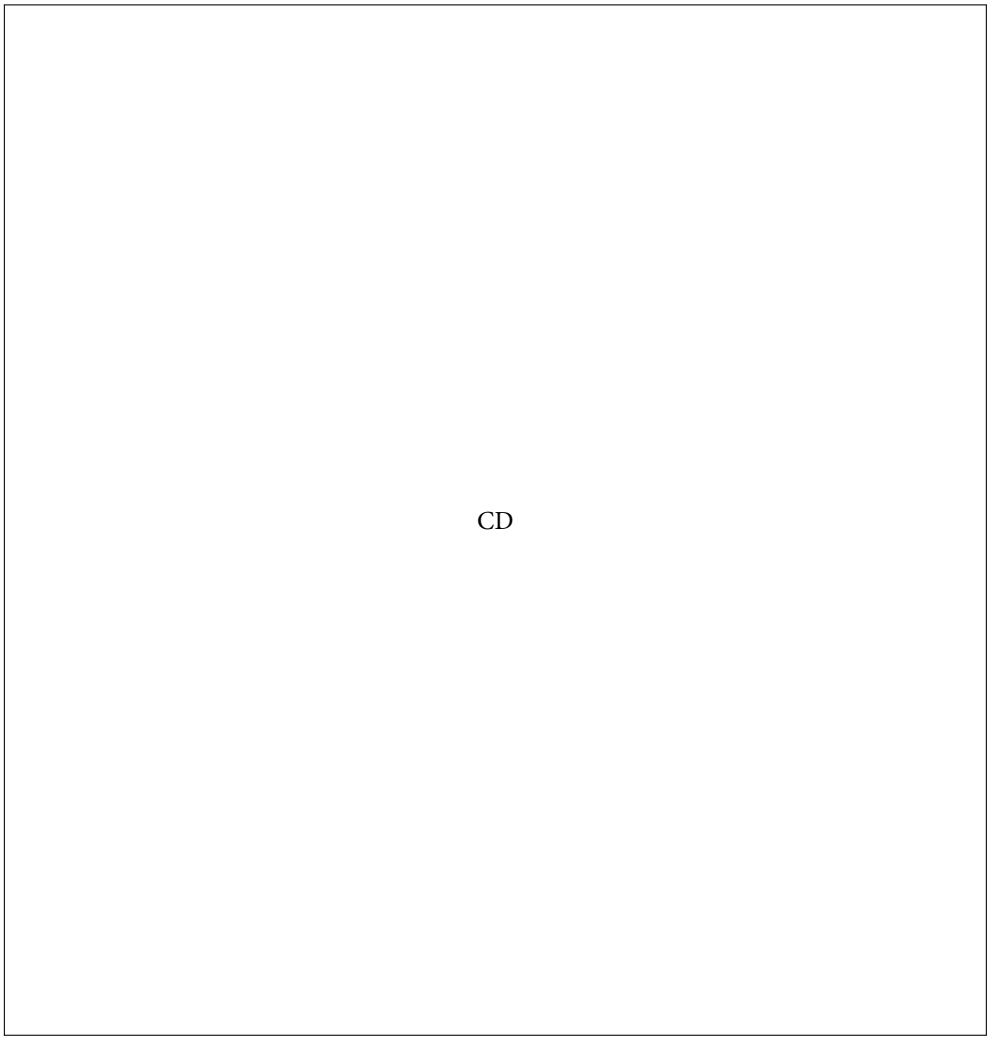


Abbildung A.1 CD

Im Folgenden werden die Ordner der CD, sowie deren Inhalte, in der obersten Ebene beschrieben.

- **cnf_gen**: Programme zur Erzeugung der konjunktiven Normalform und zur Analyse der von CryptoMiniSat gefundenen Klauseln. Durch einen Aufruf von „make“ werden sowohl CryptoMiniSat als auch alle anderen Programme kompiliert. CryptoMiniSat ist Voraussetzung für die in den anderen Ordnern enthaltenen Evaluationen.

- **espresso**: Heuristic Logic Minimizer der Berkeley Universität (siehe Abschnitt 2.3). Durch einen Aufruf von „make“ wird Espresso kompiliert und einen Berechnung der konjunktiven Normalformen für die im Ordner enthaltenen Gleichungen durchgeführt.

- **eval_capiman**: Konjunktive Normalform von Martin Maurer für den Vergleich in der Evaluation.

- **eval_cbmc**: Model Checker für C und C++ Programme (siehe Abschnitt 2.5) mit dem eine konjunktive Normalform aus einem C Programm erzeugt wird, um diese in der Evaluation zu vergleichen.

- **eval_initial**: Evaluation der in dieser Arbeit erstellen konjunktiven Normalform mit einer Initialwertberechnung (siehe Abschnitt 3.5.3).

- **eval_miter**: Evaluation der in dieser Arbeit erstellen konjunktiven Normalform mit einer Kollisionsberechnung (siehe Abschnitt 3.5.4).

- **eval_thesis**: Evaluation der in dieser Arbeit erstellen konjunktiven Normalform mit einer Urbildberechnung (siehe Abschnitt 3.5.2) für den direkten Vergleich mit CBMC und der Arbeit von Martin Maurer.

- **graphics**: Allgemeine Grafiken für die Verwendung in der Arbeit und der Präsentation.

- **presentation**: Folien für die Präsentation bei der Vorstellung und Verteidigung der Arbeit.

- **source**: In dieser Arbeit verwendete Programme und Dokumente.

- **thesis**: Quellcode dieses Dokuments der durch Eingabe von „rake“ kompiliert werden kann.

Anhang B

C-Programm zur Berechnung von SHA256

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

uint32_t sha_k[64] = {\
 0x428A2F98, 0x71374491, 0xB5C0FBCF, 0xE9B5DBA5, 0x3956C25B, 0x59F111F1, 0x923F82A4, 0xAB1C5ED5,\
 0xD807AA98, 0x12835B01, 0x243185BE, 0x550C7DC3, 0x72BE5D74, 0x80DEB1FE, 0x9BDC06A7, 0xC19BF174,\
 0xE49B69C1, 0xEFBE4786, 0x0FC19DC6, 0x240CA1CC, 0x2DE92C6F, 0x4A7484AA, 0x5CB0A9DC, 0x76F988DA,\
 0x983E5152, 0xA831C66D, 0xB00327C8, 0xBF597FC7, 0xC6E00BF3, 0xD5A79147, 0x06CA6351, 0x14292967,\
 0x27B70A85, 0x2E1B2138, 0x4D2C6DFC, 0x53380D13, 0x650A7354, 0x766A0ABB, 0x81C2C92E, 0x92722C85,\
 0xA2BFE8A1, 0xA81A664B, 0xC24B8B70, 0xC76C51A3, 0xD192E819, 0xD6990624, 0xF40E3585, 0x106AA070,\
 0x19A4C116, 0x1E376C08, 0x2748774C, 0x34B0BCB5, 0x391C0CB3, 0x4ED8AA4A, 0x5B9CCA4F, 0x682E6FF3,\
 0x748F82EE, 0x78A5636F, 0x84C87814, 0x8CC70208, 0x90BEFFFA, 0xA4506CEB, 0xBEF9A3F7, 0xC67178F2};

// process one chunk of a message, updating state (which after the last chunk is the hash)
void sha_processchunk(uint32_t *state, uint32_t *chunk) {
    uint32_t w[64], s0, s1;
    uint32_t a,b,c,d,e,f,g,h;
    uint32_t t1, t2, maj, ch, S0, S1;
    int n;

    // Read in chunk. When these 32bit words were read, they should have been taken as big endian.
    for (n = 0; n < 16; n++)
        w[n] = *(chunk + n);

    // Extend the sixteen 32-bit words into sixty-four 32-bit words:
    for (n = 16; n < 64; n++) {
        s0 = (w[n-15] >> 7 | w[n-15] << (32-7))^(w[n-15] >> 18 | w[n-15] << (32-18))^(w[n-15] >> 3);
        s1 = (w[n-2] >> 17 | w[n-2] << (32-17))^(w[n-2] >> 19 | w[n-2] << (32-19))^(w[n-2] >> 10);
        w[n] = w[n-16] + s0 + w[n-7] + s1;
```

```
    }

    for (n = 0; n < 64; n++) {
        w[n] += sha_k[n];
    }

    // Initialize hash value for this chunk:
    a = state[0];
    b = state[1];
    c = state[2];
    d = state[3];
    e = state[4];
    f = state[5];
    g = state[6];
    h = state[7];

    // Main loop:
    for (n = 0; n < 64; n++) {
        S0 = (a >> 2 | a << (32-2)) ^ (a >> 13 | a << (32-13)) ^ (a >> 22 | a << (32-22));
        maj = (a & b) ^ (a & c) ^ (b & c);
        t2 = S0 + maj;
        S1 = (e >> 6 | e << (32-6)) ^ (e >> 11 | e << (32-11)) ^ (e >> 25 | e << (32-25));
        ch = (e & f) ^ ((~e) & g);
        t1 = h + S1 + ch + w[n];

        h = g; g = f; f = e; e = d + t1;
        d = c; c = b; b = a; a = t1 + t2;
    }

    // Add this chunk's hash to result so far:
    state[0] += a;
    state[1] += b;
    state[2] += c;
    state[3] += d;
    state[4] += e;
    state[5] += f;
    state[6] += g;
    state[7] += h;
}

int main(int argc, char* argv[]) {
    uint32_t block[16] = {\
     0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000,\
```

```
     0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x80000000, 0x00000000, 0x000001A0};

#ifdef CBMC
    block[0] = nondet_uint();
    block[1] = nondet_uint();
    block[2] = nondet_uint();
    block[3] = nondet_uint();
    block[4] = nondet_uint();
    block[5] = nondet_uint();
    block[6] = nondet_uint();
    block[7] = nondet_uint();
    block[8] = nondet_uint();
    block[9] = nondet_uint();
    block[10] = nondet_uint();
    block[11] = nondet_uint();
    block[12] = nondet_uint();
#endif

    uint32_t result[8] = {\
     0x6A09E667, 0xBB67AE85, 0x3C6EF372, 0xA54FF53A, 0x510E527F, 0x9B05688C, 0x1F83D9AB, 0x5BE0CD19};

    // Process it.
    sha_processchunk(result, block);

#ifdef CBMC
    assert(
        result[0] != 0x6A09E667 &&
        result[1] != 0xBB67AE85 &&
        result[2] != 0x3C6EF372 &&
        result[3] != 0xA54FF53A &&
        result[4] != 0x510E527F &&
        result[5] != 0x9B05688C &&
        result[6] != 0x1F83D9AB &&
        result[7] != 0x5BE0CD19
    );
#endif

    int n;
    for (n = 0; n < 8; n++) printf("%08x ", result[n]);
    printf("\n");

    return 0;
}
```