Preimages for SHA-1

Submitted in fulfilment
of the requirements of the degree of

Doctor of Philosophy

of Rhodes University

Yusuf Moosa Motara

Grahamstown, South Africa
August 31, 2017

Abstract This research explores the problem of finding a preimage — an input that, when passed through a particular function, will result in a pre-specified output — for the compression function of the SHA-1 cryptographic hash. This problem is much more difficult than the problem of finding a collision for a hash function, and preimage attacks for very few popular hash functions are known.

The research begins by introducing the field and giving an overview of the existing work in the area. A thorough analysis of the compression function is made, resulting in alternative formulations for both parts of the function, and both statistical and theoretical tools to determine the difficulty of the SHA-1 preimage problem. Different representations (And-Inverter Graph, Binary Decision Diagram, Conjunctive Normal Form, Constraint Satisfaction form, and Disjunctive Normal Form) and associated tools to manipulate and/or analyse these representations are then applied and explored, and results are collected and interpreted.

In conclusion, the SHA-1 preimage problem remains unsolved and insoluble for the foreseeable future. The primary issue is one of efficient representation; despite a promising theoretical difficulty, both the diffusion characteristics and the depth of the tree stand in the way of efficient search. Despite this, the research served to confirm and quantify the difficulty of the problem both theoretically, using Schaefer's Theorem, and practically, in the context of different representations.

Acknowledgements

بِسْمِ اللَّهِ الرَّحْمَٰنِ الرَّحِيمِ

I gratefully acknowledge the assistance and support of my immediate and extended family; my supervisor (Barry V Irwin); my proofreaders; the cryptographic community at large; the Kresge Foundation; the teams and people behind F# and other amazing tools that I've used; and Rhodes University. And, of course, Sci-Hub, which removes barriers that are in the way of science.

This thesis has been a labour of labour. I've had to dive deeply into the waters of many, many different (but related!) subfields, and I can only hope that I've done them some justice in this work.

Contents

List of Figures

List of Tables

List of Examples

Glossary

***DD**	Decision Diagram variant
AC-n	Arc Consistency algorithm n
ADD	Algebraic Decision Diagram
AI	Artificial Intelligence
AIG	And-Inverter Graph
AIGER	And-Inverter Graph file format
AMD	Advanced Micro Devices
ANF	Algebraic Normal Form
ARX	And-, Rotation-, and XOR-using function
BAB	Branch-And-Bound search
BBDD	Biconditional Binary Decision Diagram
BDD	Binary Decision Diagram
BOOM-II	BOOlean Minimizer version 2
CBDD	Complemented reduced ordered Binary Decision Diagram
CDCL	Conflict-Driven Clause Learning
CNF	Conjunctive Normal Form
CPU	Central Processing Unit
CRHF	Collision-Resistant Hash Function
CROBDD	Complemented Reduced Ordered Binary Decision diagram
CSP	Constraint Satisfaction Problem
CUDD	Colorado University Decision Diagram package

DAG	Directed Acyclic Graph
DFS	Depth First Search
DIMACS	Centre for Discrete Mathematics and Theoretical Computer Science; also a format for CNF files that originates at DIMACS
DNF	Disjunctive Normal Form
DPLL	Davis-Putnam-Logemann-Loveland approach
DSPACE	Space requirement of an algorithm on a Deterministic Turing machine
FIPS	Federal Information Processing Standard
FRAIG	Functionally-Reduced And-Inverter Graph
GF(n)	Galois Field of order n
GNU	GNU's Not Unix
GPGPU	General-Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
IPsec	Internet Protocol security
L	Logarithmic complexity class
LDS	Least Discrepancy Search
LHS	Left-Hand Side
LUT	Look-Up Table
MDn	Message Digest Algorithm version n
MDD	Multi-valued Decision Diagram
MSB	Most Significant Bit
NIST	National Institute of Standards and Technology
NL	Nondeterministic Logarithmic complexity class
NP	Nondeterministic Polynomial complexity class
NSPACE	Space requirement of an algorithm on a Nondeterministic Turing machine
OBDD	Ordered Binary Decision Diagram
P	Polynomial complexity class
PC(n)	Propagation Criterion of degree n

PDAG	Propositional Directed Acyclic Graph
PFLOPS	10^{15} (i.e. Peta-scale) Floating-point OPerations per Second
PGP	Pretty Good Privacy
ROBDD	Reduced Ordered Binary Decision Diagram
RSA	Rivest-Shamir-Adleman
S/MIME	Secure/Multipurpose Internet Mail Extensions
SAC	Strict Avalanche Criterion
SAT	Boolean SATisfiability problem; sometimes means a SATisfiable outcome for such a problem
SHA-n	Secure Hash Algorithm version n
SLS	Stochastic Local Search approach
SOP	Sum-Of-Products
SSH	Secure SHell
SSL	Secure Sockets Layer
TLS	Transport Layer Security
UNSAT	UNSATisfiable boolean satisfaction problem
ZBDD	Zero-suppressed Binary Decision Diagram

Part I

Introductory material

Chapter 1

Introduction

A hash function takes a variable-sized input and produces a fixed-size output.

Example 1.1. *A simple hash function.* A dictionary or hashset data structure uses hash functions to access items by key in $\mathcal{O}(1)$ time. Such hash functions usually result in a 32-bit value as their output. A key is hashed, and the resultant 32-bit value is used to determine which "bucket" (out of many possible buckets) the desired item might be in. The bucket is then searched, and the item is returned.

The domain in Example 1.1 is of effectively infinite size since the hash function can accept very large keys, and the range has size 2^{32} since that is the number of unique values that could possibly be generated. Any input that results in a particular output is called a *preimage* of that output. This work is concerned with analysing and exploring a particular cryptographic hash, SHA-1 (**S**ecure **H**ash **A**lgorithm version **1**) (NIST, 1995), within the context of finding preimages.

A cryptographic hash function is a hash function which is collision-resistant and preimage-resistant; the "resistant" suffix denotes computational infeasibility. By "computational infeasibility", it is practically meant that the power to obtain such a preimage — despite the fact that such a preimage may be known to exist — is beyond the capabilities of any entity, no matter how much computing power they may bring to bear. A *collision* occurs when two inputs result in the same output. In the data structures used in Example 1.1, collisions occur in the normal course of operation and a linear search is typically used to find the correct item in a bucket. As long as the hash values are uniformly distributed, the linear search time is negligible and lookup is therefore still $\mathcal{O}(1)$. However, for a cryptographic hash function, any collision is extremely serious and it should not be possible to deliberately create two inputs which collide; in other words, a cryptographic hash should present the façade of being 1-to-1. More formal definitions of resistance and computational infeasibility will be provided later.

Such computational infeasibility is achieved by a process of *confusion* and *diffusion*. The importance of these was first described by Shannon, who regarded them as methods "for frustrating a statistical analysis" (Shannon, 1949a). On the subject of diffusion, Shannon (1949a, p. 708-9) says:

> In the method of diffusion the statistical structure of M which leads to its redundancy is "dissipated" into long range statistics—i.e., into statistical structure involving long combinations of letters in the cryptogram. The effect here is that the enemy must intercept a tremendous amount of material to tie down this structure, since the structure is evident only in blocks of very small individual probability. Furthermore, even when he has sufficient material, the analytical work required is much greater since the redundancy has been diffused over a large number of individual statistics.

One can think of this as the relationship between input and output being obscured; each bit of the input affects each bit of the output, and although a relationship *does* exist, it is difficult to unravel what that relationship is. As for confusion, Shannon (1949a, p. 709) explains that:

> The method of *confusion* is to make the relation between the simple statistics of E and the simple description of K a very complex and involved one. In the case of simple substitution, it is easy to describe the limitation of K imposed by the letter frequencies of E. If the connection is very involved and confused the enemy may still be able to evaluate a statistic S_1, say, which limits the key to a region of the key space. This limitation, however, is to some complex region R in the space, perhaps "folded ever" [sic] many times, and he has a difficult time making use of it. A second statistic S_2 limits K still further to R_2, hence it lies in the intersection region; but this does not help much because it is so difficult to determine just what the intersection is.

Although confusion and diffusion work towards the same goal, they do so in different ways. Using the principle of confusion, transformations which only make sense in a *local* context are used to eventually produce a final result. Successfully "undoing" any local transformation is of limited use since the result obtained is difficult to relate to the original input; and local transformations which build upon other local transformations exacerbate this issue.

Example 1.2. *The importance of preimage resistance.* When passwords are stored on a remote system, they are usually hashed[a]. When the system needs to check whether an input is the correct password, it hashes the input and compares the result to the stored n-bit hash value. If the hash values match, then the input was the correct password. An attacker who obtains the database of passwords would need to find a preimage for each of the stored passwords; assuming that the passwords are uniformly distributed bit sequences (in reality, they are not), this would take approximately 2^n guesses *if* the hash function used was preimage-resistant. If n was large enough (if $n = 50$, then $2^{50} \approx 1 \times 10^{15}$, for example), the work factor required would be insurmountable and it does an attacker little good to have the hashed passwords. If the hash function used was *not* preimage-resistant then obtaining the hash values is akin to obtaining the passwords and there would be little point in hashing the passwords in the first place. This shows that preimage-resistance is not only of academic value.

[a]In fact, they are usually *salted* and hashed. This detail adds nothing to the example and has been elided.

Technically speaking, a preimage is either a *preimage* or a *second-preimage*. The former is the original input that resulted in a particular output, and the latter is any input which results in a particular output.

Example 1.3. *The importance of second-preimage resistance.* Aragorn discovers that recently-terminated employee Bilbo has absconded with important source code. Fortunately, the logs of the version control system provide the cryptographic hashes of the source code that was taken. Aragorn therefore demands that Bilbo hand over the source code. Importantly, Aragorn can verify that the data that Bilbo provides is correct since Bilbo, despite having the original code in his possession, is unable to generate data that will result in the same hash output. If the hash function used was *not* second-preimage resistant, then Bilbo would be able to generate nonsensical data, supply it to Aragorn, and insist that the data is correct. Aragorn would be unable to contest this assertion.

Similar examples can be constructed to demonstrate the value of collision-resistance. Cryptographic hashes are used in many areas — by banks, commercial and open-source software developers, governments, grassroots organizations, certificate authorities, browser vendors, and many others — and their preimage-resistance (and collision-resistance) is extremely important to the functioning of the modern world.

An "attack" on a cryptographic hash function is any method or algorithm that reduces the difficulty of obtaining a collision, preimage, or second-preimage. The hash function that is the subject of such a successful attack is considered to be either "weakened" or "broken", depending on the reduction in difficulty. Attacks may exploit oversights on the part of the hash function designers (e.g. Leurent (2008)), or they may take advantage of advances in technology such as General-Purpose Graphics Processing Units to perform operations faster than would be expected (e.g. Grechnikov (2010); Adinetz and Grechnikov (2012)), or they may use recent mathematical results to remove impediments that the algorithm places in the way; any method is permissible.

1.1 SHA-1

The SHA-1 hash (NIST, 1995) is a well-known cryptographic hash function which generates a 160-bit hash value. It is the successor to the equally well-known and -used MD5 (Rivest, 1992b) cryptographic hash function which generated a 128-bit hash value. SHA-1 was designed by the National Security Agency of the United States of America and published in 1995 as National Institute of Standards and Technology (NIST) Federal Information Processing Standard 180-1. Since this standardisation, it has become widely used across the world; Wikipedia (2014) lists some of the broad areas where it has been used:

> SHA-1 forms part of several widely used security applications and protocols, including TLS and SSL, PGP, SSH, S/MIME, and IPsec. [...] SHA-1 hashing is also used in distributed revision control systems like Git, Mercurial, and Monotone to identify revisions, and to detect data corruption or tampering. The algorithm has also been used on Nintendo's Wii gaming console for signature verification when booting [...].

Both the hash function and the problem domain of preimage-resistance are worthwhile research topics. The hash function itself is well-specified and long-standing, having been formally promulgated as an official standard of the United States government more than 20 years ago (at the time of this writing). Many attempts were made to find weaknesses in the hashing algorithm before, during, and after the standardisation process (Cryptographic Technology Group, 2016); however, to date, there has been no successful attack against the preimage-resistance of the hash function. Therefore, it is exceptionally unlikely that there is an obvious oversight on the part of the hash designers that could be exploited.

The rationale behind the design of the SHA-1 hash has never been made clear, nor are the security guarantees that it makes backed up by any formal proof of difficulty. Why, for example, does the hash function progress over 80 "rounds" instead of 40, or 60, or 200? Why are particular three-input boolean functions used? What degree of confusion and diffusion does the hash provide? Its ancestry is clear: it takes many cues from the MD5 (**M**essage **D**igest version **5**) algorithm (and predecessors). However, the security guarantees of those algorithms were similarly not backed up by any proof. The uncertainty around the design choices made during the creation of the hash algorithm makes it an intriguing subject to study.

Preimage research itself is an interesting area since most research into hash functions has focused on collisions instead (see Chapter 3). That focus is easy to understand since preimage research is typically more difficult than collision research. The output of a preimage is specified before the search for an input begins, and this severely restricts the set of acceptable solutions; by contrast, *any* two inputs which result in the same output constitute a valid collision, and neither inputs nor outputs need to be fixed before a collision search begins. Cryptography is widely considered to be a difficult and complex field and, since successful attacks are already so difficult to create, it makes sense to focus on an area where there is a greater chance of success.

1.2 Contribution of this work

The research described in this work does not follow directly from previous research into SHA-1 (see Chapter 3), which has typically fallen into two broad categories (attacks and optimisations), though it does intersect with that research at a number of points. The main contribution of this work is an in-depth exploration and analysis of the SHA-1 *compression function*, focusing on the problem of finding preimages. The "compression function" can be thought of as the core of a hash function; a more formal definition is provided in Section 2.2. More specifically, the contribution can be enumerated as:

1. an in-depth analysis of, and alternative valid formulations of, the SHA-1 compression function and its components (Chapter 4);
2. a statistical analysis of diffusion characteristics (Chapter 5);
3. multiple representations of the SHA-1 compression function, with preimage-focused practical experiments (Chapters 7, 8, 9, 10, and 11);
4. a reflection on representations and SHA-1 design choices (Chapter 12).

Much research is naturally written with members of the field in mind. Unfortunately, in the case of a relatively "niche" field such as cryptography, this decreases the accessibility of the research for those not in the field. Works such as Menezes, Van Oorschot, and Vanstone (1996), Ferguson and Schneier (2003), and Paar and Pelzl (2009) make it easier to gain entry to the cryptographic field, but none of these explore hash compression functions in great detail. Papers on hash function security (see, for example, Rjaško (2011) or Rogaway and Shrimpton (2009)) typically examine the structures within which a compression function is embedded, and make assumptions about the compression functions themselves. The reason is simple: compression functions are not well-understood in general, and the reasons why a compression function may be collision-resistant and preimage-resistant are not well-understood either. For example, the most recent of the accessible works cited (Paar and Pelzl, 2009) devotes approximately five pages to discussing SHA-1 – and only two of these discuss the compression function itself. Very little space (if any) is given to preimage attacks. Note that the works cited are well-regarded in the field and this should therefore not be taken to mean that they are in any way inadequate; instead, the example shows that little can be said when little has been said.

This work aims to reduce this problem by providing accessible preimage-focused analysis, discussion, and results. Note that the determination of the "hardness" of SHA-1 is not a primary goal of this work, though some theoretical and experimental analysis in this direction is provided in Chapter 5. The expected audience is comprised of three main groups: computer scientists, software developers, and interested members of the public. The knowledge that is assumed is therefore minimal: a reader should have some knowledge of basic computer programming and related terminology (e.g. "hash table", "algorithm", "indexing"), some knowledge of basic computer theory (e.g. "boolean function", "Turing machine", "time complexity"), some knowledge of basic algebra and sets (e.g. "coefficient", "disjoint sets"), but no knowledge of more specialist mathematical notation is necessary. Chapters 2 and 3 exist to bridge the gap between computer scientists and cryptographers, examples are provided throughout this work, and explanations are simplified where possible.

This thesis focuses on preimages at the expense of second-preimages, with the practical difference (for larger input sizes) being that the information that is assumed to be available to an attacker is only the final hash output, and sometimes the size of the input.

In terms of real-world impact, it must be understood that SHA-1 is a dedicated hash function: it is not based on any existing encryption algorithm, and no widely-used encryption algorithm is based upon it. Research into it may not result in general attacks that are applicable to the broader cryptographic field. Such research will also become increasingly irrelevant as time goes on since the next secure hash standard, SHA-3 / KECCAK (NIST, 2015), was conceptualized and developed independently of SHA-1 and uses very dissimilar compression function. Nevertheless, there are many SHA-1 hashes in the world, and these hashes will continue to exist for the foreseeable future. Preimage research into SHA-1 is useful in this context.

1.3 Structure

This thesis is broken up into four parts, I through IV. Introductory and prerequisite material is covered in Part I, which comprises two chapters; a reader who is familiar with the SHA-1 hash and the notation and terminology of the various fields that are discussed in this thesis may nevertheless be interested in Section 2.4, which describes the mechanics of the SHA-1 hash, and Section 2.5, which describes thesis-specific notation.

Part II is about analysing and understanding the SHA-1 hash. The three chapters of this section attempt to answer the following questions:

- What does the existing literature say about hash functions, the SHA-1 hash, and preimage research?
- What does our own analysis of SHA-1 reveal?
- How difficult is the problem, and how may it best be approached?

Part III presents different representations of the SHA-1 compression function, along with practical experiments to determine how suitable a particular representation is in the context of addressing the preimage problem. Lastly, Part IV presents some discussion and analysis, and summarises some of the lessons learned during the course of the research.

Chapter 2

Prerequisites

This chapter deals with three things: mathematical concepts and notation, essential background and terminology related to cryptographic hashes, and the definition of the SHA-1 hash. Additional notations and definitions which relate to fields other than cryptography, or which are not general mathematical notations, are introduced in the chapters where they are first used. Note that some items are defined less generally than is necessary, when a less general specification happens to be more useful for this work or in the field of cryptography; for example, the "multiplication" ($*$) and "addition" ($+$) operations over a field have been particularized to refer exclusively to the $\wedge$ and $\oplus$ operations.

The well-known Merkle-Dåmgard structure within which the SHA-1 compression function is embedded is also explained here. Although this structure is out of scope for this thesis, readers will inevitably find it explained in the existing literature and it is therefore worthwhile to make the link between compression function and embedding structure clear.

2.1 Notation and mathematical concepts

The notation for the usual boolean operators is $\wedge$ ("and"), $\vee$ ("or"), $\neg$ ("not"), and $\oplus$ ("exclusive or").

A *Galois field of order 2* is the set of boolean values, denoted GF(2) or $\mathbb{Z}_2$, containing elements $\{0, 1\}$ that can be combined using the binary operators $\oplus$ and $\wedge$ such that certain conditions hold, including:

- $\oplus$ and $\wedge$ are associative ($(x \oplus y) \oplus z = x \oplus (y \oplus z)$, and similarly for $\wedge$) and commutative ($x \oplus y = y \oplus x$, and similarly for $\wedge$)
- $\wedge$ distributes over $\oplus$ ($x \wedge (y \oplus z) = (x \wedge y) \oplus (x \wedge z)$), but the converse is not true ($x \oplus (y \wedge z) \neq (x \oplus y) \wedge (x \oplus z)$).
- Identity elements 0 and 1 exist for $\oplus$ and $\wedge$ respectively, such that $x \oplus 0 = x$ and $x \wedge 1 = x$.

A *vector* of length l is an ordered sequence of elements selected from a given set, and is denoted using set notation and a superscript. A *boolean vector* of length l may be denoted by either $\mathbb{Z}_2^l$ or a two-element set and a superscript. A left-rotation of a vector (or unsigned integer) x by n positions is indicated by the notation $x \hookleftarrow n$, and a right-rotation is denoted by $x \hookrightarrow n$. The *Hamming weight* of a boolean vector is the number of "1" values in the vector, and the *Hamming distance* between two boolean vectors is the number of positions where the values in the vectors disagree.

For concision, the boolean vector may be written in hexadecimal; for readability, it may be written in binary. An "0x" prefix may precede a hexadecimal number and a "b" suffix may succeed a binary number when the base of the number is ambiguous.

Example 2.1. *Notation: vectors and rotation.* 13663, 66313 and 63116 are all different $\{3, 1, 6\}^5$ vectors; $13663 \hookleftarrow 2 = 66313$ and $66313 \hookrightarrow 2 = 13663$. An unsigned 32-bit integer can be regarded as the boolean vector $\{0, 1\}^{32}$.

A set is usually denoted by a capital letter (though not all capital letters denote sets). It may be defined extensionally by giving the elements (e.g., $X = \{9, 5, 2\}$) or intensionally by using a domain and rule(s) for set membership (e.g., $X = \{x \in \mathbb{N} : x \mod 2 = 0\}$ denotes all even natural numbers).

A function with multiple inputs can be understood as operating over variables with different names, or operating over a boolean vector. When the latter abstraction is used, the fact that the input is a vector is denoted by an underline and each component of the vector (starting from the left) is subscripted.

Example 2.2. *Truth table formats.* The truth tables below are equivalent; one uses multiple boolean variables and the other uses vector notation.

x	y	z	$a \oplus (b \wedge c)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

$\underline{x}$	$x_0 \oplus (x_1 \wedge x_2)$
000	0
001	0
010	0
011	1
100	1
101	1
110	1
111	0

The size of a vector (also called the *length*) is the number of elements in the sequence; the size of a set is the number of elements it contains; and the size of a graph is the number of nodes in the graph. In all cases, the notation $|X|$ is used to denote the size of the vector/set/graph X.

A function f which has a domain X and a range Y is denoted by $f : X \to Y$, assuming $|Y| \geq |X|$. The inverse of a function f — that is, a function maps f's range to its domain — is denoted by $f^{-1} : Y \to X$. Application of a function f to a value v is denoted by $f(v)$. A tuple is an ordered group of possibly-disparate elements, and is denoted by the elements being shown in parentheses. The number of elements in a tuple (also called the *arity*) is sometimes given as a prefix, viz. “2-tuple” or “5-tuple”. The arguments to a function are also given in parentheses; context differentiates between a tuple and arguments.

A *functionally complete* set of boolean operators can be used to represent any boolean function, no matter how complex. Where a large binary symbol is used, the meaning is the same as for when a Σ is used, with the modification that the specified binary operation should be used instead of addition.

The word “linear” is unavoidably used to mean three different, but related, things in this work. A “linear” relationship is one in which the relationship between two variables can be plotted as a straight line — in other words, the variables are directly proportional to each other. A “linear” boolean function has a particular definition that is given in Section 4.2.2. Lastly, a “linear” system of equations is one which contains multiple equations, of degree ≤ 1, over the same set of variables.

The probability of an event lies in the range 0.0 to 1.0, and is denoted by the syntax “Pr”.

Sometimes it will be necessary to choose all combinations of k elements from a set of n elements. The number of such combinations that exist is denoted by $\binom{n}{k}$.

2.2 Cryptographic terminology

This section briefly covers some cryptographic terminology that is common in the field; for fuller explanations, see (for example) Menezes *et al.* (1996), Ferguson and Schneier (2003), or Paar and Pelzl (2009). Throughout this section, assume that a hash function $h : \mathbb{Z}_2^n \to \mathbb{Z}_2^m$ exists, and further assume that the range is uniformly distributed. The input to such a hash function is sometimes called the *message*.

Given $c \in \mathbb{Z}_2^m$ generated via $h(a)$, if it is computationally infeasible to find a b such that $h(b) = c$, then h is said to be *one-way*. The value b is called a *second-preimage* of c if $a \neq b$, and a *preimage* if $a = b$. If all hash values are equally likely to be obtained, then the chance of finding a particular one using a random input is $\frac{1}{2^m}$ (often written as 2^{-m}); therefore, approximately 2^m values must be tried before a successful match is found.

A *collision* is found for values d and e if $h(d) = h(e)$. A collision can be expected to be found after evaluating $\approx 2^{\frac{m}{2}}$ values; intuitively, as one approaches the $2^{\frac{m}{2}}$th value without having found any collision, it becomes increasingly likely that the next hash value will be one that has already been seen.

Computational infeasibility is described in terms of the number of evaluations of a hash function that would be required for a collision or preimage to be found; this has also been called the *work factor*. This description of infeasibility allows researchers to consider the number of evaluations independent of advancing technology, and therefore makes it easier to compare the difficulty of finding collisions or preimages across time and technologies. The strategy of attempting different combinations of input values until a preimage or collision is found is called a *brute-force* strategy; in recent times, GPUs have been particularly helpful in this regard. An *attack* on a hash function is any algorithm or approach that obtains the desired preimage/collision after doing less work than brute-force would require. The *resistance* of a cryptographic hash function is correlated with the computational infeasibility of attacking a particular property of that hash function.

The *strength* of a cryptographic hash function is expressed as the difficulty of attacking its least-difficult property, collision-resistance. In general, collisions are necessarily easier to find than preimages since $2^{\frac{m}{2}} < 2^m$; however, collisions are only guaranteed to exist when

$n > m$. The difficulty of obtaining a useful preimage attack can be seen by observing that collisions for the 128-bit MD4 cryptographic hash function (Rivest, 1992a) — a subject of study since 1990 — can be found with a trivial work factor of approximately 2^8 (Wang, Lai, Feng, Chen, and Yu, 2005a), but preimages require a work factor of $\approx 2^{78}$ (Guo, Ling, Rechberger, and Wang, 2010). The latter work factor is far below the theoretically optimal 2^{128} and, although it is within the bounds of possibility for well-funded entities, nevertheless demonstrates the difficulty of finding preimages as opposed to collisions.

The hash function relies upon repeated invocations of a *compression function* to create a final output. The amount of input data that is accepted by a compression function is called a *block*. The compression function transforms a block by looping over it for a number of *rounds*, which are occasionally called *steps* in the literature — although the term "steps" can also refer to small subdivisions of a round, or of an attacking algorithm. If (as in this work) only single-block inputs are used, then the terms "block" and "message" are almost interchangeable: the difference is that a "message" includes additional data that is relevant to the overall hash structure (see Section 2.3).

A *nothing-up-my-sleeve* value is constructed so that an observer can be confident that the author of an algorithm has not chosen that value due to any inherent property of the value. Such values are exceptionally unlikely to have any "hidden" properties that lead to a weakness in the algorithm under particular circumstances. Furthermore, the use of a nothing-up-my-sleeve value implies that the properties of the algorithm are not dependent upon any particular specially-chosen value. Examples of nothing-up-my-sleeve values are the first n digits of π or e, the sequence $\{1, 2, 3, 4, 5, .., n\}$, and the first n prime numbers.

Most hash functions, including SHA-1, operate over *words*. A word is a 32-bit unsigned integer.

2.3 Merkle-Dåmgard structure

The core of a cryptographic hash function is a *compression function* $C : \mathbb{Z}_2^m \mapsto \mathbb{Z}_2^n$. It is typical for the relation $n < m$ to hold, which means that the range contains fewer bits that the domain — hence the use of the term "compression". Assume that a hypothetical hash function $H(a) = c$, which uses the compression function $C(x) = y$ internally as its compression function, exists. The hash function would apply the compression function to an input to generate the hash output.

The size of the input cannot be predicted in advance, yet H must accept any input size. A simple way of solving this problem is by truncating data that is too long or padding data that is too short. Truncation allows multiple "long" messages to have the same hash value, thus making collisions trivial to obtain; padding allows multiple short messages to have the same hash value after being padded, leading to the same issue. The Merkle-Dåmgard structure is a way of ensuring that any input size can be handled without introducing collisions.

By tweaking the definition of C, the situation can be improved. If C is defined as $C(x, v) = y, |v| \geq |y|$ instead of $C(x) = y$, it would be possible to pass along an *chaining value* (i.e. v) to the compression function in addition to the data x. The first chaining value is called an *initialization vector* and would be pre-specified by the algorithm; each subsequent chaining value is derived from the most-recently-generated output y. Since the chaining value is used by C during the calculation of the output, it forms a "link" to previous blocks of data which have been calculated. The H function can now handle arbitrarily-large inputs with the output being dependent upon all of the blocks of the input.

In a broader sense, a cryptographic hashing algorithm can be viewed as an encryption algorithm. During encryption, a plaintext a is converted into a ciphertext b through the use of a key k and an encryption function: $E(k, a) = b$. During hashing, a plaintext a is converted into a ciphertext b through a hashing function $H(a) = b$ which uses, internally, a compression function $C(x, v) = y$. Notice the similarity between E and C: it is due to this similarity that it is possible to regard a compression function as a function which encrypts the plaintext v using the key x. The difficulty of finding a preimage can be restated as the difficulty of finding a suitable encryption key, given a ciphertext; finding a second-preimage is equivalent to finding a suitable encryption key, given both ciphertext and plaintext. Viewing the algorithm (and, specifically, the compression function) in this light may make it easier to use results from the broader field of cryptography.

The SHA-1 hash is constructed using the Merkle-Dåmgard paradigm (Merkle, 1979; Gauravaram, Millan, and Nieto, 2005), which means that it consists of padding, chunking, and compression stages; see Figure 2.1.

unfamili

Padding begins by appending a single 1-bit to the input data, followed by a series of zero or more 0-bits, and ending with 64 bits that encode the length-in-bits of the input data. The number of 0-bits appended is the smallest number necessary for the entire padded message, including the length value, to end on a 512-bit boundary.

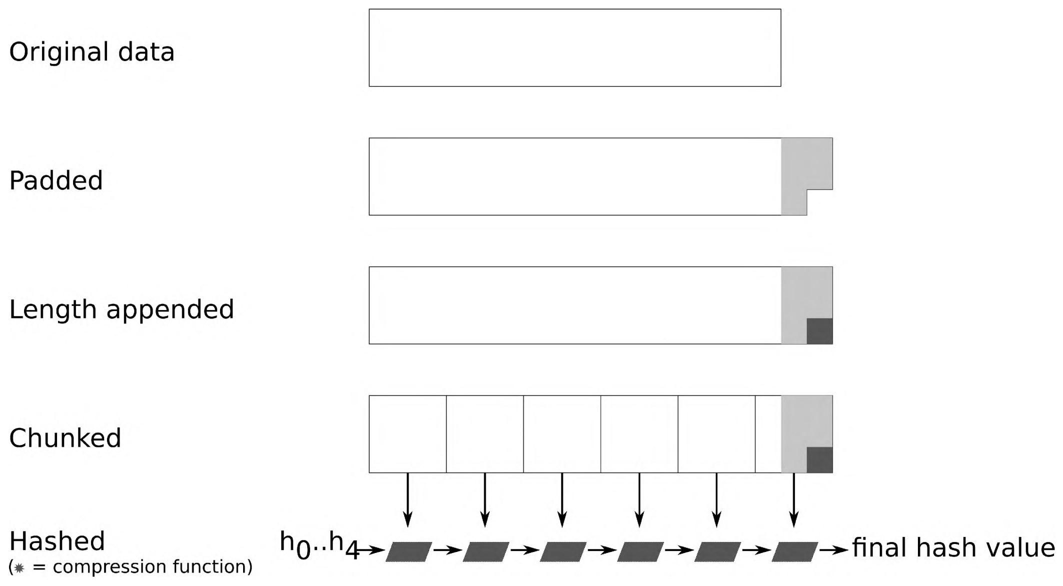

Figure 2.1: SHA-1 algorithm overview

Chunking then breaks the message into 512-bit blocks: $m_0, m_1 ... m_n$. Due to padding, there are always a discrete number of 512-bit blocks. The compression function — which is the part of the SHA-1 algorithm that this work will be looking at — is applied to the tuple (m_i, j_i), where j_i is 160 bits of data, and this results in some output o_i. o_i is added to j_i to obtain the final output of the compression function (this is called Davies-Meyer construction (Winternitz, 1984)), and the final output is used as j_{i+1} or, if all blocks are exhausted, as the final hash value.

The lengths of j and o are necessarily the same. The initial input j_0 is the nothing-up-my-sleeve constants in Table 2.1, as defined by the FIPS 180-1 standard (NIST, 1995). These constants, in little-endian[1] hexadecimal notation, are `0123456789abcdef fedcba9876543210 f0e1d2c3`. This follows a very obvious sequence of values in ascending order, values in descending order, and then interleaved descending/ascending values. If another constant were to be added, it would complete the interleaved descending/ascending sequence and be (in little-endian hexadecimal notation) `b4a59687`.

Recall that a collision occurs if two inputs are found which result in the same output. Collisions are theoretically easier to find than preimages, since *any* output is acceptable — as long as there are > 1 inputs which result in that output — whereas in the case of preimages, a particular output is desired. Merkle-Dåmgard hash functions are guaranteed to be collision-resistant if the compression function used is collision-resistant (Damgård, 1989; Merkle, 1989).

[1]A little-endian word stores the least-significant byte first

Table 2.1: Initialisation constants for SHA-1

Name	Value
a	0x67452301
b	0xefcdab89
c	0x98badcfe
d	0x10325476
e	0xc3d2e1f0

Assuming that the output has been generated via application of the compression function, it is true that a preimage must exist. The pigeonhole principle (Brualdi, 2012) states that if n objects are distributed among k compartments where $n > k$, then at least one compartment must contain more than one object. Similarly, as long as the length of the message is greater than the length of the output, it is certain that more than one preimage exists. It is therefore the case that, without assuming a valid generation process but under the assumption of uniformly distributed output, more than one preimage should exist for any particular output hash within the maximum length of 447 input bits that constitutes a single "block" of SHA-1. The preimage problem can therefore be studied exclusively with reference to the compression function and without consideration of the embedding Merkle-Dåmgard structure.

Unless otherwise stated, the single blocks used in this thesis are always valid SHA-1 inputs. This means that the last two words are always set to be appropriate values, and a terminator bit is inserted just after the end of the input data. Note that this is not the case with most preimage research such as De Canniere and Rechberger (2008); Aoki and Sasaki (2009); Knellwolf and Khovratovich (2012); Espitau, Fouque, and Karpman (2015).

2.4 Compression function mechanics

This section is purely descriptive and explanatory, and based on the FIPS 180-1 standard which defines the SHA-1 algorithm; see (NIST, 1995). Section 2.3 provided an overview of the SHA-1 algorithm as a whole, but omitted any description of how the compression function converts 672 input bits into 160 output bits. This section explains the compression function in detail.

The SHA-1 compression function occurs as two separate phases. The first phase takes a set of 16 32-bit words and combines parts of different words to produce a further 64 32-bit words, for a total of $16 + 64 = 80$ words. This phase is called *message expansion* since it "expands" 16 words into 80 words. The second phase makes use of addition, rotation, and logical functions ($\wedge$, $\vee$, $\neg$, and $\oplus$), applied over the course of 80 rounds, to turn these 80 words into a 5-word (160-bit) output. There is no agreed-upon name for the second phase. The nonsense-word *spliffling* has been coined as a way to refer to this phase easily: it has been chosen as a memorable and readable combination of syllables that avoids any confusion with terms that already exist.

Message-expansion uses the 16 words of an input block to create an additional 64 words, for a total of $16 + 64 = 80$ words. We refer to the initial 16 words as *data-words* and the rest of the 80 words as *expansion-words*. Each of these words is used in one of the 80 spliffling rounds described in Section 2.4. To obtain expansion-word $w_{r\geq 16}$, message-expansion XORs[2] together w_{r-3}, w_{r-8}, w_{r-14}, and w_{r-16}, and left-rotates the result by 1:

$$w_{r\geq 16} = (w_{r-3} \oplus w_{r-8} \oplus w_{r-14} \oplus w_{r-16}) \hookleftarrow 1$$

In most implementations of the SHA-1 compression function, if w_r is itself an expansion-word, then it must be calculated before $w_{r'>r}$ can be calculated.

Spliffling consists of 80 rounds. It has been represented in pseudo-code form as Algorithm 2.1.

There are 80 rounds, just as there are 80 words (after message-expansion). Each round utilises the corresponding word as input; after every 20 rounds, the function that produces f is changed, as is the constant k. The k-constants are nothing-up-my-sleeve values: they are the first 32 bits of the binary representations of $\sqrt{2}$, $\sqrt{3}$, $\sqrt{5}$, and $\sqrt{10}$ respectively.

Note that spliffling, as described by Algorithm 2.1, does not include the Davies-Meyer construction step (Winternitz, 1984). This step would add the initial (a,b,c,d,e) values to the final (a,b,c,d,e) values returned from spliffling. The omission is made for the sake of simplicity: the step is only relevant within the larger scheme of the hash. For inputs ≤ 447 bits, this step is irrelevant: the final values of $a..e$ can be obtained easily by subtracting the initialization constants (Table 2.1) from the appropriate words of a hash after the Davies-Meyer construction step (Winternitz, 1984); see Section 4.4 for details.

[2]"Exclusive-Or"s. We use the shortened version of various operations as verbs to avoid awkward constructions such as "the message-expansion process applies the exclusive-or operation to...".

Algorithm 2.1 "Spliffling"

Require:
w is an 80-element message-expansion array
a, b, c, d, e are unsigned 32-bit words

Ensure:
A modified 5-tuple (a, b, c, d, e) representing spliffling output

```
function SPLIFFLE($a, b, c, d, e, w$)
    for $i \leftarrow 0..79$ do
        if $i < 20$ then
            $f \leftarrow (b \wedge c) \vee (\neg b \wedge d)$        ▷ "choice" function
            $k \leftarrow$ 0x5a827999
        elif $i < 40$ then
            $f \leftarrow b \oplus c \oplus d$        ▷ "parity"/"minority" function
            $k \leftarrow$ 0x6ed9eba1
        elif $i < 60$ then
            $f \leftarrow (b \wedge c) \vee (b \wedge d) \vee (c \wedge d)$        ▷ "majority" function
            $k \leftarrow$ 0x8f1bbcdc
        else
            $f \leftarrow b \oplus c \oplus d$
            $k \leftarrow$ 0xca62c1d6
        $temp \leftarrow (a \hookleftarrow 5) + f + e + k + w_i$
        $e \leftarrow d$
        $d \leftarrow c$
        $c \leftarrow b \hookleftarrow 30$
        $b \leftarrow a$
        $a \leftarrow temp$
    return $(a, b, c, d, e)$
end function
```

For inputs > 447 bits, another invocation of the compression function is necessary. However, this invocation places the problem out of the scope of this thesis, which is content to examine the compression function in isolation.

2.4.1 f-functions

The *choice*, *parity*, and *majority* boolean functions are used during the spliffling phase:

- $\mathit{choice}(b, c, d) = (b \wedge c) \vee (\neg b \wedge d)$
- $\mathit{parity}(b, c, d) = b \oplus c \oplus d$
- $\mathit{majority}(b, c, d) = (b \wedge c) \vee (b \wedge d) \vee (c \wedge d)$

Table 2.2: Truth tables for choice, parity, and majority functions

b	c	d	choice	parity	majority
0	0	0	0	0	0
0	0	1	1	1	0
0	1	0	0	1	0
0	1	1	1	0	1
1	0	0	0	1	0
1	0	1	0	0	1
1	1	0	1	0	1
1	1	1	1	1	1

Each function takes three inputs, called b, c, and d in this section. The choice function is so named because b acts as a “chooser” between c and d: when b is true, then the value of c is output; otherwise, the value of d is output. For the same reason, the function is sometimes called the “if”, “if-then-else”, or “ite” function. The parity function outputs 1 when an odd number of the inputs are true, and 0 otherwise; it is sometimes called the “minority” function, because it always chooses the value that is least-represented among the three inputs. The majority function outputs 1 if two (or more) of the inputs are true, and outputs 0 otherwise.

2.5 Thesis-specific notation

Now that the mechanics of SHA-1 have been discussed, additional notation that is specific to the hash compression function and this thesis can be introduced.

All words are 32-bit big-endian and unsigned. The most significant bit of a word is its 0^{th} bit, and the least significant bit is its 31^{st} bit. Certain variable names refer to certain words, when not given any other meaning:

- a refers to words of the internal state during hash computation;
- f refers to the f-word, when used without parentheses, and the f-function when used with parentheses;
- k refers to the round constant called "k" in Algorithm 2.1;
- v refers to bits related to the carry of addition operations; and
- w refers to words from the message and their expansion.

When necessary, a distinction is made between the first 16 user-supplied words (called *data-words*) and the last 64 words generated via message expansion (called *expansion-words*). A similar distinction can be drawn at the level of bits, leading to the similar terms *data-bit* and *expansion-bit*. An unsigned 32-bit value x is subscripted with a number to identify the round that it participates in; where the round is not specific, the variable r is used to denote any round. Rounds are zero-indexed. Similarly, the variable i refers to a zero-indexed bit-position within a word. All calculations involving i variables or bit positions are assumed to occur (mod 32). A word is superscripted to denote a bit position. Subscripts and superscripts may indicate relations (such as "all rounds greater than 22") using the appropriate mathematical notation.

Whenever the size of the SHA-1 input is limited to a particular number of bits, the symbol ω is used to denote the number of unfixed bits. These bits always occur at the "start" of the input; in other words, there are never any unfixed bits that follow fixed bits. Each unfixed bit is sometimes called a *variable*.

"Onwards", "upwards", and similar terms mean "in the direction of propagation"; opposite terms indicate an opposite direction. A carry during addition is propagated towards the most significant bit, and this direction is sometimes called a propagation to the "left". A message is propagated towards the final hash output.

Some examples are given below to make this notation and terminology easier to understand intuitively.

- w_r denotes a data-word or expansion-word in any round.
- w_r^i denotes any bit i in a data-word or expansion-word in any round.
- w_4 is the 5th word of the input data and w_0 is the first word.
- $a_{r\geq 32}$ or $a_{r\geq 32}^i$ denote any internal state word from round 32 (using zero-indexing) upwards.
- w_{12}^{21} refers to the the 22nd bit of the 13th word.
- w^{26} or w_r^{26} both refer to bit 26 of any round.
- a_{r+1}^i refers to all internal state bits in rounds up to 79 (since $r + 1$ is not a valid round when $r = 80$).

2.6 Summary

This chapter has introduced necessary terminology and notations from mathematics and the field of cryptography, as well as thesis-specific notations and terminology. It has also provided an overview of the Merkle-Dåmgard structure and the SHA-1 hash function within which the SHA-1 compression function is embedded. This overview is provided only for background purposes since this research considers the compression function in isolation. More relevant to the remainder of this work is Algorithm 2.1 and the description of the f-functions given in Section 2.4. The next chapter examines related research, and will draw on the concepts and notations given in this chapter.

Part II

SHA-1 in specific

Chapter 3

Related work

Since its standardisation by NIST in 1995, SHA-1 has spawned a great deal of research activity, most of which has been focused on finding collisions. In this chapter, particular strands of related research are drawn together, starting with more theoretical and general questions and moving steadily towards the practically-oriented and specific preimage attacks on SHA-1.

3.1 In theory: one-way functions

The compression function is ideally a *one-way function*: a function that cannot be inverted due to some "hard" problem that prevents its inversion. Unfortunately for the cryptographic field, nobody has yet shown that a one-way function exists. Levin (2003) provides a readable overview of the situation and constructs a universal one-way function as a subject of study: if such a universal function can be shown to be one-way, then one-way functions do exist. Kojevnikov and Nikolenko (2008) also describe such universal functions, coming at the problem from a somewhat different angle.

There is no formal proof that the SHA-1 compression function is a one-way function, nor has it been constructed along the lines of a universal one-way function such that it can be shown to be one-way iff one-way functions exist. The preimage-resistance of the function is simply due to exceptionally good confusion and diffusion characteristics over the course of 80 rounds, as well as the fact that it has withstood preimage attacks for more than two decades.

3.2 A formalisation of cryptographic hash security

Rogaway and Shrimpton (2009) formalise the notions of cryptographic hash-function security. For the purposes of this work, the important formalizations that they provide are *aPre* and *aSec*. Their aPre formalisation corresponds to the notion of preimage resistance, and is given as

$$\mathbf{Adv}_H^{\mathrm{aPre}[m]}(A) = \max_{K \in \mathcal{K}} \left\{ \Pr \left[M \xleftarrow{\$} \{0,1\}^m ; Y \leftarrow H_K(M); M' \xleftarrow{\$} A(Y) : H_K(M') = Y \right] \right\} \tag{3.1}$$

It is worthwhile to expand this formalisation. Let the length of the message be m. Now, given a compression function $H(M,K) = Y$ where $M \in \mathbb{Z}_2^m$ is the input message, $K \in \mathbb{K}$ is a fixed chaining value, and $Y \in \mathbb{Z}_2^n$ is the output, let $A : \mathbb{Z}_2^n \mapsto S$ be an attacker-chosen algorithm such that $S \subset \mathbb{Z}_2^m$. Choose M from a uniform distribution, pre-specify K, and calculate $H(M,K) = Y$. Choose M' from the distribution of $A(Y)$. Then the preimage resistance of H with regard to the algorithm A is the maximum probability, over all $K \in \mathbb{K}$ and across all chosen M and M', that $H(M',K) = Y$.

Intuitively, this means that the security of the hash depends on the degree of confusion and diffusion (Shannon, 1949a) of H. If the amount of confusion and diffusion is sufficient for A to be unable to formulate any input which matches the desired output, then any function A has no advantage over a brute-force search of the space.

Rogaway and Shrimpton (2009) further point out, using a similar notation to the one already seen, that the second-preimage security of the hash function (which they call aSec) is dependent upon the difference between domain m and range n. This makes intuitive sense, since a smaller range means that a single output can map to multiple inputs. If the range was equal to or larger than the domain, then there is no guarantee that a second-preimage would even exist.

3.3 Desirable features

While Rogaway and Shrimpton (2009) do not relate the concepts of collision-resistance and preimage-resistance, Preneel (1993) does, defining a Collision-Resistant Hash Function (CRHF) to also be effectively one-way. The rationale for this, though not stated in these terms, appears to be that a collision could be obtained from cryptographic hash function which is not preimage-resistant by finding two preimages. This rationale is quite plausible if one assumes a greater domain than range and applies the pigeonhole principle.

Preneel calls an attack on a single-block compression function a "Direct Attack", stating that it "can be thwarted by imposing the requirement that [compression function] f is one-way" (Preneel, 1993, p. 35). While trivially true, this requirement has not been shown to be met by the SHA-1 compression function. The same work classifies attacks on hash functions into five types (Preneel, 1993, p. 40):

1. attacks independent of the algorithm,
2. attacks dependent on the chaining,
3. attacks dependent on an interaction with the signature scheme,
4. attacks dependent on the underlying block cipher,
5. high-level attacks.

Types (1), (3), (4), and (5) are inapplicable in the context of this work. Type (2) contains the sub-categories "Analytical weaknesses", "Meet-in-the-middle" and "Differential" attacks, and has been the focus of most preimage-focused research into the SHA-1 compression function. However, instead of targeting multi-block messages, differential characteristics of different words are used to "cancel out" changes with a certain probability, allowing meet-in-the-middle attacks to be more successful. A fuller description of these kinds of attacks is given in Section 3.6.

The recommendations of Preneel (1993) are mostly related to the overarching hash structure (see, for SHA-1, Section 2.3), but the recommendation to reuse every message bit as many times as possible does specifically apply to the compression function. Furthermore, it is recommended that a statistical evaluation be performed on the algorithm. To our knowledge, no such systematic statistical evaluation has been published for SHA-1, and this thesis contains some of the first results in that direction (see Section 5.2).

Chapter 8 of Preneel (1993) studies the importance of boolean function properties. The remaining research discussed in this chapter does not depend on such properties or their importance, and the topic is therefore mentioned here for completeness but will be discussed at greater length in Section 4.2.2.

3.4 A family of functions

SHA-1 is an "ARX" function, which means that it uses **A**ddition, **R**otation, and e**X**clusive-or operations (as well as $\wedge$, $\vee$, and $\neg$). The purpose of each of these is summarised by Khovratovich and Nikolić (2010):

> Addition provides diffusion and nonlinearity, while XOR does not. Although the diffusion is relatively slow, it is compensated by a low price of addition in both software and hardware, so primitives with relatively high number of additions (tens per byte) are still fast. The intraword rotation removes disbalance between left and right bits (introduced by the addition) and speeds up the diffusion.

A general method of analysing ARX functions was suggested in the same work. This method, called *rotational cryptanalysis*, focuses on performing an analysis of the function using a pair $(X, X \hookrightarrow r)$ throughout; the benefit is that probability analysis of the result of addition operations becomes easier. Further work in the same area (Velichkov, Mouha, De Canniere, and Preneel, 2011) extended this analysis, and Leurent (2012) published an analysis of differential path construction approaches using rotational analysis. Velichkov, Mouha, De Cannière, and Preneel (2012) suggested more improvements to the accuracy of addition probability estimation. Biryukov and Velichkov (2013) suggested an easier way to create good differential paths that were applicable to all ARX functions, but did not apply this method specifically to the case of SHA-1.

A differential path tracks the way in which message differences introduced in earlier rounds will be propagated to later rounds, with a certain probability. Such a path can then be used to identify or find a two messages which, when processed by the compression function, will result in the same output (and, therefore, a collision). To put it another way, differential path attacks aim to identify the characteristics of inputs that allow certain changes in the input bits to result in a greater probability of being "cancelled out" or "corrected" in later rounds. These characteristics can then be used to construct different inputs which would result in the same output at a particular round – in other words, a collision. Section 3.3 of Leurent (2010) gives a very readable and understandable overview of how differential paths operate, and is recommended for readers interested in the details; the original work that describes the concepts is somewhat less readable.

Although there may be future research which uses the concepts of rotational cryptanalysis to create a preimage attack on ARX functions such as SHA-1, there is no obvious link between the two. Rotational cryptanalysis works naturally from input towards output and does not provide much insight into inputs that could result in outputs. It is therefore mentioned here as a relatively new direction to be followed, but no practical preimage results are expected to arise from this research in the short-term.

3.5 Optimisations

In a sense, the preimage problem is an optimisation problem. If it were possible to obtain a hash output with very little computational effort, then the brute-force method of finding a preimage (or collision) becomes feasible. As an extreme example, if 4×10^{44} hashes could be computed every hour, then it would take just over an hour to find a SHA-1 preimage using the brute-force approach. Computational power does not yet approach that level. The Sunway TaihuLight is (as of June 2016) considered to be the fastest supercomputer in the world and can perform at a rate of approximately 93 PFLOPS; if SHA-1 could be calculated at the same rate of 93 quadrillion hashes per second, then only $\approx 3.35 \times 10^{20}$ hashes could be calculated per hour, and a preimage would take $\approx 4.37 \times 10^{27}$ hours to obtain — far longer than it will take for our sun to burn out!

There are, however, significant barriers that stand in the way of such an "optimized" calculation. Bosselaers, Govaerts, and Vandewalle (1996); Bosselaers (1997); Bosselaers, Govaerts, and Vandewalle (1997) published the first in-depth research that investigated the maximum amount of parallelism that could be extracted from SHA-1 and similar algorithms such as MD5. These concluded that SHA-1 exhibited a high degree of instruction-level parallelism and indicated a speedup factor of 1.76 for an optimised implementation on the original Pentium. Indeed, since 2013 it has been possible to use a built-in x86 instruction set extension for Intel processors (Gulley, Gopal, Yap, Feghali, Guilford, and Wolrich, 2013) to calculate a SHA-1 hash with hardware support; AMD's Ryzen CPUs, due in 2017, support the same extension (Gopalasubramanian, 2015). Nevertheless, it takes approximately 20 uses of the `sha1rnds4` instruction, usually sandwiched between `sha1msg2` and `sha1msg1` instructions, to obtain the hash value for a single block of data.

Dedicated hardware has also been used to good effect. Lien, Grembowski, and Gaj (2004) unrolled five rounds of SHA-1 on custom hardware (a Xilinx XCV1000), obtaining a hashing throughput of 1Gbit/s compared to a baseline hashing throughput of 544 Mbit/s obtained when no unrolling was done.

Lastly, alterations to the way in which the algorithm proceeds can have big effects: Steube (2004) demonstrated that unrolling the message-expansion phase of the SHA-1 compression function resulted in a speedup factor of 1.21.

Each of the above authors note, in their respective works, the importance of data dependencies in the SHA-1 algorithm. Each round is dependent on previous rounds as well as on a linear expansion of the original data. The amount of parallelism that is possible to obtain is therefore limited to a maximum of 5 rounds of the 80-round compression function, and each block must be calculated independently before calculation of the next block can proceed. The data dependencies are built into the algorithm itself and cannot be avoided in any obvious way; therefore, there is an inherent limit to how fast a SHA-1 compression function can execute. To return to the example at the beginning of this section, in the absence of optimizations that are currently unknown it is impossible for a supercomputer to calculate SHA-1 at a speed that is near its PFLOPS rating, no matter how optimized the compression function implementation is. This limits the impact of advances in hardware and makes "brute-force" an unsuccessful strategy for finding a preimage well into the future.

3.6 Differential path construction

Instead of trying to find "good" w_r^i values and then seeing whether they result in the appropriate $a_{76..80}$ values, De Canniere and Rechberger (2008) begin with unfixed a_r^i and w_r^i values and attempts to modify a_r^i in a way which satisfies w_r^i. The rationale for doing this is that changes made to a_r^i are easier to propagate and understand in terms of their overall impact on other w_r^i. They set up the following relations, expressed here in terms of the notation already described:

$$\begin{aligned}
E_r^i &= w_r^{i+31} \oplus w_{r-3}^i \oplus w_{r-8}^i \oplus w_{r-14}^i \oplus w_{r-16}^i = 0 \qquad \text{when } 16 \leq r < 80 \\
W_r^{i+5} &= C_r^i \oplus f_r^{i+5} \oplus a_{r-4}^{i+30} \oplus V_r^i \\
C_r^i &= a_{r+1}^i \oplus a_r^{i+5} \oplus k_r^{i+5} \\
V_r^i &= \begin{cases} 0 & \text{when } v_r^i \in \{0, 2, 4\} \\ 1 & \text{otherwise} \end{cases}
\end{aligned}$$

The E_r^i bits represent the amount of error in a particular solution. If a bit $a_r^{0 \leq j < 27}$ is flipped, then the E_r^j bit changes and changes in $E_r^{i>j}$ may occur; however, bits $E_r^{i<j}$ do *not* change. This occurs because carry-bits propagate towards the most significant bit of a word. Therefore, bits may be fixed column-by-column with some trial and effort (De Canniere and Rechberger (2008) cite a computational effort that is "linear in the number of rounds"), until only non-zero $E_r^{i \geq 25}$ bits remain. Removing these bits would require an exhaustive search of a $2^{7 \cdot 64} = 2^{448}$ solution space, although an optimized way of addressing the problem that requires searching through 2^{181} possibilities is also presented.

Aoki and Sasaki (2009) describe a "meet-in-the-middle" attack, a basic version of which has a time complexity of $2^{156.7}$ compression-function evaluations and a memory complexity of $11 \cdot 2^{40}$ words. The attack will result in a pseudo-preimage which does not necessarily obey the Merkle-Dåmgard structure. The fundamental idea behind such an attack is to find two different ways in which a particular hash value can be obtained. The initial step is to find two $w_{i \geq 16}$ values which are independent of different $w_{i<16}$; the chosen words are called *neutral* words. For SHA-1, an exhaustive list of these is presented as Table 3.1. Suitable tuples of choices therefore include (w_{30}, w_{43}) or (w_{22}, w_{42}), but not (w_{34}, w_{46}) or any tuple involving w_{72}.

Let (w_j, w_k) be the chosen tuple where $j < k$, and let (w_x, w_y) be the respective neutral words. Choose a value $j < t \leq k$, and let the words $a_{1..t-1}$ and $a_{t..80}$ be considered as different *chunks*. Select hash function output h to target and set $w_{i \notin \{x,y\}}$ to random values. Now compute the hash in both forward and backward directions using all possible values of w_x and w_y, stopping at a_t. If a common value of a_t is found, then the values of w_x and w_y used will result in a preimage.

Aoki and Sasaki (2009) also present two techniques for increasing the likelihood of success of the meet-in-the-middle attack. The first ("splice-and-cut") is inapplicable in the context of this work since input sizes ≥ 447 are not considered. The second ("partial-matching" and "partial-fixing") relies on being able to find the one of the two chosen neutral words within 4 rounds of each other. Unfortunately, as Table 3.1 makes clear, this requirement is not met for SHA-1 for anything but the earliest rounds. Although a 0-1-13-14-15 cycle is evident, the period of the cycle is > 4. Furthermore, two words in the cycle (w_{14} and w_{15}) are fixed by the requirements of the SHA-1 specification and cannot be modified in line with the described attack. A variation of the partial techniques based on bits, rather than words, is explored and attempted, but is superceded by later research.

Rechberger (2010) uses the idea of finding *differentials* to show that SHA-1 is more susceptible to second-preimage attacks than was believed. A differential is a 2-tuple of a/w

Table 3.1: Expansion-word independence

Word	Independent of...		Word	Independent of...
w_{16}	1,2,3,4,5,6,7,9,10,11,12,14,15		w_{42}	1,3,13
w_{17}	0,2,4,5,6,7,8,9,10,11,12,13,15		w_{43}	4,14
w_{18}	0,1,3,5,6,7,8,9,11,12,13,14		w_{44}	5,15
w_{19}	1,4,6,7,9,10,12,14,15		w_{45}	0,6
w_{20}	0,2,5,7,8,10,11,13,15		w_{46}	1,7
w_{21}	0,1,3,6,8,9,11,12,14		w_{50}	13
w_{22}	1,4,7,9,10,12,15		w_{51}	14
w_{23}	0,2,5,8,10,11,13		w_{52}	15
w_{24}	1,3,6,9,11,12,13,14		w_{53}	0
w_{25}	4,7,10,12,14,15		w_{54}	1
w_{26}	0,5,8,11,13,15		w_{62}	13
w_{27}	0,1,6,9,12,14		w_{63}	14
w_{28}	1,7,10,15		w_{64}	15
w_{29}	0,2,8,11		w_{65}	0
w_{30}	1,3,9,12,13		w_{66}	1,13
w_{31}	4,10,14		w_{67}	14
w_{32}	5,11,15		w_{68}	15
w_{33}	0,6,12		w_{69}	0
w_{34}	1,7		w_{70}	1
w_{36}	13		w_{74}	13
w_{37}	14		w_{75}	14
w_{38}	13,15		w_{76}	15
w_{39}	0,14		w_{77}	0
w_{40}	1,15		w_{78}	1
w_{41}	0,2			

bits where the value of one bit is expected to result in the other bit becoming known with probability p. A *characteristic* is a set of such probable differences. Good characteristics are useful for finding collisions; however, as Rechberger (2010) demonstrates, they are of limited utility when finding a preimage. Finding a second-preimage of a_{61} is estimated to take a work factor of $2^{159.42}$ even when constraints are relaxed somewhat, and finding a preimage of $a_{80..76}$ requires even more work.

The research of Knellwolf and Khovratovich (2012) represents the cutting-edge of SHA-1 preimage research at present. Their work ties the work of Aoki and Sasaki (2009) to a deep vein of existing research into differential cryptanalysis, reinterpreting it so that it can be understood in terms of the differential paradigm. This work will present a simplified summary of their approach, tailored more specifically to the constraints of SHA-1 and the context of single-chunk hashing. Knellwolf and Khovratovich (2012) represent the compression function $f : \mathbb{Z}_2^{0 \leq n \leq 447} \to \mathbb{Z}_2^{160}$ as the composition of functions $f_1 : \mathbb{Z}_2^{0.. \leq n \leq 447} \to \mathbb{Z}_2^{160}$ and $f_2 : \mathbb{Z}_2^{160} \to \mathbb{Z}_2^{160}$, i.e. $f = f_2 \circ f_1$.

Let the number of input bits be $\mathcal{K}$. The entire 512-bit message is M and the compression function output is C (or, in other words, $f : M \to C$). Choose sets D_1 and D_2 with size $d < \mathcal{K}$, such that $D_1 \cap D_2 = \emptyset$; for example, given $\mathcal{K} = 8$, a set D_1 of size 3 could consist of $\{w_0^0, w_0^2, w_0^3\}$, leaving 5 possible choices to choose for inclusion in D_2.

As in the work of Aoki and Sasaki (2009), it is necessary to decide on a "meet-in-the-middle" point at which f_1 produces its output and from which f_2 takes its input. Ideally, this point would be near the middle of f to provide sufficient space for the hashing of f_2 to result in C. Define $\Delta_1, \Delta_2 \in \{0,1\}^{160}$ as $\Delta_1(\delta_1) = f_1(M) \oplus f_1(M \oplus \delta_1)$ and $\Delta_2(\delta_2) = f_2^{-1}(M) \oplus f_2^{-1}(M \oplus \delta_2)$. In other words, Δ_1 is the difference in f_1's output, given a δ_1 difference in the input. Similarly, Δ_2 is the difference in f_1's output, given a δ_2 difference in the input. Since we are searching for a preimage, is only necessary to be interested in cases where the output of f_2 is C.

Compute all values $L_1[\delta_2] = f_1(M \oplus \delta_2) \oplus \Delta_2$. Now compute $L_2[\delta_1] = f_2^{-1}(M \oplus \delta_1) \oplus \Delta_1$. If $L_2[\delta_1] \oplus L_1[\delta_2] = 0$, then a preimage $(M \oplus \delta_1 \oplus \delta_2)$ has been found. To understand why, consider the expansion of these terms.

$$\begin{aligned}
L_2[\delta_1] \oplus L_1[\delta_2] &= 0 \\
f_2^{-1}(M \oplus \delta_1) \oplus \Delta_1 \oplus f_1(M \oplus \delta_2) \oplus \Delta_2 &= 0 \\
f_2^{-1}(M \oplus \delta_1) \oplus f_1(M) \oplus f_1(M \oplus \delta_1) \oplus f_1(M \oplus \delta_2) \oplus f_2^{-1}(M) \oplus f_2^{-1}(M \oplus \delta_2) &= 0
\end{aligned}$$

Since $f(x) = f_2(f_1(x)) = C$ (or, equivalently, $f = f_2 \circ f_1 = C$), it follows that $f_1(x) = f_2^{-1}(C)$ for all preimages. Identical terms "cancel" each other out, causing the LHS to be 0.

At least $160 \cdot 2^d$ bits are required to store L_1 data naïvely (Knellwolf and Khovratovich state a figure of $(n + d) \cdot 2^d$, but d can be inferred from implied data such as array/record index). The benefit of using a meet-in-the-middle approach is that 2^n messages are tested at a cost of 2^{n-2d}, making it easier to find a (second-)preimage. However, the time-space trade-off that results from using larger d does not scale well. Knellwolf and Khovratovich (2012) provide an estimated time complexity of $2^{158.44}$ to find a correctly-structured 52-round SHA-1 preimage.

The amount of space required can be reduced by using *truncated* differentials — in short, reducing the number of bits that are stored and compared for L_i values. This leads to an unknown number of false positives, and a corresponding amount of time that must be spent retesting candidate values to see if they are correct.

Espitau *et al.* (2015) build on work of Knellwolf and Khovratovich (2012) by suggesting the use of higher-order derivatives to find better D_i spaces. The original work on higher-order derivatives was done by Lai (1994), and it is worth noting that their definition of "derivative" differs from the standard one:

> **Definition** Let $(S, +)$ and $(T, +)$ be Abelian [i.e. commutative] groups. For a function $f : S \to T$, the *derivatives of f at point $a \in S$* is defined as
>
> $$\Delta_a f(x) = f(x + a) - f(x)$$

The net result of using this definition is the ability to effectively split each D_i space into two, resulting in $D_{i,1}$ and $D_{i,2}$. This results in four L arrays ($L_{1,1}$, $L_{1,2}$, $L_{2,1}$, and $L_{2,2}$) in addition to the two existing L_i arrays, making each check for a preimage require six lookups. The space complexity increases as a consequence, and it is only by ignoring any complexity that this adds that the time complexity remains the same. As a result, the authors have estimated a time complexity of $2^{159.4}$ to find a preimage for 56-round SHA-1.

The best time complexity estimates for finding a SHA-1 preimage have come about as a result of bringing together the strands of differential cryptanalysis and meet-in-the-middle attacks. Knellwolf and Khovratovich (2012) and Espitau *et al.* (2015) represent the best research in this area thus far. A large problem is finding "good" characteristics for SHA-1, and also overcoming the enormous space/time requirements that exist at present. The literature suggests no good way around either of these obstacles and, unfortunately, the best efforts in this direction have thus far yielded attacks with a work factor that is far beyond that of the brute-force strategy.

3.7 Other research

As has been mentioned, a great deal of research has focused on collision attacks and is therefore not directly applicable to the focus of this thesis. Nevertheless, for completeness, it is worth briefly mentioning some of the more important threads of this research so that interested readers can pursue what they find to be interesting. The research in this section is presented in mostly-chronological order and, where appropriate, the evolution of particular attacks has been explicitly noted.

Joux and Peyrin (2007) suggested a “boomerang” attack on SHA-1 which provides a better way of finding differential paths, based on previous research by Wagner (1999) into block ciphers, and reduced the work required to find a collision by a factor of 2^5.

In a separate vein of research, Wang, Yin, and Yu (2005b) published a differential path attack, based on previous work on the MD5 and SHA-0 algorithms (Wang and Yu, 2005; Wang, Yu, and Yin, 2005c), which was theoretically able to obtain a collision after 2^{69} hash evaluations. De Canniere and Rechberger (2006) published a further analysis of differential paths for SHA-1 that generalised Wang *et al.*'s results, making it easier to find suitable inputs which might lead to a collision. Cochran (2007), using improvements on Wang *et al.*'s method, verified an attack that required approximately 2^{63} operations.

Manuel (2011) analysed the optimality of differential paths and their manner of generation, making it easier to select the most suitable differential path to use for an attack. Eichlseder, Mendel, Nad, Rijmen, and Schlaeffer (2013) published an optimisation to assist in the construction of valid differential paths. Stevens (2013) demonstrated a near-collision at a cost of $2^{57.5}$ hash evaluations using additional optimisations to previous differential path techniques.

Grechnikov approached the problem from a general-purpose graphics processing unit (GPGPU) point-of-view and used GPU clusters and performance optimisations to find collisions for 73-round (Grechnikov, 2010) and, subsequently, 75-round (Adinetz and Grechnikov, 2012) versions of SHA-1.

More recently, Stevens, Bursztein, Karpman, Albertini, and Markov (2017) built upon previous research into differential paths (Stevens, 2013; Stevens, Karpman, and Peyrin, 2016) to obtain the very first collision for the full 80 rounds of SHA-1. This feat was achieved using a time complexity of approximately $2^{63.1}$ hash function evaluations.

For completeness, it is worth mentioning the work of Kelsey and Schneier (2005) in this section. Although the claim of finding second preimages on a n-bit hash function for less than 2^n work is accurate, it is only accurate when very large inputs are used. In the case of the SHA-1 hash function, for 2^k chunks (or message-blocks) of input, the work required is $k \times 2^{81} + 2^{160-k+1}$.

Example 3.1. *Kelsey and Schneier (2005).* Assuming that one wishes to find a preimage for a hashed 25-byte password, for example, one would use $k = 0$ since 25 bytes fits within a single 512-bit chunk. The work required would therefore be $0 \times 2^{81} + 2^{160-0+1} = 2^{161}$, which is slightly more work than the brute-force work factor of 2^{160}.

Their attack is compression-function-agnostic: it can be applied to any hash function that uses a Merkle-Dåmgard structure, including SHA-1. For the purposes of this research, however, this means that Kelsey and Schneier's research is not of any use since this research focuses specifically on the SHA-1 compression function.

3.7.1 Logical cryptanalysis

Massacci and Marraro (2000) coined the term *logical cryptanalysis* to refer to the idea of modeling and encoding a cryptographic problem in a way that makes them feasible to solve via heuristic methods. These heuristic methods include artificial intelligence (AI) techniques and boolean satisfiability (SAT) solving.

More recently, Legendre, Dequen, and Krajecki (2012) modeled and encoded both MD5 and SHA-1, focusing more on the former than the latter, in a form suitable for SAT-solving. This approach was able to invert "about 1 round 3 steps" of SHA-1 which, as the authors point out, is somewhat worse than the results of De Canniere and Rechberger (2008) and Rechberger (2010). More in-depth work by the same authors analyses the representation and encoding of the same cryptographic hash functions in much more detail (Legendre, Dequen, and Krajecki, 2014).

The work of Li and Ye (2014) is representative of the direction that logical cryptanalysis has taken. It presents a different encoding of MD5 to the encoding chosen by Legendre *et al.* (2012) and this, in turn, makes it more suitable as a challenge for SAT-solvers.

Logical cryptanalysis is an avenue that is explored in more depth in Chapters 7 and 11; these use SAT-solving and a constraint solver respectively. This research, and other related research, will be covered in more depth there, and is mentioned for completeness here.

3.7.2 Summary

This chapter has presented an overview of preimage and related research. There are two items that are particularly of note as the chapter ends: firstly, that SHA-1 preimage research (and preimage research in most cases) has not yet resulted in any significant success; and, secondly, that there are some alternative formulations of SHA-1 that have been suggested in the literature. Notable alternative formulations include:

1. A spliffling phase that uses only a-values, by De Canniere and Rechberger (2008) and others:

$$a_{i+1} = (a_i \hookleftarrow 5) + w_i + f(a_{i-1}, a_{i-2} \hookrightarrow 2, a_{i-3} \hookrightarrow 2) + (a_{i-4} \hookrightarrow 2) + k_i$$

2. A partially-unrolled spliffling phase, by Lien *et al.* (2004):

$$\begin{array}{lcl}
a_{i+1} & = & a_i \hookleftarrow 5 + f_i(b_i, c_i, d_i) + e_i + \sum k_i w_i \\
a_{i+2} & = & a_{i+1} \hookleftarrow 5 + [f_{i+1}(a_i, b_i \hookleftarrow 30, c_{i+1}) + d_i + \sum k_{i+1} w_{i+1}] \\
a_{i+3} & = & a_{i+2} \hookleftarrow 5 + [f_{i+2}(a_{i+1}, a_i \hookleftarrow 30, b_i \hookleftarrow 30) + [c_i + \sum k_{i+2} w_{i+2}]] \\
a_{i+4} & = & a_{i+3} \hookleftarrow 5 + [f_{i+3}(a_{i+2}, a_{i+1} \hookleftarrow 30, a_i \hookleftarrow 30) + [b_i \hookleftarrow 30 + \sum k_{i+3} w_{i+3}]] \\
a_{i+5} & = & a_{i+4} \hookleftarrow 5 + [f_{i+4}(a_{i+3}, a_{i+2} \hookleftarrow 30, a_{i+1} \hookleftarrow 30) + [a_i \hookleftarrow 30 + \sum k_{i+4} w_{i+4}
\end{array}$$

Formulation (1) will be discussed in Chapter 4. Formulation (2) is worth noting because it explicitly draws out the data dependencies of a single round; these dependencies will be discussed in their own section in Chapter 4.2.4 as well.

Chapter 4

An in-depth exploration

In this chapter an in-depth exploration of the SHA-1 hash is conducted. The chapter begins with an analysis of the message-expansion phase, followed by a similar analysis of the spliffling phase. A summary of the analysis thus far is presented and this is followed by a more unusual non-binary formulation. The chapter ends with a short recap of how to go from a "final" single-block hash function output to the compression function output.

4.1 Message-expansion analysis

Recall from Section 2.4 that the message-expansion equation is

$$w_{r \geq 16} = (w_{r-3} \oplus w_{r-8} \oplus w_{r-14} \oplus w_{r-16}) \hookleftarrow 1$$

As the expansion sequence progresses, more of the data-words tend to be used in each round. For the initial expansion-word w_{16}, only four data-words are used: $w_{16} = (w_{13} \oplus w_8 \oplus w_2 \oplus w_0) \hookleftarrow 1$. However, by the time that w_{34} (for example[1]) is considered, every data-word is used at least once.

[1]Note that w_{34} will appear in many places, but that should not be taken as meaning that it is significant in and of itself. It is of sufficient complexity to demonstrate some points, while still being simple enough for the necessary calculations to not overwhelm the reader or dominate the text. It also provides a common point of reference for comparing different calculation methods and representations.

Example 4.1. *Increasing usage of data-words.* To see that every data-word will be used at least once by w_{34}, the expansion-word can be unpacked in its entirety through substitution:

$$\begin{aligned}
w_{34} &= (w_{31} \oplus w_{26} \oplus w_{20} \oplus w_{18}) \hookleftarrow 1 \\
&= (((w_{28} \oplus w_{23} \oplus w_{17} \oplus w_{15}) \hookleftarrow 1) \oplus ((w_{23} \oplus w_{18} \oplus w_{12} \oplus w_{10}) \hookleftarrow \\
&\quad 1) \oplus ((w_{17} \oplus w_{12} \oplus w_{6} \oplus w_{4}) \hookleftarrow 1) \oplus ((w_{15} \oplus w_{10} \oplus w_{4} \oplus w_{2}) \hookleftarrow 1)) \hookleftarrow 1 \\
&= (((((w_{25} \oplus w_{20} \oplus w_{14} \oplus w_{12}) \hookleftarrow 1) \oplus ((w_{20} \oplus w_{15} \oplus w_{9} \oplus w_{7}) \hookleftarrow 1) \oplus \\
&\quad ((w_{14} \oplus w_{9} \oplus w_{3} \oplus w_{1}) \hookleftarrow 1) \oplus w_{15}) \hookleftarrow 1) \oplus ((((w_{20} \oplus w_{15} \oplus w_{9} \oplus w_{7}) \hookleftarrow \\
&\quad 1) \oplus ((w_{15} \oplus w_{10} \oplus w_{4} \oplus w_{2}) \hookleftarrow 1) \oplus w_{12} \oplus w_{10}) \hookleftarrow 1) \oplus ((((w_{14} \oplus w_{9} \oplus w_{3} \oplus \\
&\quad w_{1}) \hookleftarrow 1) \oplus w_{12} \oplus w_{6} \oplus w_{4}) \hookleftarrow 1) \oplus ((w_{15} \oplus w_{10} \oplus w_{4} \oplus w_{2}) \hookleftarrow 1)) \hookleftarrow 1 \\
&\vdots \\
&= ((((((((((((w_{13} \oplus w_{8} \oplus w_{2} \oplus w_{0}) \hookleftarrow 1) \oplus w_{11} \oplus w_{5} \oplus w_{3}) \hookleftarrow 1) \oplus \\
&\quad w_{14} \oplus w_{8} \oplus w_{6}) \hookleftarrow 1) \oplus ((w_{14} \oplus w_{9} \oplus w_{3} \oplus w_{1}) \hookleftarrow 1) \oplus w_{11} \oplus w_{9}) \hookleftarrow \\
&\quad 1) \oplus ((((w_{14} \oplus w_{9} \oplus w_{3} \oplus w_{1}) \hookleftarrow 1) \oplus w_{12} \oplus w_{6} \oplus w_{4}) \hookleftarrow 1) \oplus w_{14} \oplus w_{12}) \hookleftarrow \\
&\quad 1) \oplus ((((((w_{14} \oplus w_{9} \oplus w_{3} \oplus w_{1}) \hookleftarrow 1) \oplus w_{12} \oplus w_{6} \oplus w_{4}) \hookleftarrow 1) \oplus w_{15} \oplus w_{9} \oplus \\
&\quad w_{7}) \hookleftarrow 1) \oplus ((w_{14} \oplus w_{9} \oplus w_{3} \oplus w_{1}) \hookleftarrow 1) \oplus w_{15}) \hookleftarrow 1) \oplus ((((((((w_{14} \oplus \\
&\quad w_{9} \oplus w_{3} \oplus w_{1}) \hookleftarrow 1) \oplus w_{12} \oplus w_{6} \oplus w_{4}) \hookleftarrow 1) \oplus w_{15} \oplus w_{9} \oplus w_{7}) \hookleftarrow \\
&\quad 1) \oplus ((w_{15} \oplus w_{10} \oplus w_{4} \oplus w_{2}) \hookleftarrow 1) \oplus w_{12} \oplus w_{10}) \hookleftarrow 1) \oplus ((((w_{14} \oplus w_{9} \oplus w_{3} \oplus \\
&\quad w_{1}) \hookleftarrow 1) \oplus w_{12} \oplus w_{6} \oplus w_{4}) \hookleftarrow 1) \oplus ((w_{15} \oplus w_{10} \oplus w_{4} \oplus w_{2}) \hookleftarrow 1)) \hookleftarrow 1
\end{aligned}$$

Another formulation of the expansion process takes into account the fact that an expansion-word, once calculated, may be used several times but cannot change its value. For instance, w_{17} is referenced directly by w_{20}, w_{25}, and w_{31}, and indirectly by any expansion-words that make use of these (such as w_{23}, which references w_{20}). Expansion-words can therefore be represented as partial results that lead to a final result.

At the level of bits, the bit which is affected by flipping another bit depends entirely on the number of rotations used to calculate the expansion-word. The number of rotations is the same as the depth to which the expansion is nested, since each level of nesting results in a left-rotation by a single bit-position. Therefore w_r^i will have the equation

$$w_r^i = \begin{cases} w_r^i & \text{when } r < 16 \\ w_{r-3}^{i+1} \oplus w_{r-8}^{i+1} \oplus w_{r-14}^{i+1} \oplus w_{r-16}^{i+1} & \text{otherwise} \end{cases} \tag{4.1}$$

Example 4.2. *Partial results leading to a final result.*

$$\begin{aligned}
w_{16} &= (w_{13} \oplus w_8 \oplus w_2 \oplus w_0) \hookleftarrow 1\\
w_{17} &= (w_{14} \oplus w_9 \oplus w_3 \oplus w_1) \hookleftarrow 1\\
w_{18} &= (w_{15} \oplus w_{10} \oplus w_4 \oplus w_2) \hookleftarrow 1\\
w_{19} &= (w_{16} \oplus w_{11} \oplus w_5 \oplus w_3) \hookleftarrow 1\\
w_{20} &= (w_{17} \oplus w_{12} \oplus w_6 \oplus w_4) \hookleftarrow 1\\
w_{22} &= (w_{19} \oplus w_{14} \oplus w_8 \oplus w_6) \hookleftarrow 1\\
w_{23} &= (w_{20} \oplus w_{15} \oplus w_9 \oplus w_7) \hookleftarrow 1\\
w_{25} &= (w_{22} \oplus w_{17} \oplus w_{11} \oplus w_9) \hookleftarrow 1\\
w_{26} &= (w_{23} \oplus w_{18} \oplus w_{12} \oplus w_{10}) \hookleftarrow 1\\
w_{28} &= (w_{25} \oplus w_{20} \oplus w_{14} \oplus w_{12}) \hookleftarrow 1\\
w_{31} &= (w_{28} \oplus w_{23} \oplus w_{17} \oplus w_{15}) \hookleftarrow 1\\
w_{34} &= (w_{31} \oplus w_{26} \oplus w_{20} \oplus w_{18}) \hookleftarrow 1
\end{aligned}$$

Using equation 4.1 it is trivial to obtain an equation which is dependent upon data-words only, for any bit of any word, by expanding each term fully.

Example 4.3. *Expanding a message-expansion equation.*

$$\begin{aligned}
w_{26}^{30} &= w_{23}^{31} \oplus w_{18}^{31} \oplus w_{12}^{31} \oplus w_{10}^{31}\\
&= w_{20}^{0} \oplus w_{15}^{0} \oplus w_{9}^{0} \oplus w_{7}^{0} \oplus w_{15}^{0} \oplus w_{10}^{0} \oplus w_{4}^{0} \oplus w_{2}^{0} \oplus w_{12}^{31} \oplus w_{10}^{31}\\
&= w_{17}^{1} \oplus w_{12}^{1} \oplus w_{6}^{1} \oplus w_{4}^{1} \oplus w_{15}^{0} \oplus w_{9}^{0} \oplus w_{7}^{0} \oplus w_{15}^{0} \oplus w_{10}^{0} \oplus w_{4}^{0} \oplus w_{2}^{0} \oplus w_{12}^{31} \oplus w_{10}^{31}\\
&= w_{14}^{2} \oplus w_{9}^{2} \oplus w_{3}^{2} \oplus w_{1}^{2} \oplus w_{12}^{1} \oplus w_{6}^{1} \oplus w_{4}^{1} \oplus w_{15}^{0} \oplus w_{9}^{0} \oplus w_{7}^{0} \oplus w_{15}^{0} \oplus w_{10}^{0} \oplus w_{4}^{0} \oplus\\
&\quad w_{2}^{0} \oplus w_{12}^{31} \oplus w_{10}^{31}
\end{aligned}$$

Later rounds tend to use more data-words than earlier rounds, as shown by Example 4.1, and they also tend to use more data-bits. The identities $x \oplus x = 0$ and $x \oplus 0 = x$ can be used to reduce the number of terms: any symbol that appears an even number of times can be removed from the equation entirely, and any symbol that appears an odd number of times can be replaced by a single instance of itself. For simple equations such as w_{34}, the usefulness of this optimisation is limited. After application of the optimisation, w_{34}^{0}'s terms are reduced from 67 to 23. For later expansion-words with many dependencies, the optimisation is more significant:

Example 4.4. *An analysis of w_{79}^0.* The expansion of w_{79}^0 results in a set of 162,007 terms, viz. truncated:

$$\begin{aligned}
w_{79}^0 \quad = \quad & w_{15}^4 \oplus w_1^5 \oplus w_3^5 \oplus w_9^5 \oplus w_{14}^5 \oplus w_7^5 \oplus w_9^5 \oplus w_{15}^5 \oplus w_4^6 \oplus w_6^6 \oplus w_{12}^6 \oplus w_1^7 \oplus w_3^7 \oplus w_9^7 \oplus w_{14}^7 \oplus \\
& w_{12}^5 \oplus w_{14}^5 \oplus w_4^6 \oplus w_6^6 \oplus w_{12}^6 \oplus w_1^7 \oplus w_3^7 \oplus w_9^7 \oplus w_{14}^7 \oplus w_9^6 \oplus w_{11}^6 \oplus w_1^7 \oplus w_3^7 \oplus w_9^7 \oplus \\
& w_{14}^7 \oplus w_6^7 \oplus w_8^7 \oplus w_{14}^7 \oplus w_3^8 \oplus w_5^8 \oplus w_{11}^8 \oplus w_0^9 \oplus w_2^9 \oplus w_8^9 \oplus w_{13}^9 \oplus w_1^5 \oplus w_3^5 \oplus w_9^5 \oplus w_{14}^5 \oplus \\
& w_3^5 \oplus w_5^5 \oplus w_{11}^5 \oplus w_0^6 \oplus w_2^6 \oplus w_8^6 \oplus w_{13}^6 \oplus w_9^5 \oplus w_{11}^5 \oplus w_1^6 \oplus w_3^6 \oplus w_9^6 \oplus w_{14}^6 \oplus w_6^6 \oplus w_8^6 \oplus \\
& w_{14}^6 \oplus w_3^7 \oplus w_5^7 \oplus w_{11}^7 \oplus w_0^8 \oplus w_2^8 \oplus w_8^8 \oplus w_{13}^8 \oplus w_{14}^5 \oplus w_0^6 \oplus w_2^6 \oplus w_8^6 \oplus w_{13}^6 \oplus w_6^6 \oplus \\
& w_8^6 \oplus w_{14}^6 \oplus w_3^7 \oplus w_5^7 \oplus w_{11}^7 \oplus w_0^8 \oplus w_2^8 \oplus w_8^8 \oplus w_{13}^8 \oplus w_{11}^6 \oplus w_{13}^6 \oplus w_3^7 \oplus w_5^7 \oplus w_{11}^7 \oplus \\
& w_0^8 \oplus w_2^8 \oplus w_8^8 \oplus w_{13}^8 \oplus w_8^7 \oplus w_{10}^7 \oplus w_0^8 \oplus w_2^8 \oplus w_8^8 \oplus w_{13}^8 \oplus w_5^8 \oplus w_7^8 \oplus w_{13}^8 \oplus w_2^9 \oplus w_4^9 \oplus \\
& w_{10}^9 \oplus w_{15}^9 \oplus w_7^5 \oplus w_9^5 \oplus w_{15}^5 \oplus w_4^6 \oplus w_6^6 \oplus w_{12}^6 \oplus w_1^7 \oplus w_3^7 \oplus w_9^7 \oplus w_{14}^7 \oplus w_9^5 \oplus w_{11}^5 \oplus \\
& w_1^6 \oplus w_3^6 \oplus w_9^6 \oplus w_{14}^6 \oplus w_6^6 \oplus w_8^6 \oplus w_{14}^6 \oplus w_3^7 \oplus w_5^7 \oplus w_{11}^7 \oplus w_0^8 \oplus w_2^8 \oplus w_8^8 \oplus w_{13}^8 \oplus w_{15}^5 \oplus \\
& w_1^6 \oplus w_3^6 \oplus w_9^6 \oplus w_{14}^6 \oplus w_7^6 \oplus w_9^6 \oplus w_{15}^6 \oplus w_4^7 \oplus w_6^7 \oplus w_{12}^7 \oplus w_1^8 \oplus w_3^8 \oplus w_9^8 \oplus w_{14}^8 \oplus w_{12}^6 \oplus \\
& w_{14}^6 \oplus w_4^7 \oplus w_6^7 \oplus w_{12}^7 \oplus w_1^8 \oplus w_3^8 \oplus w_9^8 \oplus w_{14}^8 \oplus w_9^7 \oplus w_{11}^7 \oplus w_1^8 \oplus w_3^8 \oplus w_9^8 \oplus w_{14}^8 \oplus \\
& w_6^8 \oplus w_8^8 \oplus w_{14}^8 \oplus w_3^9 \oplus w_5^9 \oplus w_{11}^9 \oplus w_0^{10} \oplus w_2^{10} \oplus w_8^{10} \oplus w_{13}^{10} \oplus w_4^6 \oplus w_6^6 \oplus w_{12}^6 \oplus w_1^7 \oplus \\
& w_3^7 \oplus w_9^7 \oplus w_{14}^7 \oplus w_6^6 \oplus w_8^6 \oplus w_{14}^6 \oplus w_3^7 \oplus w_5^7 \oplus w_{11}^7 \oplus w_0^8 \oplus w_2^8 \oplus w_8^8 \oplus w_{13}^8 \oplus w_{12}^6 \oplus \ldots
\end{aligned}$$

These can be reduced from 162,007 terms to a mere 75.

$$\begin{aligned}
w_{79}^0 \quad = \quad & w_0^8 \oplus w_0^{22} \oplus w_1^7 \oplus w_1^8 \oplus w_1^{15} \oplus w_1^{18} \oplus w_1^{20} \oplus w_2^8 \oplus w_2^{22} \oplus w_3^{12} \oplus w_3^{15} \oplus w_3^{16} \oplus w_3^{18} \oplus \\
& w_3^{20} \oplus w_3^{21} \oplus w_4^6 \oplus w_4^{12} \oplus w_4^{14} \oplus w_5^7 \oplus w_5^{12} \oplus w_5^{21} \oplus w_6^6 \oplus w_6^{12} \oplus w_6^{14} \oplus w_6^{20} \oplus w_7^5 \oplus \\
& w_7^8 \oplus w_7^{11} \oplus w_7^{12} \oplus w_7^{13} \oplus w_7^{18} \oplus w_8^8 \oplus w_8^{12} \oplus w_8^{16} \oplus w_8^{20} \oplus w_8^{22} \oplus w_9^5 \oplus w_9^7 \oplus w_9^8 \oplus \\
& w_9^{11} \oplus w_9^{13} \oplus w_9^{15} \oplus w_9^{16} \oplus w_9^{19} \oplus w_9^{20} \oplus w_{10}^{12} \oplus w_{10}^{16} \oplus w_{11}^7 \oplus w_{11}^8 \oplus w_{11}^{11} \oplus w_{11}^{14} \oplus \\
& w_{11}^{15} \oplus w_{11}^{16} \oplus w_{11}^{19} \oplus w_{11}^{21} \oplus w_{12}^6 \oplus w_{12}^8 \oplus w_{12}^{12} \oplus w_{12}^{14} \oplus w_{12}^{18} \oplus w_{13}^{11} \oplus w_{13}^{15} \oplus w_{13}^{22} \oplus \\
& w_{14}^7 \oplus w_{14}^{15} \oplus w_{15}^4 \oplus w_{15}^5 \oplus w_{15}^6 \oplus w_{15}^7 \oplus w_{15}^8 \oplus w_{15}^{11} \oplus w_{15}^{13} \oplus w_{15}^{14} \oplus w_{15}^{17} \oplus w_{15}^{18}
\end{aligned}$$

4.1.1 The impact of bits

The equation for w_{16}^0 (i.e., $w_{16}^0 = w_{13}^1 \oplus w_8^1 \oplus w_2^1 \oplus w_0^1$) is a simple one in the sense that each position is only mentioned once. Solving for acceptable values is a trivial task: any four bits which, when XORed together, result in the desired bit are acceptable. Later expansion-bits, such as w_{22}, have more interesting equations.

$$w_{22}^{0} = w_{0}^{3} \oplus w_{2}^{3} \oplus w_{3}^{2} \oplus w_{5}^{2} \oplus w_{6}^{1} \oplus \mathbf{w_{8}^{1}} \oplus \mathbf{w_{8}^{3}} \oplus w_{11}^{2} \oplus w_{13}^{3} \oplus w_{14}^{1} \tag{4.2}$$

Notice that w_8 appears twice in this equation, with offsets 1 and 3. If we choose

01000000000000000000000000000000b

as our w_8 and leave all other words as zero, then

$$\begin{aligned}
w_{22}^{0} &= 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus \mathbf{1} \oplus 0 \oplus 0 \oplus 0 \oplus 0 &= 1 \\
w_{22}^{1} &= 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 &= 0 \\
\vdots\ \ & w_{22}^{2..28} \textit{ elided for brevity} \\
w_{22}^{29} &= 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 &= 0 \\
w_{22}^{30} &= 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus \mathbf{1} \oplus 0 \oplus 0 \oplus 0 &= 1 \\
w_{22}^{31} &= 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 &= 0
\end{aligned}$$

The final value of w_{22} is

100000000000000000000000000000**1**0b

In other words, a 1-bit at offset 1 of w_8 has affected *two* output bits. This is because the set bit participates in the final value twice: see the boldfaced entries above. This happens because a unique reference to w_8 occurs *twice* in Equation 4.2. If a unique reference were to occur n times, bits set from that word would participate n times in the expansion-word.

One consequence of this is that bits within the same word expansion-word can affect each other: if w_8 had the value

01010000000000000000000000000000b

the final value of w_{22} would be

00100000000000000000000000000010b

Notice that index 0 of w_{22} is no longer 1, despite index 1 of w_8 being set. This is because the two bits have canceled each other out.

The two affected output bits are separated by a single bit-position — just as the data-word indices, 1 and 3, are separated by a single bit-position. This makes it possible to describe the combined effect of the data-word terms on the expansion-word more concisely by expressing them as *bit-patterns*. Bit-patterns for each expansion-word, for each position, are shown in Table 4.1. Upon examination of this table, it is interesting to note that in some of the later expansions (such as w_{78}) it is evident that certain data-words (such as w_1) have no effect upon the expansion-word whatsoever. For other expansions, such as w_{79}, the table shows some words (such as w_9) having a marked effect upon the expansion-word, whereas other words (w_9 or w_2, for example) have little impact.

Bit-patterns represent only the effect of a particular bit on the final expansion-word; information about positions and indices of the equation terms is lost. When combined with this information, bit-patterns can be used to determine the value of a particular expansion-word via lookup tables instead of partial calculations. This means that the calculation of later expansion-words, such as w_{79}, can be accomplished without needing to calculate intervening expansion-words. The size of the lookup table is not exorbitant: $16 \times 64 = 1024$ entries are needed.

If we assume that such a lookup table exists and is indexed by data-word position *dp* and expansion-word position *ep* such that $lookup(dp, ep)$ returns the appropriate bit-pattern for index 0, then the equation for w_r^i would be

$$w_r^i = \bigoplus_{dp=0}^{15} (lookup(dp, r) \hookrightarrow i) \tag{4.3}$$

This method of calculating an expansion word will be called the *bitpattern* method of calculation in the remainder of this thesis.

Algorithm 4.1 expresses Equation 4.3 in a more concrete form, unrolling the hypothetical *lookup* function. The input to this is a table (i.e. two-dimensional array) of *bitpatterns*; the desired round r; the desired index i; and the 16 data-words dw. The *bitpatterns* input is indexed first by the desired round, and next by the data-word rounds. Each bitpattern encodes the pattern (bp) and the offset from the start of the word (*shift*) as a 2-tuple of words.

The majority of the time in Algorithm 4.1 is spent checking whether particular bits in words are set and, consequently, whether the relevant bitpattern should be XORed. An alternative implementation could precalculate and store the appropriate w_0 calculations

Algorithm 4.1 Bitpattern calculation method

```
Require:
    A 80 × 16 2-dimensional (unsigned integer, unsigned integer) array $bitpatterns$
    $0 \leq r \leq 79$
    $0 \leq i \leq 31$
    A 16-element array unsigned integer data-words $dw$
Ensure:
    A single bit representing $w_r^i \in 0, 1$
 1: function wValue($bitpatterns, r, i, dw$)
 2:     $final \leftarrow 0$
 3:     $pattern \leftarrow bitpatterns_r$
 4:     for $idx \leftarrow 0..31$ do
 5:         for $dr \leftarrow 0..15$ do
 6:             $shift, bp \leftarrow pattern_{dr}$
 7:             if $(0x80000000 \hookrightarrow (idx + shift)) \wedge dw_{dr}$ then
 8:                 $final \leftarrow (bp \hookrightarrow idx) \oplus final$
 9:     return $final_i$
10: end function
```

Algorithm 4.2 Obtaining terms for w_r^i

```
Require:
    A 64-element array of precalculated ($0 \leq r \leq 15$, $0 \leq i \leq 31$) tuples $baseEquations$
    $0 \leq r \leq 79$
    $0 \leq i \leq 31$
Ensure:
    A sequence of ($0 \leq r \leq 15$, $0 \leq i \leq 31$) tuples indicating relevant data-word bits
 1: function wTerms($baseEquations, r, i$)
 2:     if $r \leq 15$ then
 3:         yield $(r, i)$
 4:     else
 5:         for $(r', i')$ in $baseEquations_{r-16}$ do
 6:             yield $(r', (i' + i)\%32)$
 7: end function
```

for each position as 2-tuples, resulting in a 64-entry lookup table. Algorithm 4.2 shows such an algorithm and Appendix A provides the concrete data to be used as the *baseEquations* input. Each tuple obtained from the wTERMS function refers to a bit of the data-words which can be looked up easily or analysed further.

Table 4.1: Message-expansion bit-patterns (● = 1, · = 0)

w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	●		●						●					●		
17		●		●						●					●	
18			●		●						●					●
19	●		●	●		●			●			●		●		
20		●		●	●		●			●			●		●	
21			●		●	●		●			●			●		●
22	●		●	●		●	●		●·●			●		●	●	
23		●		●	●		●	●		●·●			●		●	●
24	●		●●		●	●		●	●●		●·●					●
25	●	●	●	●●		●	●		●·●	●●		●·●		●		
26		●	●	●	●●		●	●		●·●	●●		●·●		●	
27			●	●	●	●●		●	●		●·●	●●		●·●		●
28	●		●	●	●	●	●●		●·●	●		●·●	●●	●	●·●	
29		●		●	●	●	●	●●		●·●	●		●·●	●●	●	●·●
30	●·●		●●·●		●	●	●	●	●●		●·●	●			●●	●
31	●	●·●	●	●●·●		●	●	●	●·●	●●		●·●	●	●		●●
32	●●	●	●●	●	●●·●		●	●	●	●·●	●●		●·●	●	●	
33		●●	●	●●	●	●●·●		●	●	●	●·●	●●		●·●	●	●
34	●		●···●●	●	●●	●	●●·●		●·●	●	●	●·●	●●	●	●·●	●
35	●	●	●	●···●●	●	●●	●	●●·●	●	●·●	●	●	●·●	●	●	●·●
36	●·●	●	●●·●	●	●···●●	●	●●	●	●●··●	●	●·●	●	●		●	●
37	●	●·●	●···●	●●·●	●	●···●●	●	●●	●·●	●●··●	●	●·●	●	●····●		●
38	●	●	●	●···●	●●·●	●	●···●●	●	●	●·●	●●··●	●	●·●		●····●	
39		●	●	●	●···●	●●·●	●	●···●●	●	●	●·●	●●··●	●	●·●		●····●
40	●····●		●	●	●	●···●	●●·●	●	●·●··●●●	●	●	●·●	●●··●	●	●·●	
41		●····●		●	●	●	●···●	●●·●	●	●·●··●●●	●	●	●·●	●●··●	●	●·●
42	●·●		●●·●·●		●	●	●	●···●	●●··●	●	●·●··●●●	●	●		●●··●	●
43	●	●·●	●	●●·●·●		●	●	●	●·●···●	●●··●	●	●·●··●●●	●	●······●		●●··●
44	●●··●	●	●●···●	●	●●·●·●		●	●	●···●	●·●···●	●●··●	●	●·●··●●●	●···●	●······●	
45		●●··●	●	●●···●	●	●●·●·●		●	●	●···●	●·●···●	●●··●	●	●·●··●●●	●···●	●······●
46	●······●		●···●●·●●	●	●●···●	●	●●·●·●		●·●····●	●	●···●	●·●···●	●●··●	●	●·●··●●●	●···●
47	●···●	●······●	●···●	●···●●·●●	●	●●···●	●	●●·●·●	●···●	●·●····●	●	●···●	●·●···●	●	●	●·●··●●●
48	●·●··●●●	●···●	●●·●··●·●	●···●	●···●●·●●	●	●●···●	●	●●··●●·●	●···●	●·●····●	●	●···●	●·●	●	●
49	●	●·●··●●●	●···●···●	●●·●··●·●	●···●	●···●●·●●	●	●●···●	●·●	●●··●●·●	●···●	●·●····●	●	●····●···●	●·●	●
50	●	●	●····●●●	●···●···●	●●·●··●·●	●···●	●···●●·●●	●	●···●	●·●	●●··●●·●	●···●	●·●····●		●····●···●	●·●
51	●·●	●	●·····●·●	●····●●●	●···●···●	●●·●··●·●	●···●	●···●●·●●	●···●·●	●···●	●·●	●●··●●·●	●···●	●·●··●		●····●···●
52	●····●···●	●·●	●········●	●·····●·●	●····●●●	●···●···●	●●·●··●·●	●···●	●·●··●●●··●	●···●·●	●···●	●·●	●●··●●·●	●	●·●··●	
53		●····●···●	●·●	●········●	●·····●·●	●····●●●	●···●···●	●●·●··●·●	●···●	●·●··●●●··●	●···●·●	●···●	●·●	●●··●●·●	●	●·●··●
54	●·●··●		●●·●·●●··●	●·●	●········●	●·····●·●	●····●●●	●···●···●	●●··●●·●·●	●···●	●·●··●●●··●	●···●·●	●···●	●	●●··●●·●	●
55	●	●·●··●	●	●●·●·●●··●	●·●	●········●	●·····●·●	●····●●●	●·●···●···●	●●··●●·●·●	●···●	●·●··●●●··●	●···●·●	●······●···●	●	●●··●●·●
56	●●··●●·●	●	●●····●·●	●	●●·●·●●··●	●·●	●········●	●·····●·●	●···●●●·●	●·●···●···●	●●··●●·●·●	●···●	●·●··●●●··●	●···●	●······●···●	●
57	●	●●··●●·●	●·····●	●●····●·●	●	●●·●·●●··●	●·●	●········●	●···●·●·●	●···●●●·●	●·●···●···●	●●··●●·●·●	●···●	●·●···●●··●	●···●	●······●···●
58	●······●···●	●	●···●●·●●●	●·····●	●●····●·●	●	●●·●·●●··●	●·●	●·●····●	●···●·●·●	●···●●●·●	●·●···●···●	●●··●●·●·●	●	●·●···●●··●	●···●
59	●···●	●······●···●	●	●···●●·●●●	●·····●	●●····●·●	●	●●·●·●●··●	●···●·●·●	●·●····●	●···●·●·●	●···●●●·●	●·●···●···●	●···●·●·●	●	●·●···●●··●
60	●·●···●●··●	●···●	●●·●····●	●	●···●●·●●●	●·····●	●●····●·●	●	●●··●	●···●·●·●	●·●····●	●···●·●·●	●···●●●·●	●	●···●·●·●	●
61	●	●·●···●●··●	●···●···●	●●·●····●	●	●···●●·●●●	●·····●	●●····●·●	●·●	●●··●	●···●·●·●	●·●····●	●···●·●·●	●····●···●●●·●	●	●···●·●·●
62	●···●·●·●	●	●······●●	●···●···●	●●·●····●	●	●···●●·●●●	●·····●	●··●	●·●	●●··●	●···●·●·●	●·●····●		●····●···●●●·●	●
63	●	●···●·●·●	●·······●	●······●●	●···●···●	●●·●····●	●	●···●●·●●●	●	●··●	●·●	●●··●	●···●·●·●	●·●		●····●···●●●·●
64	●····●···●●●·●	●	●·········●	●·······●	●······●●	●···●···●	●●·●····●	●	●·●··●●●·····●	●	●··●	●·●	●●··●	●·········●	●·●	
65		●····●···●●●·●	●	●·········●	●·······●	●······●●	●···●···●	●●·●····●	●	●·●··●●●·····●	●	●··●	●·●	●●··●	●·········●	●·●
66	●·●		●●·●·●···●●●·●	●	●·········●	●·······●	●······●●	●···●···●	●●··●····●	●	●·●··●●●·····●	●	●··●		●●··●	●·········●
67	●·········●	●·●	●·········●	●●·●·●···●●●·●	●	●·········●	●·······●	●······●●	●·●···●	●●··●····●	●	●·●··●●●·····●	●	●······●		●●··●
68	●●··●	●·········●	●●···●	●·········●	●●·●·●···●●●·●	●	●·········●	●·······●	●···●···●●	●·●···●	●●··●····●	●	●·●··●●●·····●	●···●	●······●	
69		●●··●	●·········●	●●···●	●·········●	●●·●·●···●●●·●	●	●·········●	●·······●	●···●···●●	●·●···●	●●··●····●	●	●·●··●●●·····●	●···●	●······●
70	●······●		●···●●·●●	●·········●	●●···●	●·········●	●●·●·●···●●●·●	●	●·●····●····●	●·······●	●···●···●●	●·●···●	●●··●····●	●	●·●··●●●·····●	●···●
71	●···●	●······●	●···●	●···●●·●●	●·········●	●●···●	●·········●	●●·●·●···●●●·●	●···●···●	●·●····●····●	●·······●	●···●···●●	●·●···●	●·······●	●	●·●··●●●·····●
72	●·●··●●●·····●	●···●	●●·●··●·●·····●	●···●	●···●●·●●	●·········●	●●···●	●·········●	●●··●●·●··●●●●●	●···●···●	●·●····●····●	●·······●	●···●···●●	●·●·····●	●·······●	●
73	●	●·●··●●●·····●	●···●···●	●●·●··●·●·····●	●···●	●···●●·●●	●·········●	●●···●	●·●·········●	●●··●●·●··●●●●●	●···●···●	●·●····●····●	●·······●	●····●···●···●●	●·●·····●	●·······●
74	●·······●	●	●····●●●··●··●	●···●···●	●●·●··●·●·····●	●···●	●···●●·●●	●·········●	●···●··●	●·●·········●	●●··●●·●··●●●●●	●···●···●	●·●····●····●		●····●···●···●●	●·●·····●
75	●·●·····●	●·······●	●·····●·●·····●	●····●●●··●··●	●···●···●	●●·●··●·●·····●	●···●	●···●●·●●	●···●·●···●·●	●···●··●	●·●·········●	●●··●●·●··●●●●●	●···●···●	●·●··●······●●		●····●···●···●●
76	●····●···●···●●	●·●·····●	●········●····●	●·····●·●·····●	●····●●●··●··●	●···●···●	●●·●··●·●·····●	●···●	●·●··●●●··●··●●	●···●·●···●·●	●···●··●	●·●·········●	●●··●●·●··●●●●●	●·············●	●·●··●······●●	
77		●····●···●···●●	●·●·····●	●········●····●	●·····●·●·····●	●····●●●··●··●	●···●···●	●●·●··●·●·····●	●···●	●·●··●●●··●··●●	●···●·●···●·●	●···●··●	●·●·········●	●●··●●·●··●●●●●	●·············●	●·●··●······●●
78	●·●··●······●●		●●·●·●●··●	●·●·····●	●········●····●	●·····●·●·····●	●····●●●··●··●	●···●···●	●●··●●·●·●··●●·●	●···●	●·●··●●●··●··●●	●···●·●···●·●	●···●··●	●·······●	●●··●●·●··●●●●●	●·············●
79	●·············●	●·●··●······●●	●·············●	●●·●·●●··●	●·●·····●	●········●····●	●·····●·●·····●	●····●●●··●··●	●·●···●···●···●	●●··●●·●·●··●●·●	●···●	●·●··●●●··●··●●	●···●·●···●·●	●······●···●	●·······●	●●··●●·●··●●●●●

4.1.2 Common subexpressions

Bit-patterns indicate which bits of which words participate in which expansion-word calculations. They may also be used to understand inter-bit relationships by examining which bits of which words participate together, or co-occur, in expansion-word calculations.

Since message-expansion works with 32-bit values, every inter-word relationship will be repeated 32 times with an identical offset. For example, if (w_8^{31}, w_{11}^{30}) co-occur, then so will (w_8^{30}, w_{11}^{29}), (w_8^{29}, w_{11}^{28}), (w_8^{28}, w_{11}^{27}), (w_8^{27}, w_{11}^{26}), and so on for 27 additional co-occurrences. It adds little to the understanding of message-expansion to repeat such co-occurrences, and co-occurrences have therefore been represented as a *base round* followed by co-occurring round- and bit-offsets. The sequence that has been mentioned would thus be compactly represented as $11|^{+1}_{-3}$; and in general, an n-tuple co-occurrence would be represented as $base|^{\pm i_0}_{\pm p_0}|\cdots|^{\pm i_n}_{\pm p_n}$.

Table 4.2: Co-occurring 2-tuples during message-expansion

Participation	Co-occurrences
98	$11|^{+1}_{-3}$
93	$12|^{+1}_{-3}$
81	$2|^{+0}_{-2}$
79	$3|^{+0}_{-2}$
75	$8|^{+0}_{-8}$
74	$8|^{+0}_{-6}$, $5|^{+1}_{-3}$, $13|^{+1}_{-3}$
73	$9|^{+0}_{-8}$
72	$9|^{+0}_{-6}$, $10|^{+0}_{-8}$, $11|^{+0}_{-8}$, $6|^{+1}_{-3}$, $14|^{+1}_{-3}$
70	$4|^{+0}_{-2}$, $5|^{+0}_{-2}$, $7|^{+1}_{-3}$, $15|^{+1}_{-3}$
69	$10|^{+0}_{-6}$, $12|^{+0}_{-8}$, $10|^{+0}_{-2}$
68	$11|^{+0}_{-6}$, $6|^{+0}_{-2}$
67	$8|^{+1}_{-3}$, $11|^{+0}_{-2}$
66	$12|^{+0}_{-6}$, $9|^{+1}_{-3}$
65..1	elided for brevity

Table 4.2 lists 2-tuples, in order of the number of expansion-word calculations that they appear in, and Tables 4.3 and 4.4 do the same for 3- and 4-tuples. The "Participation" column of these tables shows the number of calculations that a particular co-occurrence participates in. This number may be greater than the number of expansion-words (i.e. 64) since there may be multiple co-occurrences per calculation. For example, the calculation of w_{25}^0 involves w_8^2, w_8^4, w_{11}^1, and w_{11}^3. The relationships (w_8^2, w_{11}^1) and (w_8^4, w_{11}^3) can both

Table 4.3: Co-occurring 3-tuples during message-expansion

Participation	Co-occurrences
77	$14\vert^{+1}_{-3}\vert^{+1}_{-6}$
74	$14\vert^{+1}_{-3}\vert^{+3}_{-4}$, $13\vert^{+1}_{-3}\vert^{+2}_{-4}$
72	$15\vert^{+1}_{-3}\vert^{+4}_{-4}$, $15\vert^{+1}_{-3}\vert^{+2}_{-6}$, $14\vert^{+1}_{-3}\vert^{+2}_{-5}$, $13\vert^{+1}_{-3}\vert^{+1}_{-5}$
70	$13\vert^{+1}_{-3}\vert^{-2}_{-8}$, $14\vert^{+1}_{-3}\vert^{-1}_{-8}$
68	$13\vert^{+1}_{-3}\vert^{-4}_{-10}$, $14\vert^{+1}_{-3}\vert^{-3}_{-10}$
66	$15\vert^{+1}_{-3}\vert^{+0}_{-8}$, $14\vert^{+1}_{-3}\vert^{-5}_{-12}$
65	$15\vert^{+1}_{-3}\vert^{-2}_{-10}$
64	$15\vert^{+1}_{-3}\vert^{-4}_{-12}$
63..1	elided for brevity

Table 4.4: Co-occurring 4-tuples during message-expansion

Participation	Co-occurrences
44	$11\vert^{+1}_{-3}\vert^{+2}_{-6}\vert^{+3}_{-9}$
43	$12\vert^{+1}_{-3}\vert^{+2}_{-6}\vert^{+3}_{-9}$
39	$8\vert^{-2}_{-2}\vert^{+0}_{-6}\vert^{+0}_{-8}$, $11\vert^{+1}_{-3}\vert^{+0}_{-6}\vert^{+1}_{-9}$, $13\vert^{+0}_{-5}\vert^{+0}_{-11}\vert^{+0}_{-13}$
38	$8\vert^{+0}_{-2}\vert^{+1}_{-3}\vert^{+1}_{-5}$, $14\vert^{+0}_{-5}\vert^{+0}_{-11}\vert^{+0}_{-13}$
37	$13\vert^{-1}_{-2}\vert^{-1}_{-8}\vert^{-1}_{-10}$, $8\vert^{-1}_{-5}\vert^{+0}_{-6}\vert^{+0}_{-8}$, $9\vert^{-2}_{-2}\vert^{+0}_{-6}\vert^{+0}_{-8}$, $11\vert^{+1}_{-3}\vert^{+0}_{-6}\vert^{+1}_{-11}$, $11\vert^{+1}_{-3}\vert^{+0}_{-6}\vert^{+0}_{-8}$, $12\vert^{+1}_{-3}\vert^{+0}_{-6}\vert^{+1}_{-9}$, $9\vert^{+0}_{-2}\vert^{+1}_{-3}\vert^{+1}_{-5}$, $7\vert^{+0}_{-2}\vert^{+1}_{-3}\vert^{+1}_{-5}$
36	$14\vert^{-1}_{-2}\vert^{-1}_{-8}\vert^{-1}_{-10}$, $9\vert^{-1}_{-5}\vert^{+0}_{-6}\vert^{+0}_{-8}$, $8\vert^{-1}_{-3}\vert^{+0}_{-6}\vert^{+0}_{-8}$, $11\vert^{+0}_{-2}\vert^{+1}_{-3}\vert^{+1}_{-5}$, $12\vert^{+1}_{-3}\vert^{+0}_{-6}\vert^{+0}_{-8}$, $11\vert^{+0}_{-6}\vert^{+0}_{-8}\vert^{+1}_{-9}$, $5\vert^{+0}_{-2}\vert^{+1}_{-3}\vert^{+1}_{-5}$, $11\vert^{+0}_{-6}\vert^{+1}_{-9}\vert^{+1}_{-11}$, $11\vert^{+1}_{-3}\vert^{+1}_{-9}\vert^{+1}_{-11}$, $11\vert^{+1}_{-3}\vert^{+0}_{-8}\vert^{+1}_{-11}$
35	$8\vert^{-1}_{-3}\vert^{-1}_{-5}\vert^{+0}_{-8}$, $10\vert^{+0}_{-2}\vert^{+1}_{-3}\vert^{+2}_{-6}$, $11\vert^{+0}_{-2}\vert^{+1}_{-3}\vert^{+2}_{-6}$, $12\vert^{+1}_{-3}\vert^{+0}_{-6}\vert^{+1}_{-11}$, $6\vert^{+0}_{-2}\vert^{+1}_{-3}\vert^{+1}_{-5}$, $12\vert^{+1}_{-3}\vert^{+0}_{-8}\vert^{+1}_{-11}$, $11\vert^{+0}_{-6}\vert^{+0}_{-8}\vert^{+1}_{-11}$, $12\vert^{+0}_{-6}\vert^{+0}_{-8}\vert^{+1}_{-9}$, $15\vert^{+0}_{-5}\vert^{+0}_{-11}\vert^{+0}_{-13}$
34..1	elided for brevity

be represented by the co-occurrence $11|^{+1}_{-3}$, and this co-occurrence is therefore counted twice for the calculation of w_{25}.

Many of the inter-word relationships are simply a restatement of the existing spacings of 2, 3, 5, 6, 8, 11, 13, 14, and 16 which are a consequence of the offsets used for message-expansion calculation in the canonical SHA-1 algorithm. Some, however, are not; for example, neither $14|^{+1}_{-3}|^{+3}_{-4}$'s round offset nor $2|^{-2}_{+0}$'s bit offset is a obvious spacing.

An understanding of co-occurrences makes it easy to create additional intermediate words which simplify expansion-word calculations. These intermediate words are not necessarily identical to the words used during traditional message-expansion calculations. The length of the tuple that co-occurs and the participation of the tuple indicate the decrease in bitpattern components that is possible, if an intermediate word representing that tuple is generated. For example, Example 4.4 shows that w_{79}^{0} involves

$$\ldots \oplus w_{11}^{11} \oplus w_{12}^{8} \oplus w_{15}^{7} \oplus w_{11}^{15} \oplus w_{12}^{12} \oplus w_{15}^{11} \oplus w_{11}^{21} \oplus w_{12}^{18} \oplus w_{15}^{17} \oplus \ldots$$

A quick examination reveals that this is equivalent to three occurrences of the 3-tuple $15|_{-3}^{+1}|_{-4}^{+4}$, which participates 72 times in SHA-1 message-expansion (see Table 4.3). A 3-tuple a can represent the exclusive-or of three words, $b \oplus c \oplus d$, reducing the number of exclusive-or operations by 2 each time that it us used. If one creates an intermediate word $w' = w_{15} \oplus (w_{12} \hookleftarrow 3) \oplus (w_{11} \hookleftarrow 4)$, then the specified portion of w_{79}^{0}'s equation can be represented as $w'_{7} \oplus w'_{11} \oplus w'_{17}$ instead, reducing 8 exclusive-or operations to $8 - 2 \times 3 = 2$. The intermediate word w' can still be used a further $72 - 3 = 69$ times during SHA-1 calculation, reducing the number of exclusive-or operations by an additional amount of $69 \times 2 = 138$.

Although it is possible to generate exhaustive lists of co-occurring n-tuples, the utility of doing so decreases sharply as n increases. This is because the participation of co-occurring tuples decreases as n gets larger, as can be seen from Tables 4.2, 4.3 and 4.4.

4.1.3 Inversion

For a particular expansion-word w_r, the data-word bits that could affect the expansion-word are given by expanding equation 4.1. Since bits in the expansion-word are determined through XORing, a 1-bit in an expansion-word must be due to an odd number of relevant data-word bits being set. Similarly, an 0-bit in an expansion-word must be due to an even number of relevant data-word bits being set.

By going through the bits of any concrete expansion-word, we can obtain a set of linear equations which we have called *expansion-equations*. Any set of data-word bits which satisfies all of these expansion-equations will result in that concrete expansion-word. A simple Davis-Putnam-Logemann-Loveland-style backtracking algorithm (Davis, Logemann, and Loveland, 1962) is sufficient to enumerate all possible data-word values. Alternatively, the data-word bits can be regarded as a boolean vector that should have either an odd (if the result should be true) or even (if the result should be false) Hamming weight. Setting the bits can be done with very little overhead, and it is easy to see that a total of 2^{n-1} suitable combinations exist for every n-bit vector.

Padding and length-extension occur before message-expansion and affect the data-words to be processed by message-expansion. Therefore, while every solution to a set of expansion-equations is a valid solution, it is not true that every solution is a valid set of single-block SHA-1 data-words. For example, a solution that has $w_{15} = 0$, $w_{14} = 0$, and $w_0 \neq 0$ cannot be valid: the length (as encoded in $w_{14,15}$) indicates that the data is zero bits long, but w_0 contradicts this. While not strictly necessary, it is interesting to find an efficient way to choose only valid solutions. Such solutions must satisfy the following conditions:

1. Each block is 512 bits long, with the exception of the last block, which may be up to 447 bits long. Let *len* be the total length of the input, as encoded in $w_{14,15}$ of the last block. Since $0 \leq len \leq 447$, $w_{14} = 0$, $w_{15} \wedge 11111111111111111111111000000000\text{b} = 0$, and the terminator bit must be found at w'_{len}.

2. The bit at the terminator position must be 1.

3. All bits between the terminator position and w_{15} must be 0.

One way to ensure compliance with these conditions is to discard any solution which doesn't meet the conditions. A better way can be found by observing that message-expansion is achieved by XORing values together, and neither terminator nor padding nor encoded-length are treated specially in this regard. Therefore, we can use the following steps to create expansion-equations which must result in a valid solution:

1. For an expansion word w_i which is expanded from a data-word block of length *len*, we can create a "template" set of data-words with the length and terminator bits filled-in and use it to calculate an expansion-word $w_{i'}$. Let $w_{i''} = w_i \oplus w_{i'}$. The modified expansion-word $w_{i''}$ is equivalent to an expansion-word generated from correctly-terminated data-words where $w_{14} = 0$ and $w_{15} = len$, and expansion-equations based on it are equivalent to expansion-equations that took the terminator and length into account.

2. Generate expansion-equations for each bit in $w_{i''}$. Modify each expansion-equation to only contain values in the range $[0..len)$. If there are no expansion-equation terms within that range, then

   (a) if the result of an expansion-equation is 0, then the expansion-equation can be left out entirely.

   (b) if the result of an expansion-equation is 1, then there is no possible combination of terms within $[0..len)$ that can result in the desired expansion-word.

4.2 Spliffling analysis

The SHA-1 compression function applies a series of operations over the course of 80 rounds. In theory, if the operations to be applied are known, then it should possible to find a single equation which represents the end result of *all* the operations.

As noted in the literature (Lien *et al.*, 2004; De Canniere and Rechberger, 2008), the standard formulation of spliffling (see Algorithm 2.1 on page 18) can be streamlined somewhat. Whereas the standard formulation uses a, b, c, d, and e variables, only the a variable is necessary. The a value is the only one that undergoes a destructive change from one round to the next: the new b, d, and e values merely copy values that already exist, and the new c value is a left-rotated b value, which is a non-destructive operation. In other words, for a round r:

$$\begin{aligned} b_r &= a_{r-2} \\ c_r &= a_{r-3} \hookleftarrow 30 \\ d_r &= a_{r-4} \hookleftarrow 30 \\ e_r &= a_{r-5} \hookleftarrow 30 \end{aligned}$$

We can therefore obtain an equation similar to De Canniere and Rechberger's (2008) for a_r:

$$\begin{aligned} a_r &= (a_{r-1} \hookleftarrow 5) + f(b_{r-1}, c_{r-1}, d_{r-1}]) + e_{r-1} + k_{r-1} + w_{r-1} \\ &= (a_{r-1} \hookleftarrow 5) + f(a_{r-2}, a_{r-3} \hookleftarrow 30, a_{r-4} \hookleftarrow 30) + a_{r-5} \hookleftarrow 30 + k_{r-1} + w_{r-1} \end{aligned} \tag{4.4}$$

Consider the operations that occur to obtain the value of a^i_r. Based on Equation 4.4 and taking into account the necessary rotations, an initial formulation may look like

$$\begin{aligned} a^i_r &= a^{i+5}_{r-1} + f^{i+5}_{r-1}(b^{i+5}_{r-1}, c^{i+5}_{r-1}, d^{i+5}_{r-1}) + e^{i+5}_{r-1} + k^{i+5}_{r-1} + w^{i+5}_{r-1} \\ &= a^{i+5}_{r-1} + f^{i+5}_{r-1}(a^{i}_{r-2}, a^{i+30}_{r-3}, a^{i+30}_{r-4}) + a^{i+30}_{r-5} + k^{i+5}_{r-1} + w^{i+5}_{r-1} \end{aligned} \tag{4.5}$$

4.2.1 Carry calculations

When bits are added in a column to arrive at the value of a^i_r, carries into one or more columns $a^{i-1}_r, ..., a^{i-m}_r$ may occur. The number of columns directly affected by the carry from a single column is one less than the bit-length of the maximum possible result of a column addition. We give the name $v0^i_r$ to the carry-digit generated by column a^{i+1}_r which affects column a^i_r; $v1^i_r$ is the carry-digit generated by column a^{i+2}_r which affects column a^i_r; and so on.

Example 4.5. *$v0$ and $v1$ values.* To see the use of $v0$ and $v1$ values in a more visual way, consider the following example of adding five binary numbers where each $v0$ and $v1$ value has been colored to be the same as the column that resulted in its creation.

2^5	2^4	2^3	2^2	2^1	2^0	column value
1	1	1	0	0	0	$v0$
0	1	1	1	0	0	$v1$
			1	1	1	
			1	1	1	
			1	1	1	
			1	1	1	
			1	1	1	
1	0	0	0	1	1	result

The value of $v0^i_r$ depends on a^{i+1}_r for all values of i except for 26. This is because when $i = 27$, Equation 4.6 is $a^{27}_r = a^0_{r-1} + f^0_{r-1} + e^0_{r-1} + k^0_{r-1} + w^0_{r-1}$ and it can be readily seen that this is an equation involving the most significant bits of these words. Since any carries from this position is irrelevant, $v0^{26}_r = 0$. The same reasoning makes it clear that $v1^{26}_r = 0$ and $v1^{25}_r = 0$.

There are five bits, obtained from the respective values of a, f, e, k, and w, that participate in the addition in equation 4.6. The maximum possible result of adding the appropriate bits to generate a^{26}_r is therefore 5 (101b). The bit-length of 101b is 3; therefore, the maximum number of columns that can be directly affected by the carry from a single column is 2. However, $v0^{24}_r$ and $v1^{24}_r$ are potentially non-zero, and so the calculation of a^{24}_r must account for bits to arrive at a maximum possible value of 7 (111b). Happily, the bit-length of this is 3, and so the maximum number of columns that can be directly affected by the carry from a single column is still 2.

When will $v0^i_r$ be 1? The result of the appropriate addition must be either 10b, 11b, 110b, or 111b for this to happen; or, equivalently, we can say that exactly 2, 3, 6, or 7 of the bits must be 1. If we take the $\binom{|S|}{n}$ combinations of a set of boolean variables S, and say that for each combination U, let U' be the complementary set $\neg(S \setminus U)$, then by joining all elements of each U and U' using $\wedge$ it is trivial to obtain individual explicit formulae for each case where exactly $\frac{n}{|S|}$ bits are set. Let $q(n, S)$ be a function which generates such formulae and joins each formula to the next with $\vee$. Now we can describe a function $v0(r, i)$, which calculates a $v0_r^i$ value, as

$$\begin{aligned}
v0(r, 26) &= 0 \\
v0(r, 25) &= q(2, S) \vee q(3, S) \\
&\quad \text{where } S = \{a_r^{26}, f_r^{26}, e_r^{26}, k_r^{26}, w_r^{26}\} \\
v0(r, i \notin \{25, 26\}) &= q(2, S) \vee q(3, S) \vee q(6, S) \vee q(7, S) \\
&\quad \text{where } S = \{a_r^{i+1}, f_r^{i+1}, e_r^{i+1}, k_r^{i+1}, w_r^{i+1}, v0_r^{i+1}, v1_r^{i+1}\}
\end{aligned}$$

A similar argument applies to the question of when $v1_r^i$ will be 1. The result of the appropriate addition must be either 100b, 101b, 110b, or 111b; or, equivalently, we can say that exactly 4, 5, 6, or 7 of the bits must be 1. Therefore,

$$\begin{aligned}
v1(r, i \in \{25, 26\}) &= 0 \\
v1(r, 24) &= q(4, S) \vee q(5, S) \\
&\quad \text{where } S = \{a_r^{26}, f_r^{26}, e_r^{26}, k_r^{26}, w_r^{26}\} \\
v1(r, i \notin \{24..26\}) &= q(4, S) \vee q(5, S) \vee q(6, S) \vee q(7, S) \\
&\quad \text{where } S = \{a_r^{i+2}, f_r^{i+2}, e_r^{i+2}, k_r^{i+2}, w_r^{i+2}, v0_r^{i+2}, v1_r^{i+2}\}
\end{aligned}$$

The basic idea presented above is to model carries separately from additions, and then reintegrate them into the calculation at a later stage. The idea is not unique, and should be familiar to a student who has studied basic circuit design as a ripple-carry adder. Legendre *et al.* (2012) describe essentially the same model of addition, arriving at it in a different way and pointing out that it is a model which uses the fewest number of variables out of all the considered alternatives, and Nossum (2013) represents the constraints algebraically with the same ripple-carry design being the result.

A great deal of work has examined the probabilities of carries during addition, given particular conditions, and their cryptographic significance from the collision perspective. A representative précis of this work is that 5-operand addition is a biased operation that tends to generate a carry slightly more often (a $+13\frac{1}{3}\%$ probability) than not (Staffelbach and Meier, 1991); efficient algorithms exist to calculate the differential properties of addition (Lipmaa and Moriai, 2001); and an analysis of the probability of a run of carries is quite possible (Borodin, Diaconis, and Fulman, 2010). From the perspective of SHA-1 preimage-resistance, the probabilities involved in 5-ary addition provide an edge that is too slight to lead directly to a practical preimage attack; however, they may be useful for guiding the actions of a constraint solver (see Chapter 11).

Optimisation: accounting for known k

The value of a k bit is known beforehand. Following on from our reasoning in the previous section, if $k = 0$ then the k value cannot affect any carry and we need not consider it as part of S. Conversely, if $k = 1$ then there is are fewer bits in S that are needed for a carry to occur; therefore, we can remove k from S and reduce the n parameter of the q function by 1.

We remove k from S and add it as a parameter to the existing (r, i) parameters of $v0$ and $v1$ functions. After making this change, the $v0(r, i, k)$ functions look like

$$\begin{aligned}
v0(r,26,k) &= 0 \\[1ex]
v0(r,25,0) &= q(2,S) \vee q(3,S) \\
v0(r,25,1) &= q(1,S) \vee q(2,S) \\
&\ \text{where } S = \{a_r^{26}, f_r^{26}, e_r^{26}, w_r^{26}\} \\[1ex]
v0(r, i \notin \{25,26\}, 0) &= q(2,S) \vee q(3,S) \vee q(6,S) \\
v0(r, i \notin \{25,26\}, 1) &= q(1,S) \vee q(2,S) \vee q(5,S) \vee q(6,S) \\
&\ \text{where } S = \{a_r^{i+1}, f_r^{i+1}, e_r^{i+1}, w_r^{i+1}, v0_r^{i+1}, v1_r^{i+1}\}
\end{aligned}$$

Similarly-modified equations can be obtained for $v1(r, i, k)$.

This optimisation reduces the number of equations that need to be combined. For example, $q(2,S) \vee q(3,S) \vee q(6,S) \vee q(7,S)$ when $|S| = 7$ means that $\binom{7}{7} + \binom{7}{6} + \binom{7}{3} + \binom{7}{2} = 64$ equations must be combined. Since k is known, we can remove it from S so that $|S| = 6$. Then if $k = 0$, we need only combine $\binom{6}{2} + \binom{6}{3} + \binom{6}{6} = 36$ equations; and if $k = 1$, we need only combine $\binom{6}{1} + \binom{6}{2} + \binom{6}{5} + \binom{6}{6} = 28$ equations.

Optimisation: minimising equations

A second optimisation can be realised by minimising equations. Given an equation such as $q(n_0, S) \vee q(n_1, S) \vee \ldots \vee q(n_m, S) \vee q(6, S)$, we can use boolean function minimisation to obtain a minimal form for this equation. The Espresso algorithm (Brayton, Sangiovanni-Vincentelli, McMullen, and Hachtel, 1984), as implemented by Logic Friday[2], was used to find exact minimal forms. Assuming suitable $a, f, e, w, v0, v1$ values in S, the resulting non-constant $v0(r, i, k)$ and $v1(r, i, k)$ equations are as follows:

$$\begin{aligned}
v0(r,25,0) &= f \wedge (a \wedge \neg w \vee \neg e \wedge w \vee \neg a \wedge e) \vee \neg f \wedge (w \wedge (e \vee a) \vee a \wedge e) \\
v0(r,25,1) &= \neg a \wedge (f \wedge \neg w \vee \neg e \wedge w \vee e \wedge \neg f) \vee a \wedge (\neg w \wedge (\neg f \vee \neg e) \vee \neg e \wedge \neg f)
\end{aligned}$$

$$\begin{aligned}
v0(r, i \notin \{25,26\}, 0) &= \neg a \wedge \neg e \wedge \neg v0 \wedge v1 \wedge w \vee f \wedge (a \wedge e \wedge v0 \wedge v1 \wedge w \vee \neg w \wedge (a \wedge \neg e \wedge \\
&\quad \neg v0 \vee (a \wedge \neg v1 \vee \neg a \wedge v1) \wedge \neg(v0 \vee \neg e)) \vee \neg v1 \wedge (a \wedge \neg e \wedge \neg v0 \vee \\
&\quad \neg a \wedge (v0 \wedge \neg w \vee \neg e \wedge w \vee e \wedge \neg v0))) \vee \neg f \wedge \neg(a \wedge v0 \wedge \neg v1 \wedge w \vee \\
&\quad \neg e \wedge \neg(a \wedge v0 \wedge v1 \vee a \wedge (v0 \wedge \neg w \vee \neg v1 \wedge w \vee \neg v0 \wedge v1)) \vee e \wedge \\
&\quad \neg(v0 \wedge \neg(a \wedge w \vee a \wedge \neg v1) \vee \neg w \wedge (v1 \wedge \neg(v0 \vee \neg a) \vee v0 \wedge \neg v1))) \\
v0(r, i \notin \{25,26\}, 1) &= a \wedge f \wedge v0 \wedge v1 \wedge w \vee \neg e \wedge \neg(a \wedge \neg f \wedge v1 \wedge \neg w \vee \neg v0 \wedge \neg(a \wedge \neg f \wedge \\
&\quad w \vee \neg w \wedge (a \oplus f)) \vee \neg v1 \wedge \neg(f \wedge (a \wedge \neg(w \vee \neg v0) \vee \neg a \wedge v0) \vee \\
&\quad f \wedge \neg(w \wedge \neg(v0 \vee \neg a) \vee \neg a \wedge \neg v0))) \vee e \wedge \neg(a \wedge \neg f \wedge \neg v1 \wedge \neg w \vee \\
&\quad a \wedge f \wedge v1 \wedge w \vee v0 \wedge (a \wedge f \wedge w \vee v1 \wedge (w \wedge (f \vee a) \vee a \wedge f)) \vee \\
&\quad \neg v0 \wedge \neg(a \wedge \neg f \wedge \neg w \vee \neg v1 \wedge \neg(w \wedge \neg(f \vee \neg a) \vee \neg a \wedge \neg f))) \\
v1(r,24,0) &= a \wedge f \wedge e \wedge w \\
v1(r,24,1) &= a \wedge e \wedge w \vee f \wedge (w \wedge (e \vee a) \vee a \wedge e)
\end{aligned}$$

$$\begin{aligned}
v1(r, i \notin \{24..26\}, 0) &= a \wedge e \wedge v1 \wedge w \vee f \wedge (a \wedge e \wedge w \vee v1 \wedge (w \wedge (e \vee a) \vee a \wedge e)) \vee v0 \wedge (a \wedge e \wedge \\
&\quad w \vee f \wedge (w \wedge (e \vee a) \vee a \wedge e) \vee v1 \wedge (f \wedge (e \vee a) \vee w \wedge (f \vee e \vee a) \vee a \wedge e)) \\
v1(r, i \notin \{24..26\}, 1) &= a \wedge e \wedge w \vee f \wedge (w \wedge (e \vee a) \vee a \wedge e) \vee v0 \wedge (f \wedge (e \vee a) \vee w \wedge (f \vee e \vee a) \vee \\
&\quad a \wedge e) \vee v1 \wedge (f \wedge (e \vee a) \vee v0 \wedge (f \vee e \vee a) \vee w \wedge (v0 \vee f \vee e \vee a) \vee a \wedge e)
\end{aligned}$$

4.2.2 Component functions

As briefly discussed in Section 2.4, the component f-equations are "choice", "majority", and "parity". Their truth tables are given as Table 2.2 (p. 19).

[2]Homepage: `http://sontrak.com/`

The f-value is calculated using b, c, and d values — which are, as discussed above, merely previous incarnations of a values — and therefore relies upon a comprehensive equation for a to have any sensible meaning. Assume that such an equation exists for the moment; it will be fully specified in Section 4.2.3. The f-value (as distinct from the f-function) is calculated before the additions of equation 4.5, and it is therefore possible (and usually convenient) to regard f as a single value rather than a function. The f-function and the f-value are distinguished by always following the former with its parameters in parentheses.

A more readable formulation of equation 4.5 is thus

$$\begin{aligned} a^i_r &= a^{i+5}_{r-1} + f^{i+5}_{r-1} + e^{i+5}_{r-1} + k^{i+5}_{r-1} + w^{i+5}_{r-1} \\ &= a^{i+5}_{r-1} + f^{i+5}_{r-1} + a^{i+30}_{r-5} + k^{i+5}_{r-1} + w^{i+5}_{r-1} \end{aligned} \tag{4.6}$$

Five-operand addition can be represented as the function $+ : \{0,1\}^5 \rightarrow \{0,1\}^3$, with the least significant bit being the "sum" bit and the remaining two being "carry" bits. The $\{0,1\}^3$ vector will be denoted by s in this section and, by an abuse of notation, the function which generates each bit will be denoted in the same way. The truth tables for 5-ary addition are given as Table 4.5.

Table 4.5: 5-ary addition truth tables

s_0	`01101001100101101001011001101001`
s_1	`00010111011111100111111011101000`
s_2	`00000000000000010000000100010111`

The properties of these boolean functions will be considered in this section; Chapter 8 of Preneel's 1993 thesis, as mentioned in Section 3.3 gives an overview of most of the important features that should be examined and Braeken (2006) provides a more up-to-date (but narrower) exploration. A boolean function can be stated in many ways. Some of those ways are "canonical": when two functions which may have different initial formulations are formulated in the same canonical way, then they are guaranteed to be identical. The most important canonical forms — Algebraic Normal Form (ANF), Conjunctive Normal Form (CNF), and Disjunctive Normal Form (DNF) — are presented as Table 4.6.

Table 4.6: Canonical forms of component functions

	ANF	CNF	DNF
Choice	$(b \wedge c) \oplus (b \wedge d) \oplus d$	$(\neg b \vee c) \wedge (b \vee d)$	$(b \wedge c) \vee (\neg b \wedge d)$
Majority	$(b \wedge c) \oplus (b \wedge d) \oplus (c \wedge d)$	$(b \vee c) \wedge (b \vee d) \wedge (c \vee d)$	$(b \wedge c) \vee (b \wedge d) \vee (c \wedge d)$
Parity	$b \oplus c \oplus d$	$(\neg b \vee \neg c \vee d) \wedge (\neg b \vee c \vee \neg d) \wedge (b \vee \neg c \vee \neg d) \wedge (b \vee c \vee d)$	$(b \wedge c \wedge d) \vee (b \wedge \neg c \wedge \neg d) \vee (\neg b \wedge c \wedge \neg d) \vee (\neg b \wedge \neg c \wedge d)$
s_0	$i_0 \oplus i_1 \oplus i_2 \oplus i_3 \oplus i_4$	$(i_0 \vee \neg i_1 \vee \neg i_2 \vee \neg i_3 \vee \neg i_4) \wedge (\neg i_0 \vee i_1 \vee \neg i_2 \vee \neg i_3 \vee \neg i_4) \wedge (\neg i_0 \vee \neg i_1 \vee i_2 \vee \neg i_3 \vee \neg i_4) \wedge (i_0 \vee i_1 \vee i_2 \vee \neg i_3 \vee \neg i_4) \wedge (\neg i_0 \vee \neg i_1 \vee \neg i_2 \vee i_3 \vee \neg i_4) \wedge (i_0 \vee i_1 \vee \neg i_2 \vee i_3 \vee \neg i_4) \wedge (i_0 \vee \neg i_1 \vee i_2 \vee i_3 \vee \neg i_4) \wedge (\neg i_0 \vee i_1 \vee i_2 \vee i_3 \vee \neg i_4) \wedge (\neg i_0 \vee \neg i_1 \vee \neg i_2 \vee \neg i_3 \vee i_4) \wedge (i_0 \vee i_1 \vee \neg i_2 \vee \neg i_3 \vee i_4) \wedge (i_0 \vee \neg i_1 \vee i_2 \vee \neg i_3 \vee i_4) \wedge (\neg i_0 \vee i_1 \vee i_2 \vee \neg i_3 \vee i_4) \wedge (i_0 \vee \neg i_1 \vee \neg i_2 \vee i_3 \vee i_4) \wedge (\neg i_0 \vee i_1 \vee \neg i_2 \vee i_3 \vee i_4) \wedge (\neg i_0 \vee \neg i_1 \vee i_2 \vee i_3 \vee i_4) \wedge (i_0 \vee i_1 \vee i_2 \vee i_3 \vee i_4)$	$(i_0 \wedge i_1 \wedge i_2 \wedge i_3 \wedge i_4) \vee (\neg i_0 \wedge \neg i_1 \wedge i_2 \wedge i_3 \wedge i_4) \vee (\neg i_0 \wedge i_1 \wedge \neg i_2 \wedge i_3 \wedge i_4) \vee (i_0 \wedge \neg i_1 \wedge \neg i_2 \wedge i_3 \wedge i_4) \vee (\neg i_0 \wedge i_1 \wedge i_2 \wedge \neg i_3 \wedge i_4) \vee (i_0 \wedge \neg i_1 \wedge i_2 \wedge \neg i_3 \wedge i_4) \vee (i_0 \wedge i_1 \wedge \neg i_2 \wedge \neg i_3 \wedge i_4) \vee (\neg i_0 \wedge \neg i_1 \wedge \neg i_2 \wedge \neg i_3 \wedge i_4) \vee (\neg i_0 \wedge i_1 \wedge i_2 \wedge i_3 \wedge \neg i_4) \vee (i_0 \wedge \neg i_1 \wedge i_2 \wedge i_3 \wedge \neg i_4) \vee (i_0 \wedge i_1 \wedge \neg i_2 \wedge i_3 \wedge \neg i_4) \vee (\neg i_0 \wedge \neg i_1 \wedge \neg i_2 \wedge i_3 \wedge \neg i_4) \vee (i_0 \wedge i_1 \wedge i_2 \wedge \neg i_3 \wedge \neg i_4) \vee (\neg i_0 \wedge \neg i_1 \wedge i_2 \wedge \neg i_3 \wedge \neg i_4) \vee (\neg i_0 \wedge i_1 \wedge \neg i_2 \wedge \neg i_3 \wedge \neg i_4) \vee (i_0 \wedge \neg i_1 \wedge \neg i_2 \wedge \neg i_3 \wedge \neg i_4)$
s_1	$(i_0 \wedge i_1) \oplus (i_0 \wedge i_2) \oplus (i_0 \wedge i_3) \oplus (i_0 \wedge i_4) \oplus (i_1 \wedge i_2) \oplus (i_1 \wedge i_3) \oplus (i_1 \wedge i_4) \oplus (i_2 \wedge i_3) \oplus (i_2 \wedge i_4) \oplus (i_3 \wedge i_4)$	$(\neg i_1 \vee \neg i_2 \vee \neg i_3 \vee \neg i_4) \wedge (\neg i_0 \vee \neg i_2 \vee \neg i_3 \vee \neg i_4) \wedge (\neg i_0 \vee \neg i_1 \vee \neg i_3 \vee \neg i_4) \wedge (\neg i_0 \vee \neg i_1 \vee \neg i_2 \vee \neg i_4) \wedge (i_0 \vee i_1 \vee i_2 \vee i_3) \wedge (\neg i_0 \vee \neg i_1 \vee \neg i_2 \vee \neg i_3) \wedge (i_0 \vee i_1 \vee i_2 \vee i_4) \wedge (i_0 \vee i_1 \vee i_3 \vee i_4) \wedge (i_0 \vee i_2 \vee i_3 \vee i_4) \wedge (i_1 \vee i_2 \vee i_3 \vee i_4)$	$(\neg i_0 \wedge \neg i_1 \wedge i_2 \wedge i_3) \vee (i_0 \wedge i_1 \wedge \neg i_2 \wedge \neg i_3) \vee (\neg i_0 \wedge i_1 \wedge \neg i_2 \wedge i_4) \vee (\neg i_1 \wedge \neg i_2 \wedge i_3 \wedge i_4) \vee (i_0 \wedge \neg i_1 \wedge \neg i_3 \wedge i_4) \vee (\neg i_0 \wedge i_2 \wedge \neg i_3 \wedge i_4) \vee (\neg i_0 \wedge i_1 \wedge i_2 \wedge \neg i_4) \vee (i_0 \wedge \neg i_1 \wedge i_3 \wedge \neg i_4) \vee (i_1 \wedge \neg i_2 \wedge i_3 \wedge \neg i_4) \vee (i_0 \wedge i_2 \wedge \neg i_3 \wedge \neg i_4)$
s_2	$(i_0 \wedge i_1 \wedge i_2 \wedge i_3) \oplus (i_0 \wedge i_1 \wedge i_2 \wedge i_4) \oplus (i_0 \wedge i_1 \wedge i_3 \wedge i_4) \oplus (i_0 \wedge i_2 \wedge i_3 \wedge i_4) \oplus (i_1 \wedge i_2 \wedge i_3 \wedge i_4)$	$(i_0 \vee i_1) \wedge (i_0 \vee i_2) \wedge (i_1 \vee i_2) \wedge (i_0 \vee i_3) \wedge (i_1 \vee i_3) \wedge (i_2 \vee i_3) \wedge (i_0 \vee i_4) \wedge (i_1 \vee i_4) \wedge (i_2 \vee i_4) \wedge (i_3 \vee i_4)$	$(i_0 \wedge i_1 \wedge i_2 \wedge i_3) \vee (i_0 \wedge i_1 \wedge i_2 \wedge i_4) \vee (i_0 \wedge i_1 \wedge i_3 \wedge i_4) \vee (i_0 \wedge i_2 \wedge i_3 \wedge i_4) \vee (i_1 \wedge i_2 \wedge i_3 \wedge i_4)$

The algebraic normal form (ANF), also called Zhegalkin polynomial form (Crama and Hammer, 2010; Zhegalkin, 1927), is of particular note. This form expresses the function as the exclusive-or of conjunctions (i.e., using $\oplus$ and $\wedge$ only), and is useful for examining some of the properties of functions. An ANF function makes some properties of a function relatively simple to obtain: for example, the *degree* of a function (also called the *nonlinear order*) is simply the maximum number of variables in any conjunction. It is trivial to see that the degrees of choice, majority, and parity are 2, 2, and 1 respectively. The ANF of addition gives degrees of 1, 2, and 4 for $s_{0..2}$ respectively.

The basic properties of note are as follows:

linearity Both confusion and diffusion are achieved through the use of *nonlinear* functions (Pieprzyk and Finkelstein, 1988). A minimal *linear* function is a function in which the output always depends on all of the inputs. By contrast, a minimal nonlinear function is a function in which the output does not *always* depend on all of the inputs. For example, consider the functions $f(a, b, c) = (a \wedge b) \vee c$ and $g(a, b, c) = a \oplus b \oplus c$ which have the following truth tables:

a	b	c	$f(a,b,c)$	$g(a,b,c)$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Note that when c is 1, the other inputs to f are irrelevant: the output will be 1. However, when c is 0, the other inputs to f determine what the output value will be. It is clear that $f(a, b, c)$ is a nonlinear function since its output does not *always* depend on all the inputs. By contrast, the output of g always depends on each input and g is therefore a linear function. Given an output of 0, it is possible to say that an even number of the variables a, b, c are 1-valued in the case of g; however, a similar deduction cannot be made in the case of f. Repeated application of different nonlinear functions, using different inputs, increases the amount of confusion (O'Connor and Klapper, 1994).

The parity and s_0 functions are linear, but the choice, majority, and $s_{1..2}$ functions are nonlinear. In fact, all boolean functions which have a degree of at most 1 are linear. Addition *in toto* is therefore a nonlinear operation due to the degrees of s_1 and s_2. Leurent (2010) makes the point that in "any non-linear function has an absorption property for at least one variable"; this is essentially a restatement of the linear/nonlinear definition, but is nevertheless an interesting way in which to regard the difference.

A related metric is *nonlinearity*, which is the number of 1-outputs of a function which, if changed to 0, would cause the function to be linear. This gives an indication of how much distance there is between a nonlinear function and a linear function. Choice and majority have a nonlinearity of 2; s_1 has a nonlinearity of 12, and s_2 has a nonlinearity of 6.

affine An affine boolean function is a function which has a degree of 1. All affine functions are linear, but may contain a constant term. An f-function which refers to constant values — for example, those in the first rounds of the SHA-1 compression function — may be affine, and s_0 is affine in SHA-1 due to the use of the k constant.

balance A boolean function is *balanced* if half of the possible inputs produce the output 1, and the other half produce the output 0. All of the f-functions are balanced. Addition is not balanced (Staffelbach and Meier, 1991), but a rotation done during splifflng attempts to compensate for this. More specifically, s_0 is balanced, but an examination of the Hamming weight of s_1 and s_2 shows increasing disbalance ($\frac{20}{32}$ and $\frac{6}{32}$ bits set respectively).

non-degeneracy A boolean function is *degenerate* if it contains variables which cannot affect the output (Dubuc, 2001). For example, the function $f(a, b, c) = a \vee b$ is degenerate: it ignores the c input entirely. It is important to make the distinction that a *nonlinear* function is not necessarily a *degenerate* function: the former disregards certain inputs based on the value of other inputs, and the latter always disregards certain inputs. All of the component functions used in the calculation of the SHA-1 hash are non-degenerate.

correlation immunity The *correlation immunity* of a boolean function is the degree to which the output cannot be correlated with any particular set of inputs. A function f that is mth-order correlation-immune is one in which there is no significant correlation between any boolean function comprised of $\geq m$ of f's inputs and f's outputs. Siegenthaler (1984) showed that a balanced non-degenerate function with n inputs and degree d satisfies the equation $m + d \leq n - 1$. The balanced, non-degenerate choice and majority are therefore 0th-order correlation-immune; parity is 1st-order correlation-immune; and s_0 is 3rd-order correlation-immune.

Neither s_1 nor s_2 are balanced, so Siegenthaler's inequality does not apply. However, Xiao and Massey (1988) provides a way to nevertheless find their correlation-immunity. s_1 is 1st-order correlation-immune and s_2 is 0th-order correlation-immune. A balanced nth-order correlation-immune is said to be nth-order *resilient*.

autocorrelation The *autocorrelation* of a boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ is defined as (Preneel, 1993)

$$\Delta_f(j) = \sum_{i \in \{0,1\}^n} (-1)^{f(i) \oplus f(i \oplus j)}$$

This measures how often the input and output disagree if changes to the input are made. If, for all values, $f(i) = f(i \oplus j)$, then $\Delta_f(j) = 2^n$; and, similarly, if $f(i) \neq f(i \oplus j)$ for all values then $\Delta_f(j) = -(2^n)$. The magnitude of the autocorrelation value shows how sensitive particular rows are to input bits being flipped. The j vector ranges from $0..2^n$–1, and the autocorrelation values therefore form a spectrum. The spectra for SHA-1 component functions are as follows.

Choice	8, 0, 0, -8, 0, 0, 0, 0
Majority	8, 0, 0, 0, 0, 0, 0, -8
Parity	8, -8, -8, 8, -8, 8, 8, -8
s_0	32, -32, -32, 32, -32, 32, 32, -32, -32, 32, 32, -32, 32, -32, -32, 32, -32, 32, 32, -32, 32, -32, -32, 32, 32, -32, -32, 32, -32, 32, 32, -32
s_1	32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 32
s_2	32, 16, 16, 16, 16, 16, 16, 8, 16, 16, 16, 8, 16, 8, 8, 8, 16, 16, 16, 8, 16, 8, 8, 8, 16, 8, 8, 8, 8, 8, 8, 8

The maximum absolute value in the spectrum is called the *absolute indicator*. This can be given as a figure relative to 2^n, where n is the size of the input vector, to provide a better picture of the autocorrelation property.

bent A *bent* boolean function is one which is as nonlinear as it is possible for a boolean function to be (Rothaus, 1976). Balanced functions cannot be bent (Seberry and Zhang, 1993) and functions with an odd number of inputs cannot be bent (Braeken, 2006); therefore, none of the component functions are bent.

symmetry A *symmetric* boolean function is one which depends only on the Hamming weight of the input. Parity, majority, and all addition functions are symmetric. The only non-symmetric function is choice.

A summary table of properties is given as Table 4.7.

Table 4.7: Summary table of boolean function properties

	Parity	Majority	Choice	s_0	s_1	s_2
degree	1	2	2	1	2	4
nonlinearity	0	2	2	0	12	6
balanced?	✓ $(\frac{4}{8})$	✓ $(\frac{4}{8})$	✓ $(\frac{4}{8})$	✓ $(\frac{16}{32})$	✗ $(\frac{20}{32})$	✗ $(\frac{6}{32})$
degenerate?	✗	✗	✗	✗	✗	
correlation-immunity	1	0	0	3	1	0
absolute indicator	2^n	2^n	2^n	2^n	2^n	2^n
bent?	✗	✗	✗	✗	✗	✗
symmetric?	✓	✓	✗	✓	✓	✓

4.2.3 A better equation

A more comprehensive equation for a_r^i is

$$a_r^i = a_{r-1}^{i+5} + f(a_{r-2}^i, a_{r-3}^{i+30}, a_{r-4}^{i+30})_{r-1}^{i+5} + a_{r-5}^{i+30} + k_{r-1}^{i+5} + w_{r-1}^{i+5} + v0_r^i + v1_r^i$$

At the level of bits, because XOR is equivalent to addition over GF(2) this can be restated as

$$\begin{aligned} a_r^i &= a_{r-1}^{i+5} \oplus f(a_{r-2}^i, a_{r-3}^{i+30}, a_{r-4}^{i+30})_{r-1}^{i+5} \oplus a_{r-5}^{i+30} \oplus k_{r-1}^{i+5} \oplus w_{r-1}^{i+5} \oplus v0_r^i \oplus v1_r^i \\ &= a_{r-1}^{i+5} \oplus f_{r-1}^{i+5} \oplus a_{r-5}^{i+30} \oplus k_{r-1}^{i+5} \oplus w_{r-1}^{i+5} \oplus v0_r^i \oplus v1_r^i \end{aligned} \tag{4.7}$$

Expanding this equation gives an equation which is fully-specified in terms of data-word bits alone. The equation for a^i_r is recursive, and known initial values provide the “base case” values. In the case of single-block SHA-1, these are:

$$\begin{aligned}
f_1 &= \text{0xfbfbfefe} \\
a_0 &= \text{0xe8a4602c} \\
f_0 &= \text{0x98badcfe} \\
a_{-1} &= \text{0xf9b5713d} \\
a_{-2} &= \text{0x5d6e7f4c} \\
a_{-3} &= \text{0x192a3b08} \\
a_{-4} &= \text{0xe970f861}
\end{aligned}$$

Of these, the a_0, f_0 and f_1 values are the only ones calculated from the initialization values; the rest are straightforward rotations of the initialization values.

4.2.4 Inputs and Outputs

It is useful to diagram the dynamic inputs and outputs of various functions that have been discussed, in order to understand their dependencies. The k inputs have been left out because they are static: no function must “wait” for them to be calculated. Similarly, the w inputs may be calculated independently, and can therefore not be dependencies of any other function.

At the highest level we find equation 4.7,

$$\xrightarrow[\textit{inputs}]{a^{i+5}_{r-1}, f^{i+5}_{r-1}, a^{i+30}_{r-5}, v0^i_r, v1^i_r} \boxed{\|} \xrightarrow[\textit{output}]{a^i_r}$$

The dependencies for f are

$$\xrightarrow[\textit{inputs}]{a^{i+27}_{r-1}, a^{i+25}_{r-2}, a^{i+25}_{r-3}} \boxed{\|} \xrightarrow[\textit{output}]{f^i_r}$$

The $v0$ function has the dependencies

$$\xrightarrow[\textit{inputs}]{a^{i+6}_{r-1}, f^{i+6}_{r-1}, a^{i+31}_{r-5}, v0^{i+1}_r, v1^{i+1}_r} \boxed{\|} \xrightarrow[\textit{output}]{v0^i_r}$$

Similarly, the $v1$ function's dependencies are

$$\xrightarrow[\textit{inputs}]{a_{r-1}^{i+7}, f_{r-1}^{i+7}, a_{r-5}^{i}, v0_r^{i+2}, v1_r^{i+2}} \boxed{} \xrightarrow[\textit{output}]{v1_r^i}$$

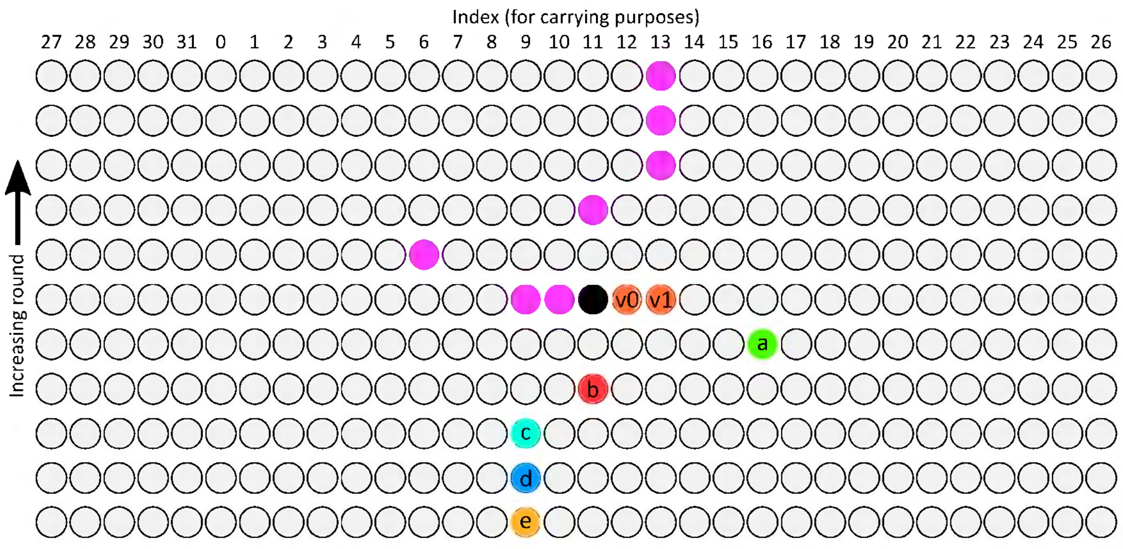

Figure 4.1: a-dependencies

The dependencies of a single a value are shown in Figure 4.1, which visually illustrates the a values for 12 rounds of the SHA-1 compression function. The bit under examination is shown in black in the center of this diagram, and the values that it is dependent on are shown in different colors and labelled using the traditional variable names; f has been expanded into (b, c, d), and the carries are $v0$ and $v1$. In turn, the bits that depend on the central black a value are shaded in magenta.

4.3 Summary and non-binary formulation

The equation for each a_r^i-value can be differentiated by the following factors:

1. whether the k_r^i value is 0 or 1;
2. whether the bit-position i falls into the category:
   (a) $i = 26$, in which case there is no $v0$ or $v1$ term;
   (b) $i = 25$, in which case there is no $v1$ term;
   (c) $i = 24$, in which case there is both a $v0$ and $v1$ term

3. whether the k_r^{i+1} value is 0 or 1, which affects the equation of $v0$;
4. whether the k_r^{i+2} value is 0 or 1, which affects the equation of $v1$;
5. whether the interval the round r falls into the category:
   (a) $r = [0..3]$, in which case the f-value is constant;
   (b) $r = 4$, in which case a single argument of the f-function is constant, causing one of the two cases to be true:
      i. the constant $a_0^{i+30} = 1$, in which case $f(b, c) = \neg b \vee c$
      ii. the constant $a_0^{i+30} = 0$, in which case $f(b, c) = b \wedge c$
   (c) $r = [5..19]$, in which case the f function is choice: $f(b, c, d) = (b \wedge c) \vee (\neg b \wedge c)$;
   (d) $r = [40..59]$, in which case the f function is majority: $f(b, c, d) = (b \wedge c) \vee (b \wedge d) \vee (c \wedge d)$;
   (e) $r = [20..39, 60..80]$, in which case the f function is parity: $f(b, c, d) = b \oplus c \oplus d$

The calculation of each individual a_i^r value can be represented using six variables, assuming that f is expanded and w is not expanded: a, b, c, d, e, and w. If calculation proceeds from $i = 26$ and continues onward, wrapping around towards $i = 27$, then $v0$ and $v1$ can be represented in terms of previously-defined a variables.

Thus far, an entirely binary formulation of the hashing process has been used. This is not necessary: if a non-binary perspective is more tractable, then such a perspective should be used. Consider the following non-binary formulation which meets all of the above conditions:

$$
\begin{aligned}
q_r^i &= a_{r-1}^{i+5} + f_{r-1}^{i+5} + a_{r-5}^{i+30} + w_{r-1}^{i+5} + v_r^i + k_{r-1}^{i+5} \\
a_r^i &= \begin{cases} 0 & \text{iff } q_r^i \in \{0,2,4,6,8\} \\ 1 & \text{otherwise} \end{cases} \\
v_r^i &= \begin{cases} 0 & \text{when } i = 26 \text{ or } q_r^{i+1} \in \{0,1\} \\ 1 & \text{when } q_r^{i+1} \in \{2,3\} \\ 2 & \text{when } q_r^{i+1} \in \{4,5\} \\ 3 & \text{when } q_r^{i+1} \in \{6,7\} \\ 4 & \text{when } q_r^{i+1} \in \{8,9\} \end{cases} \\
f_r^i &= \begin{cases} (a_{r-1}^{i+27} \wedge a_{r-2}^{i+25}) \oplus (a_{r-1}^{i+27} \wedge a_{r-3}^{i+25}) \oplus a_{r-3}^{i+25} & \text{when } 0 \le r \le 19 \\ (a_{r-1}^{i+27} \wedge a_{r-2}^{i+25}) \oplus (a_{r-1}^{i+27} \wedge a_{r-3}^{i+25}) \oplus (a_{r-2}^{i+25} \wedge a_{r-3}^{i+25}) & \text{when } 40 \le r \le 59 \\ a_{r-1}^{i+27} \oplus a_{r-2}^{i+25} \oplus a_{r-3}^{i+25} & \text{otherwise} \end{cases} \\
k_r^i &= \begin{cases} \text{0x5a827999}^i & \text{when } 0 \le r \le 19 \\ \text{0x6ed9eba1}^i & \text{when } 20 \le r \le 39 \\ \text{0x8f1bbcdc}^i & \text{when } 40 \le r \le 59 \\ \text{0xca62c1d6}^i & \text{when } 60 \le r \le 79 \end{cases} \\
w_r^i &= \begin{cases} w_{r-3}^{i+1} \oplus w_{r-8}^{i+1} \oplus w_{r-14}^{i+1} \oplus w_{r-16}^{i+1} & \text{when } r \ge 16 \\ w_r^i & \text{otherwise} \end{cases}
\end{aligned}
\tag{4.8}
$$

The traditional algorithm for calculating SHA-1 (NIST, 1995) uses addition; the above relations include v_r^i and q_r^i terms instead, and are semantically equivalent. The hash output is, as in the binary formulation, $a_{75..80}$.

4.4 Preimage equations

Assume that a "full" SHA-1 hash value, including Davies-Meyer construction, is generated from ω bits of data, $\omega \le 447$. The original $a..e$ values (which correspond to $a_{80}..a_{76}$ respectively) can be obtained via subtraction.

$$\begin{aligned} a &= x_0 - h_0 \\ b &= x_1 - h_1 \\ c &= x_2 - h_2 \\ d &= x_3 - h_3 \\ e &= x_4 - h_4 \end{aligned}$$

Appropriate rotations result in concrete $a_{76..80}$ values.

$$\begin{aligned} a_{80} &= a \\ a_{79} &= b \\ a_{78} &= c \hookrightarrow 30 \\ a_{77} &= d \hookrightarrow 30 \\ a_{76} &= e \hookrightarrow 30 \end{aligned}$$

At this point, it is only necessary to consider bits; the boundaries between words may safely be ignored. Therefore, for ease of notation, let u be the concatenated $a_{80..76}$ values, regarded as a single 160-bit value with u^0 being the MSB (i.e. $u^0 = a_{80}^0$). Each bit of u is associated with a corresponding equation a_r^i. Let $C(n)$ be the corresponding equation for u^n, and let $C'(n)$ be the negation of $C(n)$. These equations may be combined into a single equation F:

$$F = 0 = \bigvee_{n=0}^{160} \begin{cases} C'(n) & \text{if } u^n = 1 \\ C(n) & \text{if } u^n = 0 \end{cases} \tag{4.9}$$

or, equivalently,

$$F = 1 = \bigwedge_{n=0}^{160} \begin{cases} C(n) & \text{if } u^n = 1 \\ C'(n) & \text{if } u^n = 0 \end{cases} \tag{4.10}$$

4.5 Summary

This chapter has comprehensively analysed the SHA-1 compression function. Analysis of message expansion has led to the bitpattern method of calculation, as well as a comprehensive understanding of which bits are used for each expansion-word and how bits interact with each other. Common sub-sequences were analysed and an algorithm was proposed to efficiently enumerate valid single-block SHA-1 data-words that result in a particular expansion-word.

Spliffling, and the component functions used during spliffling, was likewise subjected to comprehensive analysis. Table 4.7 summarises the salient properties for each function. The analysis covered the constants that occur during single-block spliffling as well as the inputs and outputs of component functions. An alternative non-binary formulation was proposed, followed by recursive equations that represent a SHA-1 preimage. For completeness, a way to obtain the output of the compression function from the single-block SHA-1 hash function was also covered in this chapter.

Chapter 5

Phrasing the question

This chapter uses results from the fields of constraint satisfaction and complexity theory to quantify the theoretical difficulty of finding a preimage. After this, the practical difficulty is then addressed by some statistical analyses of SHA-1. The chapter and part closes with some reflections on how best to approach the problem, given the explorations thus far.

Each of the operations used in the SHA-1 algorithm is well-understood and has well-defined behaviour that any computer scientist can understand; yet by the end of the compression function, there is no observable correlation between input and the output. The SHA-1 compression function has been subjected to continuous attention for at least two decades, and it can be assumed that any trivial issues which would prevent it from being used as a one-way function would likely have been uncovered in this time. This also means that there is a wealth of literature to draw on while studying the compression function, unlike the universal one-way functions — which are "one-way functions" if and only if one-way functions exist — described by Levin (2003) or Kojevnikov and Nikolenko (2008).

The preimage-resistance of the SHA-1 compression function has not been proven, though it has been demonstrated over the years. There is no "intractable" problem that lies behind SHA-1's preimage resistance; it is not based on the inherent difficulty of finding an efficient algorithm to solve any particular mathematical problem. By contrast, the security of the RSA public-key cryptosystem (Rivest, Shamir, and Adleman, 1983) lies in the difficulty of efficiently factorising large prime numbers. To the best of the researcher's knowledge, there has been no work which has focused specifically on the question of why the SHA-1 compression function is preimage-resistant or the degree to which it is preimage-resistant.

Message-expansion and spliffling have been covered in some detail, and it is now possible to undertake a more detailed discussion of the difficulty of finding preimages. Recall that if $H(a) = c$, an input b such that $H(b) = c$ is called a second-preimage of c if $a \neq b$ and a preimage if $a = b$. Just as H maps a to c, so the SHA-1 compression function maps inputs to outputs. Any inputs which satisfy either Equations 4.9 or 4.10, which expressed the entirety of the compression function, would be a preimage of the hash.

The problem of finding such inputs is similar to the boolean satisfiability problem, expressed by Knuth (2011, p. 55) as finding "an algorithm that inputs a Boolean formula of length N and tests it for satisfiability, always giving the correct answer after performing at most $N^{O(1)}$ steps". The boolean satisfiability problem is known to be NP-complete. In the case of a particular hash, it is known that a solution does exist; however, the exact assignment of inputs (i.e. the preimage) is unknown.

The compression function maps inputs to outputs in a complex way. Any reduction in the complexity of the mapping makes it easier to link inputs and outputs. The boolean minimization problem is the problem of reducing a function to its simplest form — in other words, expressing it using the least number of terms. This problem is NP-hard (Buchfuhrer and Umans, 2008): an exact solution may be found via a Karnaugh map (Karnaugh, 1953) or the Quine-McCluskey algorithm (McCluskey, 1956), both of which are only suitable for small problems since the latter, which is more efficient, nevertheless has a time complexity of $\mathcal{O}(\frac{3^n}{n})$ (Nelson, 1995).

Although both boolean satisfiability and boolean minimization are difficult problems, they are not entirely intractable: appropriate data representations and complementary heuristic approaches can often provide "good-enough" solutions. Part III identifies potentially-useful representations and heuristics and attempts to apply them in the context of SHA-1. However, before beginning on such a path, it is worth analysing how difficult the SHA-1 preimage problem really is.

5.1 SHA-1 as a CSP(Γ)

Computational infeasibility can take many forms and the field of computational complexity exists to categorise and understand these forms. Computational complexity can be broken down into time complexity and space complexity, and a problem may be categorised as being in a particular complexity class. Time complexity is typically of more concern than space complexity, but the two are (at worst) linearly related since it can only take a limited amount of time to explore a limited amount of space.

The SHA-1 compression function can be viewed as a very structured kind of constraint satisfaction problem (CSP). CSPs are NP-hard in the general case, which means that solutions to a problem may be verified in polynomial time but obtained in nondeterministic polynomial time (Tack, 2009). However, not all CSPs are NP-hard, and adding additional constraints to a CSP — such as those shown in Equation 4.8 — typically reduces their difficulty since the interaction between constraints can make it easier to infer suitable values.

Jutla and Patthak (2005) model the internal state (i.e. the a variables) of SHA-1 as a constraint satisfaction problem that is similar to, but not equivalent to, SHA-1. Addition is modeled using only $\oplus$ and the majority function, and neither the choice function nore the linear message expansion are modeled. Such a model can be reduced to the Exact Cover problem, which is NP-hard (Karp, 1972):

> INPUT: family $\{S_j\}$ of subsets of a set $\{u_i, i = 1, 2, ..., t\}$
> PROPERTY: There is a subfamily $\{T_h\} \subseteq \{S_j\}$ such that the sets T_h are disjoint and $\cup T_h = \cup S_j = \{u_i, i = 1, 2, ..., t\}$.

This demonstrates that finding a solution to the NP-hard problem will likely result in significant advances towards finding a SHA-1 preimage. However, it does not demonstrate that SHA-1 itself is an NP-hard problem; such a demonstration would have to remodel an NP-hard problem in terms of SHA-1.

Jutla and Patthak (2005) also point out that establishing the difficulty of finding a SHA-1 preimage is equivalent to establishing the "one-wayness" of SHA-1 (Levin, 2003). To summarise the difficulty of this, they note that:

> ...actually proving the above one-way claim, even in an asymptotic sense is an extremely difficult problem. One approach could be to show that (in an asymptotic version of the above problem) the problem can be framed as a Polynomial Constraint Satisfaction Problem over $\mathbb{F}_2$ (where each polynomial has degree at most two), which is known to be NP-hard ([GJ79]). However, the notion of cryptographic one-wayness requires showing the problem to be average-case hard for NP. Unfortunately, all advances in this direction have

> been stymied by a theorem of Impagliazzo ([Imp95]) that any such result must be non-relativizing. Further, it has been shown ([FF76, BT05]) that under non-adaptive reductions this reduction is not possible unless the polynomial hierarchy collapses to the third level.

An *oracle* in complexity theory is a "black box" which answers a particular kind of difficult (or impossible) question in a single step. As such, problems can be placed into different complexity classes which are relative to a particular oracle. This technique can be used to prove that certain problems are more or less computationally-infeasible relative to certain other problems. However, such a *relativizing* proof does not necessarily extend to an implication in the real (*sans* oracle) world, and this approach is therefore not feasible at present.

Another way to look at the computational infeasibility of finding a SHA-1 preimage is to use Schaefer's theorem (Schaefer, 1978), a readable and modernised version of which is presented by Chen (2009). Schaefer's theorem permits us to evaluate the computational complexity of any two-element CSP in polynomial time. It is necessary to model SHA-1 in a way that makes it easy to analyze in terms of Schaefer's theorem, and to do this, it is necessary to understand those terms. A self-contained summary of the terminology and semantics of the field, derived from Chen (2009) and sufficient for understanding Schaefer's theorem, is as follows.

domain A domain is equivalent to a set. It specifies the legal values that a variable may take. *Example*: the boolean domain is $\{0, 1\}$.

relation A relation R over a domain D is a set of tuples of D. *Example*: the $\vee$ relation over the boolean domain is $\{(0, 1), (1, 0), (1, 1)\}$.

arity The arity of a relation is the arity of the tuples of the relation. An arity must necessarily be ≥ 1 and is here denoted by $|R|$. *Example*: the arity of the $\vee$ relation is 2.

constraint language A constraint language Γ is a set of relations — in other words, a set of sets-of-tuples. *Example*: If a constraint language contains the $\oplus$ and majority relations, then it is defined as

$$\begin{aligned}
\oplus &= \{(0,1),(1,0)\} \\
Maj &= \{(1,1,0),(0,1,1),(1,0,1),(1,1,1)\} \\
\Gamma &= \{\oplus, Maj\} \\
&= \{\{(0,1),(1,0)\},\{\{(1,1,0),(0,1,1),(1,0,1),(1,1,1)\}\}
\end{aligned}$$

variable This is an input to the CSP which may take on any value from the CSP domain.

constraint A constraint over Γ is an expression $R(v_1, ..., v_{|R|})$ where v_i is a variable. *Example*: $\oplus(x, y) = (\neg x \land y) \lor (x \land \neg y)$, which is simply a restatement of the $\oplus$ relation already described.

mapping (sometimes called **assignment**) A mapping is a function f which assigns values to variables in an attempt to satisfy a constraint.

constraint satisfaction problem CSP(Γ) is a decision problem that attempts to determine whether, given a constraint language Γ, a set of variables V, and a set of constraints C, a mapping $f : V \to D$ which satisfies all constraints exists.

polymorphism Consider a relation R and a function $f : D^n \to D$. For any n $|R|$-tuples $\in R$, apply f coordinate-wise. A polymorphism is defined as any f for which, irrespective of the tuples chosen, the result is in R. Richerby (2016) illustrates this as follows:

$$\begin{array}{ccccc}
(t_{1,1} & t_{1,2} & \cdots & t_{1,k}) & \in R \\
(t_{2,1} & t_{2,2} & \cdots & t_{2,k}) & \in R \\
\vdots & \vdots & \ddots & \vdots & \\
(t_{m,1} & t_{m,2} & \cdots & t_{m,k}) & \in R \\
\hline
(f(t_{1,1},\ldots,t_{m,1}) & f(t_{1,2},\ldots,t_{m,2}) & \cdots & f(t_{1,k},\ldots,t_{m,k})) & \in R
\end{array}$$

Example 5.1. *Is the binary operation $\oplus$ a polymorphism of the majority relation Maj?* If the majority tuples are arranged into a grid G,

1	1	0
0	1	1
1	0	1
1	1	1

A particular zero-indexed (column, row) entry is denoted $G_{col,row}$. There is at least one combination $((G_{0,0} \oplus G_{2,1}, G_{1,0} \oplus G_{2,1}, G_{2,0} \oplus G_{2,2}) = (0, 0, 1))$ where the result is not in *Maj*. Therefore, the binary operation $\oplus$ is not a polymorphism of *Maj*.

Example 5.2. *Is the binary operation $\vee$ a polymorphism of the majority relation Maj?* Using the same approach as in the previous example, it can be seen via exhaustive testing that it must be. However, a more intuitive approach could be used as well. It can be seen by scanning down the columns that each column has a single 0-value. Therefore, no matter which 0-value is used, that 0-value must be combined with a 1-value. Since $x \vee 1 = 1$, the result will always be $(1, 1, 1)$, which is in *Maj*. Therefore, the binary operation $\vee$ is a polymorphism of the majority relation *Maj*.

constraint language polymorphism An operation $f : D^n \to D$ is a polymorphism of constraint language Γ if f is a polymorphism of every relation in Γ. The set of all polymorphisms of Γ is denoted by $\mathrm{Pol}(\Gamma)$.

inverse constraint language polymorphism The set of all relations for which all operations in a set X are a polymorphism is denoted by $\mathrm{Inv}(X)$.

primitive positive definable (or **pp-definable**) Intuitively, if a relation can be used to simulate some other relation, then that relation is said to be pp-definable. The set of all relations which are pp-definable from a constraint language Γ is denoted by $\langle\Gamma\rangle$, and $\langle\Gamma\rangle = \mathrm{Inv}(\mathrm{Pol}(\Gamma))$.

reduction If all relations in a constraint language Γ are pp-definable in another constraint language Γ', then CSP(Γ) reduces to CSP(Γ').

Note that the constraint language defines which relations are *possible* in a CSP(Γ), and a constraint defines which relations *exist* between particular variables. The proofs for the above have been omitted, but are presented in Chen (2009). It is now possible to succinctly state Schaefer's theorem (Schaefer, 1978).

Schaefer's theorem CSP(Γ) where domain $D = \{0, 1\}$ is polynomial-time tractable if Γ has any of the following operations as a polymorphism:

1. Constant 0 (i.e. $f(x) = 0$)
2. Constant 1 (i.e. $f(x) = 1$)

3. Binary and (i.e. $f(x, y) = x \wedge y$)
4. Binary or (i.e. $f(x, y) = x \vee y$)
5. Ternary majority (i.e. $f(x, y, z) = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$)
6. Ternary parity (i.e. $f(x, y, z) = x \oplus y \oplus z$); this is also called the *minority* operation

Otherwise, CSP(Γ) is NP-complete.

The SHA-1 problem for the $\{0, 1\}$ domain can be modeled using the following relations.

- Choice : $\{(0, 0, 1), (0, 1, 1), (1, 1, 0), (1, 1, 1)\}$
- Majority : $\{(0, 1, 1), (1, 0, 1), (1, 1, 0), (1, 1, 1)\}$
- $\oplus$: $\{(0, 1), (1, 0)\}$

Addition can be represented by the majority and $\oplus$ functions for sum and carry bits respectively, following the approach of Jutla and Patthak (2005). All other $\oplus$-using functions, such as parity and message expansion, can be represented by extra variables and the $\oplus$ relation. Rotations change *which* variables are affected by which relations, and therefore affect constraints — but not relations themselves. The above relations should therefore be all that is needed to determine tractability. Each of the six cases mentioned by Schaefer's theorem will now be considered in turn.

Constant 0 / Constant 1 Neither of these is a polymorphism of $\oplus$ since neither $(0, 0)$ nor $(1, 1)$ are part of the $\oplus$ relation.

Binary and ($\wedge$) This is not a polymorphism of $\oplus$ since $(0, 0)$ is not a part of the $\oplus$ relation.

Binary or ($\vee$) Similarly to $\wedge$, this is not a polymorphism of $\oplus$ since $(1, 1)$ is not a part of the $\oplus$ relation.

Ternary majority This is a polymorphism of choice, majority, and $\oplus$.

Ternary parity This is not a polymorphism of choice since the tuples $(0, 0, 1)$, $(0, 1, 1)$, and $(1, 1, 0)$ give the result $(1, 0, 0)$, which is not a part of the choice relation.

Since majority is a polymorphism of all relations, it is known that SHA-1 is *not* in NP. Understanding why the majority case makes the problem tractable in polynomial time will provide clues as to how the problem may best be approached. Chen (2009) provides the following definition to illuminate the issue:

> *Definition* 4.1. Let $n \geq 0$. An instance of the CSP with variable set V has the *n-extension property* if, given any subset $W \subseteq V$ of size $|W| = n$ and a variable $v \in V$, any partial solution $f : W \to D$ can be extended to a partial solution $f' : W \cup \{v\} \to D$.

A "partial solution" is described as (Chen, 2009)

> Let ϕ be a set of constraints over variable set V. We say that $f : W \to D$, for W a subset of V, is a *partial solution* of ϕ if, for every constraint $R(v_1, ..., v_k) \in \phi$, there exists a tuple $(d_1, ..., d_k) \in R$ such that $f(v_i) = d_i$ for all $v_i \in W$.

Chen (2009) shows that every majority-polymorphism CSP(Γ) has a 2-extension property; and, furthermore, they prove that if a majority-polymorphism CSP(Γ) has the 2-extension property, then it also has a n-extension property for $n \geq 3$. Given this guarantee, a promising approach would therefore be to find a partial solution for at least two variables, after which the solution should be gradually extensible to the rest of the variables. Chen (2009) provides the outline of a generic approach to doing this as "Algorithm for Majority Polymorphism".

The work of Allender, Bauland, Immerman, Schnoor, and Vollmer (2009) makes it possible to refine this even further and discover exactly which complexity class within P the SHA-1 problem falls in. Since the majority function is a polymorphism of the SHA-1 constraint language, the applicable subclass of P is NL (nondeterministic logarithmic). NL shares the same relationship with L (logarithmic) that NP shares with P, and the question of whether NL = L is similarly an open problem. A decision problem in the L complexity class can be solved by a deterministic Turing Machine using $\mathcal{O}(\log n)$ space. Time complexity is a maximum of $\mathcal{O}(2^{\log n})$, since this is the time that is required to exhaustively explore the entire space.

A nondeterministic Turing Machine, unlike its deterministic counterpart, can simultaneously execute multiple actions. Savitch's theorem (Savitch, 1970) gives the relationship between deterministic and non-deterministic space as $\text{NSPACE}(f(n)) \subseteq \text{DSPACE}(f(n)^2)$, for any $f(n) \geq \log n$. Therefore, the exploration of the relevant space should not take more than $\mathcal{O}(\log^2 n)$ space, and a corresponding maximum of $\mathcal{O}(2^{\log^2 n})$ time.

5.2 Statistical analysis using the Strict Avalanche Criterion

Recall that a boolean n-bit hash function H is the transform $\mathbb{Z}_2^m \to \mathbb{Z}_2^n$. A cryptographic hash function attempts to obscure the relationship between the input and output of H, and the degree to which this is accomplished is directly related to the preimage resistance of the hash function. This implies that two similar inputs should have very different outputs.

The Strict Avalanche Criterion (SAC) (Webster and Tavares, 1986; Forré, 1990) formalizes this notion by measuring the amount of change introduced in the output by a small change in the input. It builds on the definition of *completeness*, which means that each bit of the output depends on all the bits of the input, in a way that is cryptographically relevant. Using the definition of H as above, an output $H(x) = y$ is obtained for an input x. The initial bit of x is now flipped, giving $H(x_0) = y_0$. This process is repeated for $x_{1..n}$, resulting in $y_{1..n}$. The SAC is met when the Hamming distance between y and $y_{0..n}$ is, on average, $\frac{n}{2}$.

Unfortunately, the Strict Avalanche Criterion was not originally defined as rigorously as it could have been. This has led to some confusion about what it is, and it is worth diverting some effort towards justifying the definition that this research uses. The original definition (Webster and Tavares, 1986) of the SAC is:

> Consider X and X_i, two n-bit, binary plaintext vectors, such that X and X_i differ only in bit i, $1 < i < n$. Let
>
> $$V_i = Y \oplus Y_i$$
>
> where $Y = f(X)$, $Y_i = f(X_i)$ and f is the cryptographic transformation, under consideration. If f is to meet the strict avalanche criterion, the probability that each bit in V_i is equal to 1 should be one half over the set of all possible plaintext vectors X and X_i. This should be true for all values of i.

Forré (1990) expresses this as:

> Let $\underline{x}$ and $\underline{x}_i$ denote two n-bit vectors, such that $\underline{x}$ and $\underline{x}_i$ differ only in bit i, $1 \leq i \leq n$. Z_2^n denotes the n-dimensional vector space over 0,1. The function $f(\underline{x}) = z, z \in \{0,1\}$ fulfills the SAC if and only if
>
> $$\sum_{\underline{x} \in Z_2^n} f(\underline{x}) \oplus f(\underline{x}_i) = 2^{n-1}, \text{ for all } i \text{ with } 1 \leq i \leq n.$$

Similarly, Lloyd (1990) understands the SAC as:

> Let $f : Z_2^n \mapsto Z_2^m$ be a cryptographic transformation. Then f satisfies the strict avalanche criterion if and only if
>
> $$\sum_{\underline{x} \in Z_2^n} f(\underline{x}) \oplus f(\underline{x} \oplus \underline{c}_i) = (2^{n-1}, ..., 2^{n-1}) \text{ for all } i,\ 1 \leq i \leq n.$$
>
> where $\oplus$ denotes bitwise exclusive or and c_i is a vector of length n with a 1 in the ith position and 0 elsewhere.

Other works (Preneel, 1993; Lloyd, 1993; Babbage, 1990; Kim, Matsumoto, and Imai, 1991) follow in the same vein. However, these definitions calculate the sum over *all* possible inputs as leading to the fulfillment of the SAC, which is contrary to the original definition. The original definition separates a *baseline* value from the *avalanche vectors*, and states that the SAC holds true when "the probability that each bit [in the avalanche vectors] is equal to 1 should be one half over the set of all possible plaintext vectors" (Webster and Tavares, 1986). Therefore, a better test of whether $f : \mathbb{Z}_2^n \mapsto \mathbb{Z}_2$ fulfills the SAC would use a universal quantifier,

$$\forall \underline{x} \in \mathbb{Z}_2^n, Pr(f(\underline{x}) = f(\underline{x}_i)) = 0.5$$
$$\text{for all } \underline{x}_i \text{ which differ from } \underline{x} \text{ in bit } i, 0 \leq i < n$$

A simple example clarifies the difference. Babbage (1990) uses Lloyd (1990)'s definition of the SAC and defines a SAC-compliant function:

Define $f : Z_2^n \mapsto Z_2$ by

$$\begin{cases} f(x_1, ..., x_n) = 0 & \text{if } x_1 = 0 \\ f(x_1, ..., x_n) = x_2 \oplus ... \oplus x_n & \text{if } x_1 = 1 \end{cases}$$

The simplest function of this nature is $f(\underline{x}) = x_0 \wedge x_1$. Then, taking $g(\underline{x}) = f(\underline{x}) \oplus f(\underline{x} \oplus 01)$ and $h(\underline{x}) = f(\underline{x}) \oplus f(\underline{x} \oplus 10)$,

$\underline{x}$	$f(\underline{x})$	$g(\underline{x})$	$h(\underline{x})$	$P(f(\underline{x}) = f(\underline{x}_i))$
00	0	0	0	1.0
01	0	0	1	0.5
10	0	1	0	0.5
11	1	1	1	1.0
	Sum:	2	2	

Note that the sum of each of the third and fourth columns is 2^{n-1}, as predicted, and that this function fulfills the summed definition of the SAC. However, the first and last rows do not fulfill the original definition of the SAC at all: the probability of change, given the baseline values 00 and 11, is 0.0 in each case. It is therefore more correct to regard the *row* probability as important. This understanding is also in accordance with the original text that defined the term. Under this definition, $x_0 \wedge x_1$ is not SAC-compliant.

It is worth noting that the original definition, as per Webster and Tavares (1986), is slightly ambiguous. They state that "the probability that each bit in V_i is equal to 1 should *be one half* over the set of all possible plaintext vectors X and X_i"; however, they also state that "to satisfy the strict avalanche criterion, every element must have a value *close to one half*" (emphasis mine). Under Lloyd's interpretation, the SAC is only satisfied when an element changes with a probability of precisely 0.5. This is an unnecessarily binary criterion, as it seems to be more useful (and more in line with the original definition) to understand how far a particular sample *diverges* from the SAC. Therefore, this work regards the SAC as a continuum but takes Lloyd's formulation as the definition of what it means to "meet" the SAC.

Preneel (1993) suggests a more general form of the SAC called the *propagation criterion* (PC), defined as

> Let f be a Boolean function of n variables. Then f satisfies the **propagation criterion of degree** k, $PC(k)$, $(1 \leq k \leq n)$, if $\hat{f}(\underline{x})$ changes with a probability of $1/2$ whenever i $(1 \leq i \leq k)$ bits of $\underline{x}$ are complemented.

The SAC is equivalent to $PC(1)$. The same work defines an *extended propagation criterion* which regards the SAC as a continuum. Much of the subsequent work (Seberry, Zhang, and Zheng, 1994; Zhang and Zheng, 1996; Carlet, 1998; Sung, Chee, and Park, 1999; Canteaut, Carlet, Charpin, and Fontaine, 2000; Gouget, 2004) in this area has more closely examined the relationship between PC and nonlinearity characteristics. Many of these extend the PC in interesting ways and examine ways of constructing functions which satisfy $PC(n)$, but experimental research that targets existing algorithms is scarce. Part of the reason for this may be that $PC(n \geq 2)$ and above is prohibitively expensive to calculate, even when using a statistical approach: there are $\binom{672}{n}$ combinations to consider. For SHA-1, this leads to $\approx 225,000$ combinations when $n = 2$, $\approx 50,000,000$ when $n = 3$, and so on.

Curiously, Bellare and Kohno (2004) define a measure called "balance" which, to distinguish it from the balance property of a boolean function, is called *h-balance* in this work. The measure comes close to the original definition of the SAC in some respects, but is not equivalent to it. Bellare and Kohno (2004) define a hash function as any $h : D \rightarrow R$ where $|D| > |R|$. For a bit of the range R_i, let $d_i = |\{x \in D : h(x) = R_i\}|$; then the h-balance $\mu(h)$ is

$$\mu(h) = \log_{|R|}\left(\frac{|D| \cdot |D|}{\sum_{i=1}^{|R|} d_i \cdot d_i}\right)$$

The ideal h-balance indicates that output bits are distributed uniformly, and has the value 1. A short example may make it easier to see the relationship between h-balance and the SAC.

Example 5.3. *Calculating h-balance.* Assume that the boolean function $h : \mathbb{Z}_2^3 \rightarrow \mathbb{Z}_2^2$ maps inputs as follows:

$$\begin{aligned}
000 &\rightarrow 00\\
001 &\rightarrow 01\\
010 &\rightarrow 00\\
011 &\rightarrow 01\\
100 &\rightarrow 11\\
101 &\rightarrow 10\\
110 &\rightarrow 11\\
111 &\rightarrow 10
\end{aligned}$$

Calculating the h-balance of this function,

$$\begin{aligned}
\mu(h) &= \log_2\left(\tfrac{8\cdot 8}{4\cdot 4+4\cdot 4}\right)\\
&= \log_2\left(\tfrac{64}{32}\right)\\
&= 1
\end{aligned}$$

Example 5.3 shows that the h-balance of the function indicates how well outputs are distributed with respect to input vectors. Note, however, that the outputs in the example are not distributed uniformly with respect to bits: the first bit of the output mirrors the first bit of the input. It can be seen from this example that whereas the h-balance measure is concerned with the uniform distribution of output bits with respect to input vectors, the SAC is instead concerned with the uniform distribution of output bits with respect to input bits.

Although there are proven theoretical ways to construct a function which satisfies the SAC (Kim *et al.*, 1991), there is no way (apart from exhaustive testing) to verify that an existing function satisfies the SAC. By contrast, useful cryptographic properties such as non-degeneracy (Dubuc, 2001) or bentness (Rothaus, 1976) are verifiable without having to resort to exhaustive testing. However, the SAC metric is no worse in this regard than the correlation immunity (Siegenthaler, 1984) and balance (Staffelbach and Meier, 1991) metrics which also require exhaustive testing.

5.2.1 Experimental design

It is computationally infeasible to exhaustively test the degree to which SHA-1 meets the SAC since the input space (2^{672}) is too large. However, it is possible to use a sampling approach instead, where representative samples are drawn from a population and inferences are made based on an analysis of those samples. This approach relies on each input being statistically independent of other inputs. Generating such input can be an extraordinarily difficult task (Chaitin, 2001); however, random.org[1] provides data which meets this requirement (Kenny, 2005; Foley, 2001). A source which may be more random, but which has undergone far less scrutiny, is HotBits[2]. Data for these experiments has therefore been obtained from random.org.

The inputs which make up the population should represent real-world usage, and the form of the input is therefore of concern. The inputs to the SHA-1 compression function are twofold: 16 32-bit words of input data and an initialization vector of 5 32-bit words, for a total of 21 32-bit words (or 672 bytes). The initial initialization vector is defined by the FIPS 180-1 specification, and the input data is padded and terminated such that the last two words processed by the algorithm encode the length of the input data. Subsequent initialization vectors are generated from the output of the previous application of the compression function. For any input which is larger than 1024 bytes, there is therefore at least one iteration of the compression function for which all 672 bytes are effectively "random" — if it is assumed that a previous iteration of the compression function can possibly result in the applicable initialization vector. To make this assumption, is sufficient to assert that there are no values which *cannot* be generated as intermediate intitialization vectors (given a pool of $\leq 2^{64}$ different bitstreams). Therefore, we can take independent 672-byte inputs as our population of concern.

The hypothesis to be tested is that SHA-1 meets the SAC. The desired margin of error is 1%, at a 99% confidence level. The required sample size is therefore determined by

$$n = \left(\frac{\mathrm{erf}^{-1}(0.99)}{0.01\sqrt{2}} \right)^2 = 16587$$

where erf^{-1} is the inverse error function

Given the 2^{672} input space, this seems to be a very small number; however, "it is the absolute size of the sample which determines accuracy, not the size relative to the population" (Freedman, Pisani, and Purves, 2007). Data collected during the experiment also indicates the degree to which SHA-1 does not meet the SAC, and the round at which the SAC comes into effect.

[1] `https://www.random.org`
[2] `https://www.fourmilab.ch/hotbits/`

Each of the 16587 inputs is passed through a custom implementation of the SHA-1 compression function. This implementation has not been validated by NIST's Cryptographic Algorithm Validation Program[3], but nevertheless passes all of the byte-oriented test vectors provided by NIST; in addition, source code for the compression function is available on request. When presented with a 672-byte input, the compression function outputs a list of 80 vectors, one for each round of the compression function. *Baseline* and *avalanche* vectors are generated for each input, and per-round compliance with the SAC is determined by these.

The primary question that this section seeks to answer is: to what degree do each of the output bits meet the SAC? To determine this, the per-input SAC value for each bit must be calculated, as described above. The geometric mean of the SAC values is representative of the central tendency. From the data that is generated to answer the primary question, two other questions may be fruitfully answered:

- **What is the distribution of SAC values per input?** The geometric mean provides a way to understand the degree to which an output bit meets the SAC, on average over a range of inputs. The distribution of SAC values quantifies how likely any particular input is to meet the SAC.
- **How quickly do the bits of the SHA-1 hash meet (or not meet) the SAC?**

For repeatability, it is disclosed that the data used to create inputs is the first $16587 \times 672 = 11,146,464$ bits generated by random.org from the 2nd to the 12th of January 2015. This data is available from `https://www.random.org/files/`.

5.2.2 Results

As shown by Figure 5.1, the SHA-1 hash diverges from the SAC by remarkably small amounts. The initial divergence is due entirely to the fact that the very last bits of a 672-bit input are found in rounds 15 and 16 and, when modified, have an exaggerated effect on subsequent rounds. This effect is largely due to the fact that the changes have not yet had time to diffuse through the rounds. Data which is most representative of the final hash output can therefore be seen in rounds ≥ 24.

[3] `http://csrc.nist.gov/groups/STM/cavp/\#03`

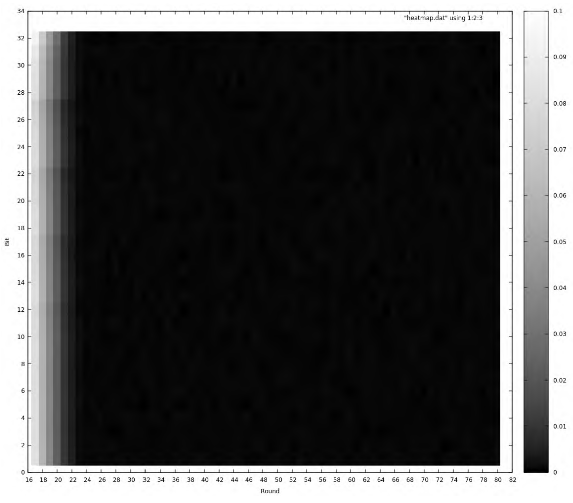

Figure 5.1: Divergence from SAC, round 17..80

If sufficient time is provided for diffusion, a different picture emerges. Figure 5.2 shows the absolute divergence from round 24 onwards. Although the heatmap looks noisier, the most important thing to note is that the maximum divergence from the "ideal" SAC value of 0.5 is only 0.0009, which is within the margin of error for this sample size.

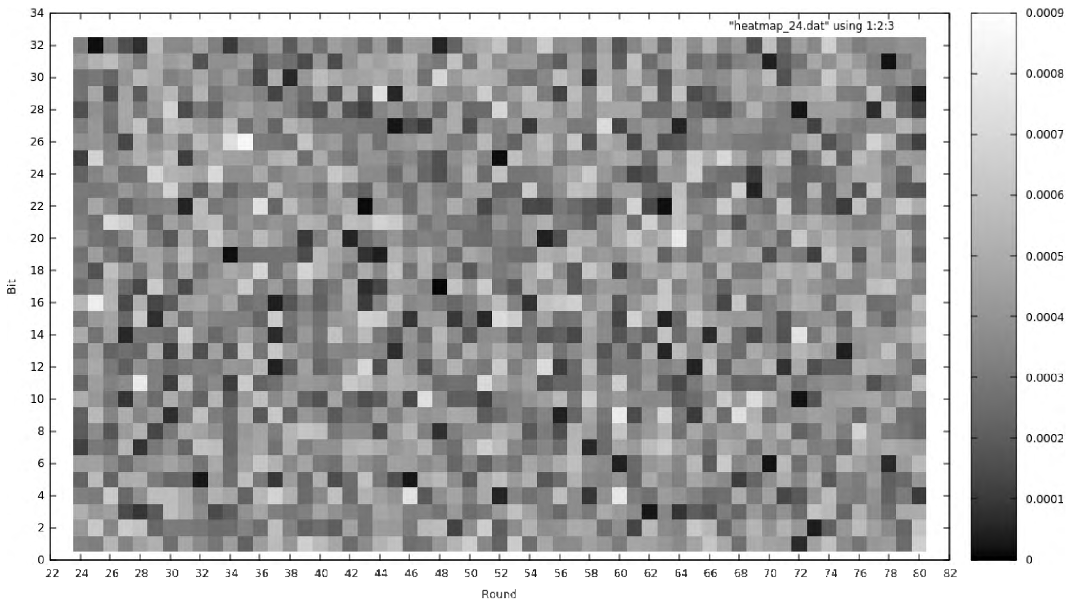

Figure 5.2: Divergence from SAC, rounds 24..80

A 5-figure statistical summary (minimum, lower quantile, median, upper quantile, and maximum) of deviation from the SAC is plotted as Figure 5.3. In this graph, (round, bit) tuples have been converted to single value bits using the function $bit(r, i) = (r - 1) \cdot 32 + (i - 1)$. This was done to better illustrate noteworthy points, and because there is no round-specific pattern in the data. The median value is 0.0 throughout, and the lower and upper quartiles demonstrate remarkable consistency across the rounds despite minima and maxima which fluctuate significantly. It is interesting to note that rounds 24..44 show the same pattern as rounds 60..80, which are the final rounds of the hash. The distribution of values appears to remain constant from round 24 all the way up to round 80.

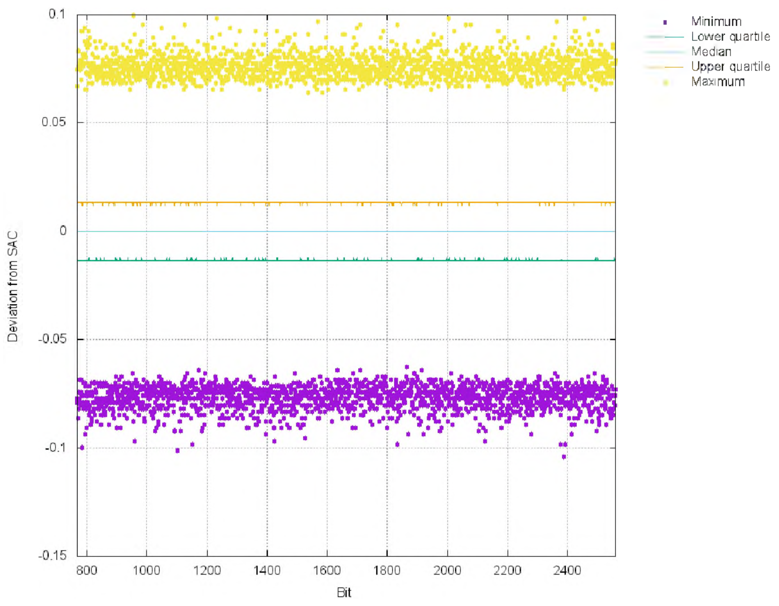

Figure 5.3: Summary statistics

The distribution of SAC values for rounds ≥ 24 is shown in Figure 5.4, and there are few surprises here. It has a median, mean, and mode of 0.5, and appears to be a normal distribution. To verify whether the distribution is, in fact, normal, a quantile-quantile plot was generated. A quantile-quantile plot overlays points from a data-set on top of the theoretically-predicted distribution; if the actual points lie along the theoretically-predicted line, then the data fits the specified distribution.

Three possible distributions were plotted (see Figure 5.5):

- Normal ($\sigma = 0.019285397$, $\mu = 0.5$), using the standard deviation and mean of the data where round ≥ 24.
- Log-normal ($\sigma = 0.059899039$, $\mu = 0.49855239$), estimated from the data.
- Weibull ($k = 9.6116811$, $\lambda = 0.52480750$), estimated from the data.

None of the distributions match the data exactly; in fact, the normal distribution is the worst fit, with log-normal and Weibull distributions being much closer fits. The distribution that the data conforms to is unknown.

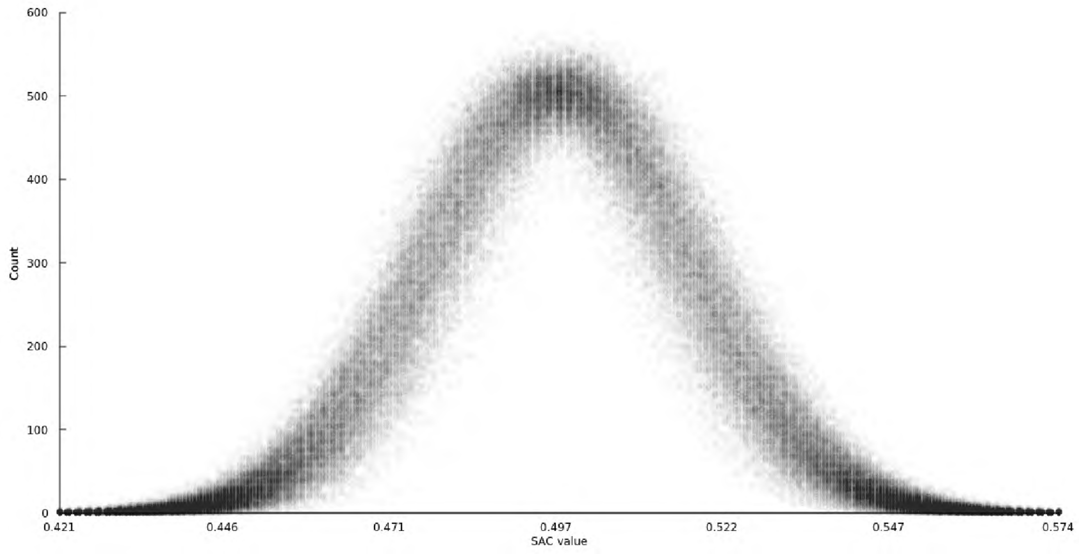

Figure 5.4: Distribution of SAC values

5.2.3 Quantifying the difficulty

As the experimental results show, each of the output bits meets the SAC by round 24, and it therefore takes only 8 rounds from the end of the input data for the SAC values to settle into a “stable” state. This stable state persists through all of the remaining rounds.

This work is concerned with preimage resistance: the computational infeasibility of finding an input that results in a particular output. The SAC results obtained from these experiments highlight the difficulty of obtaining a specific preimage since, from round 24 onwards, the SAC is either met or very closely approximated. This makes it extraordinarily difficult to determine which input bit could contribute to a particular output change, since the answer is “all of them”! The straightforward approach of attempting to find relationships between input bits and output bits is thus largely useless.

5.3 Summary

Theoretically, Schaefer’s theorem indicates that it would be possible to find a preimage in a reasonable amount of time, as long as a partial solution can be found first. The problem lies with the requirement for the partial solution to be known as a solution: values cannot simply be guessed since it would be impossible to extend such a solution in a way which is correct, given that the correctness of the original solution cannot be established.

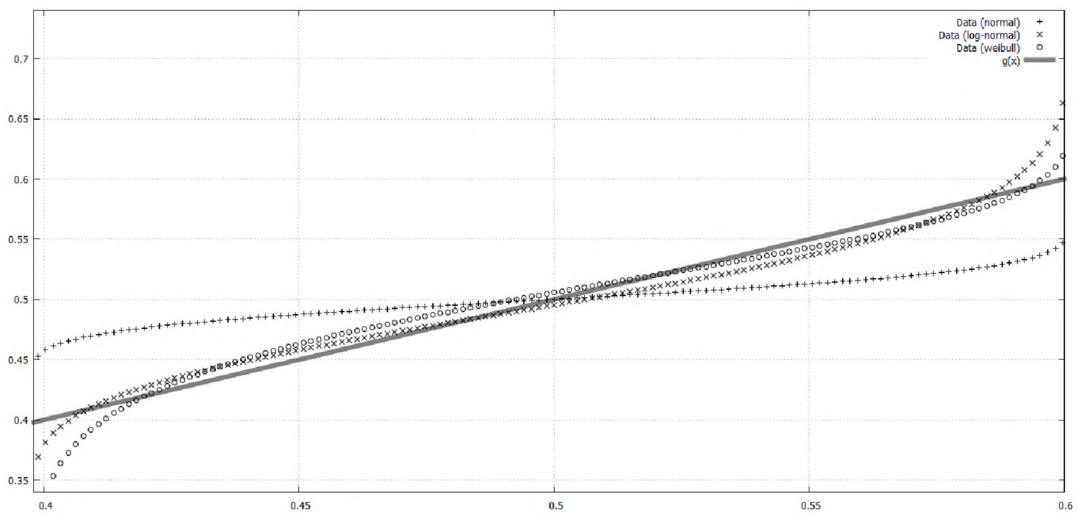

Figure 5.5: Quantile-quantile plot showing goodness of fit

Existing attempts, detailed in Chapter 3, have failed. The obstacles to meet-in-the-middle attacks (Knellwolf and Khovratovich, 2012; Espitau *et al.*, 2015) seem, at present, to be insurmountable — though it is hoped that future research will prove that statement to be incorrect. The use of GPGPU and similar technology (Grechnikov, 2010; Adinetz and Grechnikov, 2012) may be useful in finding collisions, but are very far from being practical for preimage purposes since the chance of success for finding a preimage is so much lower.

A different way will be attempted in the following part of the thesis: different ways of representing SHA-1 will be tried in the hope of finding one which will make it easier to find a partial solution which can then be extended towards a full solution.

Part III

Representations

Chapter 6

A framework for representations

This chapter sets out a framework for studying different representations of SHA-1. The word “representation”, in this context, denotes a data structure that encodes the SHA-1 algorithm. The chapter begins with a high-level description and justification of the framework that makes it easier to explore different representations. This is followed by an unsuitable representation that uses the framework. The unsuitable representation illustrates the fact that it is not useful to implement *any* representation; merely “fitting” the framework does not make a representation worthwhile. The illustration is followed by a discussion of “worthwhile” representations in the context of this research. An overview and brief justification of the chosen representations is then presented. Each chapter in the remainder of this part discusses a particular representation. The language of implementation is F# and the framework itself is open-sourced at `https://github.com/cynic/RepresentationFramework`.

6.1 Framework description

The SHA-1 algorithm is, at the core, nothing more than a very complicated boolean function that maps inputs to outputs. This means that it could be represented using any of the many ways of representing a boolean function — and, since boolean functions are fundamental to many fields, this is a very large set of possible representations. Correctness of the representation is paramount: if the representation does not faithfully encode the SHA-1 algorithm, then it is useless. The flexibility to add new representations was also very important since it was unknown whether the initial set of worthwhile representations would be comprehensive enough. It is also important to be able to increase and decrease the size of the representations, for two reasons: it became clear that many representations would be unable to handle 512 (or even 447) bits of input; and the ability to scale the problem up and down is useful for examining the behaviour of representations.

To address these issues, the developed framework represents the workings of the SHA-1 algorithm as the set $\{\wedge, \vee, \oplus, \neg\}$ of boolean operations following on from the exploration of Chapter 4. Data was represented as the abstract set $\{0, 1, w_0, ..., w_\omega\}$, where w_0 represented the first bit of the input, w_1 represented the second bit, and so on. Each representation had to provide a way to implement these abstract operations, and represent the data abstractions, in whatever way was most "natural" to that representation. The framework accepted a representation and used it to execute exactly the same sequence of abstract operations, in exactly the same order, irrespective of concrete form of the representation. Assuming that the framework represented the SHA-1 algorithm faithfully, each representation would necessarily do the same. Another benefit of this approach was that it allowed new representations to be added more easily than would otherwise have been the case.

If a representation could not handle larger numbers of bits, a smaller set of input variables was used, with the terminator and length being set as appropriate in the data-words. Since the abstractions for 0 and 1 were provided by the representation, this was reasonably simple to achieve.

Figure 6.1 shows the basic abstractions that are used. The purpose of the Constants and IRepresentation abstractions have been discussed above. An "input bit" is linked to a data representation of that input by the MakeVariable method of IRepresentation; the Evaluate method evaluates the value of a representation, given the supplied data-words, and returns a boolean result. The a, f, w, $v0$, and $v1$ bits are tracked throughout the calculation and assigned individual Designator tags; they are stored and retrieved by tag from a IStorage. Sub-calculations to generate each tracked value are implemented by a ISubCalculator, which utilizes a IWords for generating w values.

The Evaluate function was at the heart of the validation process. The "concrete" representation for each a_p^i was stored and, later on, passed back to the representation with a set of suitable randomly-initialized data-words. The value of a_p^i, given those data-words, was calculated using a known-good SHA-1 implementation that was validated against both the NIST test-vectors and the standard GNU SHA-1 implementation, *sha1sum*. Several different sets of data-words were used, and the result of Evaluate was compared against

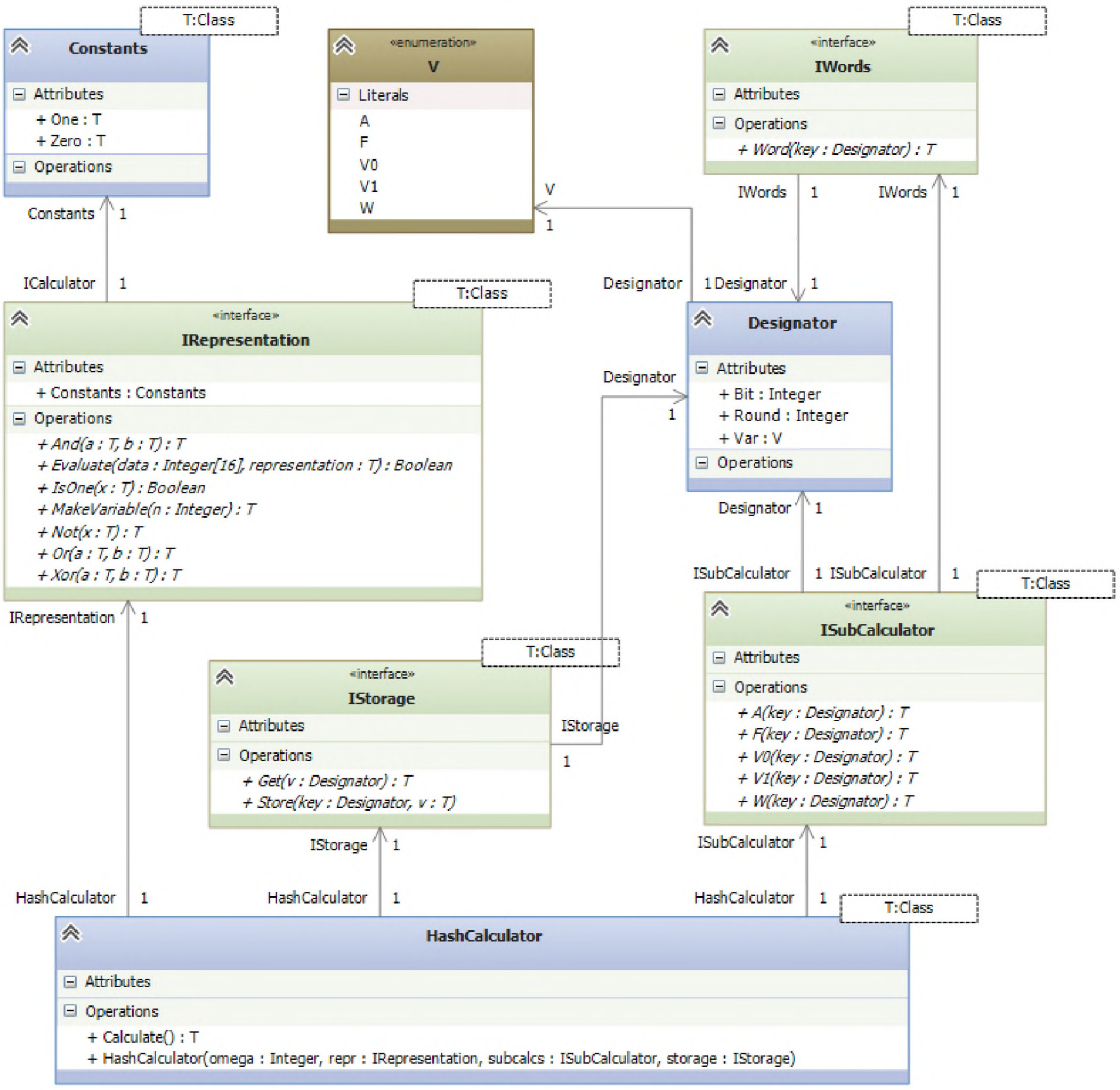

Figure 6.1: Representation framework: class diagram

the known value each time. Any deviation, for any bit, raised an error in the system. The validation was conveniently associated with the code through the use of a unit testing framework, which made it very easy to test every representation in an automated way whenever larger changes were made. The unit testing framework was also configured to verify the basic logical operation abstractions, and basic data abstractions, using truth tables.

The HashCalculator object brings all of these interfaces together, accepting a IRepresentation, ISubCalculator, and IStorage to calculate the necessary representations. The total size of the HashCalculator class is <40 lines; all significant functionality is performed by the inputs. Similar classes exist to calculate the hash in a top-down fashion; to generate a single value that represents a particular preimage, using Equation 4.10; and to perform different analyses and transformations.

Many different concrete implementations of the various interfaces were created over the course of the research. The "traditional" and "bitpattern" calculation of w-values, for example, are both implemented as IWords implementations. This flexibility made it trivial to swap in, test, and swap out different implementations and analyse their interactions. It also allowed different trade-offs to be made: for example, larger representations could be written to disk and smaller ones could be stored entirely in memory (or a database) by using suitable IStorage implementations.

6.2 Unsuitable representation: truth tables

It can be argued that the simplest and most direct way to represent any boolean function is via a truth table such as Table 6.1. A truth table explicitly lists the function value for all possible variable values, but requires 2^n bits to represent an n-variable function. A 32-variable function would therefore require 4GiB to be representable, and approximately a petabyte is needed to represent a 50-variable function.

Table 6.1: Truth table for $f(x_{0..3}) = (x_0 \wedge x_1) \vee (x_2 \vee x_3) \wedge \neg(x_2 \wedge \neg x_0)$

x_0	x_1	x_2	x_3	$f(x_{0..3})$
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

On the positive side, truth table representation is exceptionally simple to implement, requiring fewer then 100 lines in a modern garbage-collected language with bit-vector support. A variable w_r^i in this representation is a bit-vector of length 2^ω in which sets of $2^{r \cdot 32+i}$ bits are set at intervals of 2^ω – just as has been done in Table 6.1.

Boolean operations using truth table representations are embarrassingly parallel bit-operations on bit-vectors of size 2^ω, and consequently become much more expensive as ω increases. Each bit in the bit-vector represents the result of operations up to that point for the appropriate row in the table.

6.2.1 Collision detection

The truth table representation effectively calculates every possible hash value that could be obtained from an ω-bit string. If subsequent bits from different rows are equal, a collision involving the input bits (i.e. the row bit-vectors) has occurred. A simplistic Bloom-filter-inspired algorithm (Bloom, 1970) for finding such collisions is described as Algorithm 6.1.

The key insights of this algorithm are to only undertake an extended search if the first *filterSize* bits of the outputs are exactly the same, and to only record the results of an extended search if the near-collision found is better than the previous near-collision. The former feature greatly reduces the number of full checks that need to be performed: for $\omega = 22$, 524857 full checks (12.5% of a possible 4194304 checks) were performed.

Extending the algorithm to multiple cores (or multiple machines) involves a straightforward partition of the search-space. Write-contention is greatly limited by the filter. For $\omega = 22$, the actual number of writes to the shared *result* structure is 18, out of a search of 4194304 positions. Although the *fitness* value is also shared, the maximum value of *fitness* is 160 and it can therefore be represented as a 32-bit (or smaller) integer. It is worth noting that the value is only used during full checks, and most instruction sets support reasonably efficient atomic operations on such values.

For $\omega = 24$ bits, near-collisions matching between 85 and 122 (out of 160 bits) were found by a multi-core variant of Algorithm 6.1 after a runtime of approximately 8.5 minutes on a 4-core i5-class CPU. The best near-collisions for various input sizes are shown in Table 6.2. Note that these are near-collisions for the compression function *before* the Davies-Meyer step is applied. Applying the step does not add much to the runtime.

Algorithm 6.1 Near-collision detection using truth-table representation

```
Require:
    $2 \leq \omega \leq 447$
    $2 \leq filterSize$
Ensure:
    A list of near-collisions and the number of colliding bits
 1: function EXAMINEROUND($\omega, filterSize$)
 2:     $bv \leftarrow$ Array of $2^{filterSize}$ empty lists
 3:     $size \leftarrow 2^{\omega}$
 4:     $data \leftarrow$ Array of 160 truth-tables for $\omega$ bits
 5:     $best \leftarrow 0$
 6:     $result \leftarrow$ expandable list
 7:     function FULLCHECK($i, j$)
 8:         $fitness \leftarrow 0$
 9:         for $t \leftarrow 0..160$ do
10:             if $data_{t_i} = data_{t_j}$ then
11:                 $fitness \leftarrow fitness + 1$
12:         if $fitness > best$ then
13:             $result \leftarrow (i, j) :: result$
14:     end function
15:     for $row \leftarrow 0..size$ do
16:         $idx \leftarrow 0$
17:         for $j \leftarrow (27..27 + filterSize)$ do
18:             $idx_{j \bmod 32} \leftarrow data_{j_{row}}$
19:         $bv_{idx} \leftarrow row :: bv_{idx}$
20:         for $x$ in $bv_{idx}$ do
21:             if $x \neq row$ then FULLCHECK($x, row$)
22:     return $result$
23: end function
```

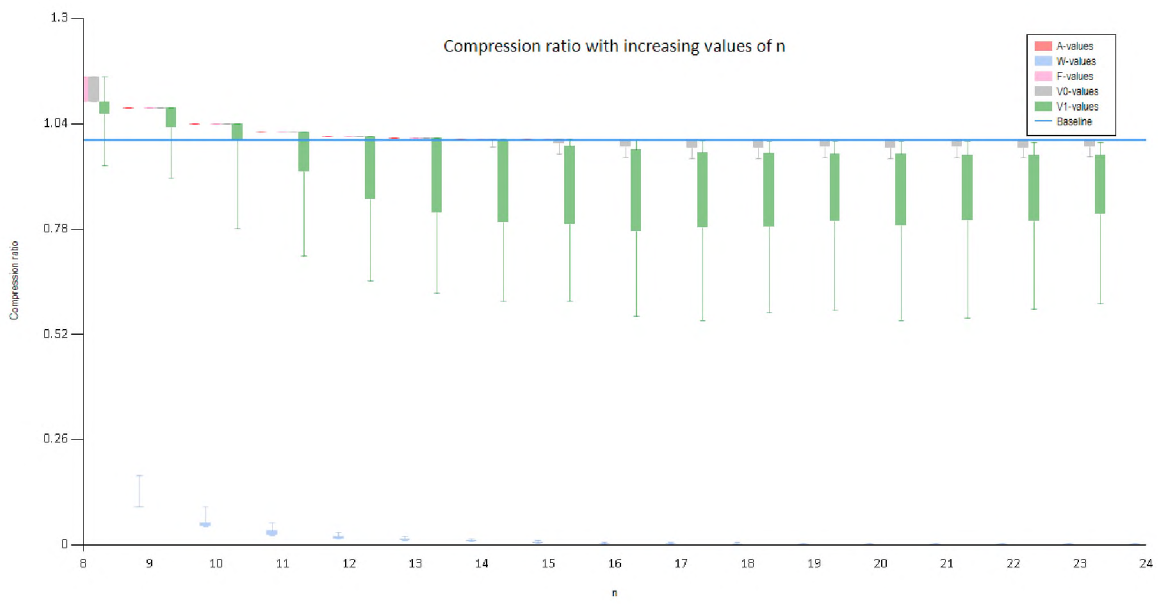

Figure 6.2: Compression ratio vs ω

Table 6.2: Near-collisions for 24-bit inputs

ω	Matching bits	Input A	Input B
12	93	0xed8	0x67f
14	98	0x5798	0x75d0
16	106	0xe329	0x9555
18	115	0x5dc2	0xa50bc
20	117	0xebd23	0x7816c
22	119	0x4d260	0x6cbd7
24	122	0xc9313b	0xe68786

This work is not concerned with collisions, but with preimages; the above, therefore, simply demonstrates one possible use of the truth-table representation that relates to SHA-1.

The effectiveness of the compression function is correlated with the incompressibility of this representation. The Shannon entropy (Shannon, 1948) quantifies the difficulty of predicting the next bit in a sequence, and hence the “randomness” of the sequence. The greater the Shannon entropy, the more difficult a sequence of bits is to compress. A hash function strives to obscure the relationship between input and output, and the output for a particular input should be effectively random — and, therefore, incompressible. Since all possible values of an n-bit hash are being computed for this representation, the incompressibility of data is an indication of the difficulty of finding a preimage for *any*

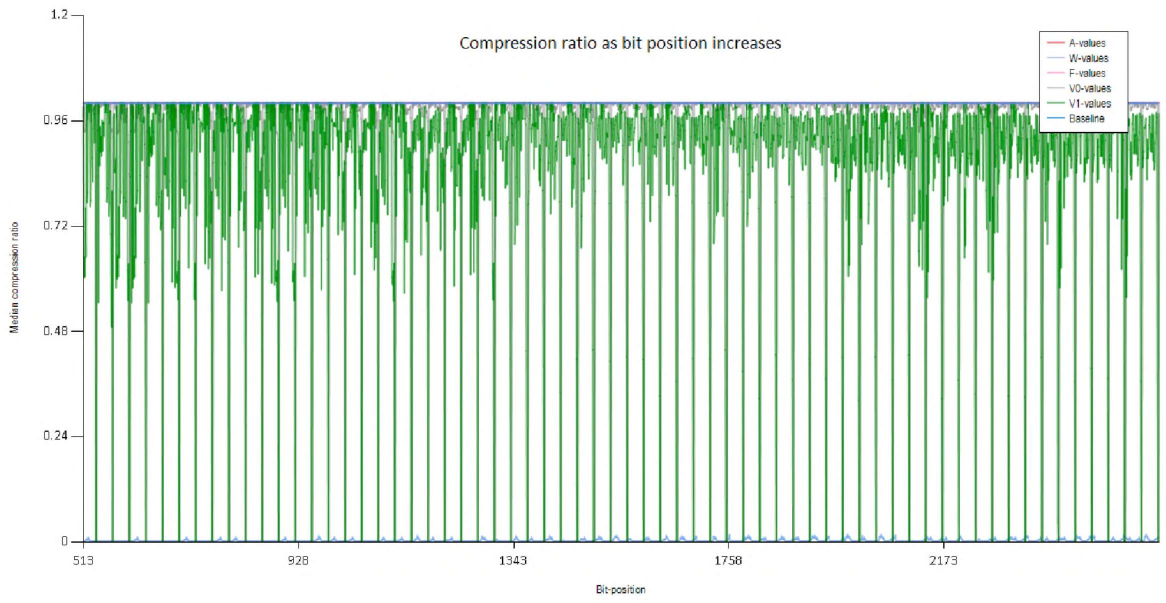

Figure 6.3: Compression ratio vs bit position

input.

The ideal compression ratio is as close to zero as possible, with 1.0 indicating that no compression was possible. Data was compressed using the DEFLATE algorithm (Deutsch, 1996). Attempting to compress data involves some overhead (such as a dictionary of symbols), and an increase in size of the compressed data is therefore possible. Figure 6.2 shows the average compression ratio for $8 \leq \omega \leq 24$. It can readily be seen that compression becomes much more difficult as ω increases, and effectively plateaus around $\omega > 14$ (i.e. a truth-table size of 16Kib).

Figure 6.3 shows the median compression ratio across $8 \leq \omega \leq 24$ as the bit-position increases. The graph is difficult to read, but illustrates the overall trend very well: except for the regular instances where $v1 = 0$ or $v0 = 0$, the data is largely incompressible. In fact, median values of a and f hover at, or just over, 1.0 throughout the rounds. This reinforces the results of Section 5.2, since a compression function which meets the SAC would tend to be incompressible. Figure 6.4 zooms into the last five rounds of the compression function to better show the incompressibility.

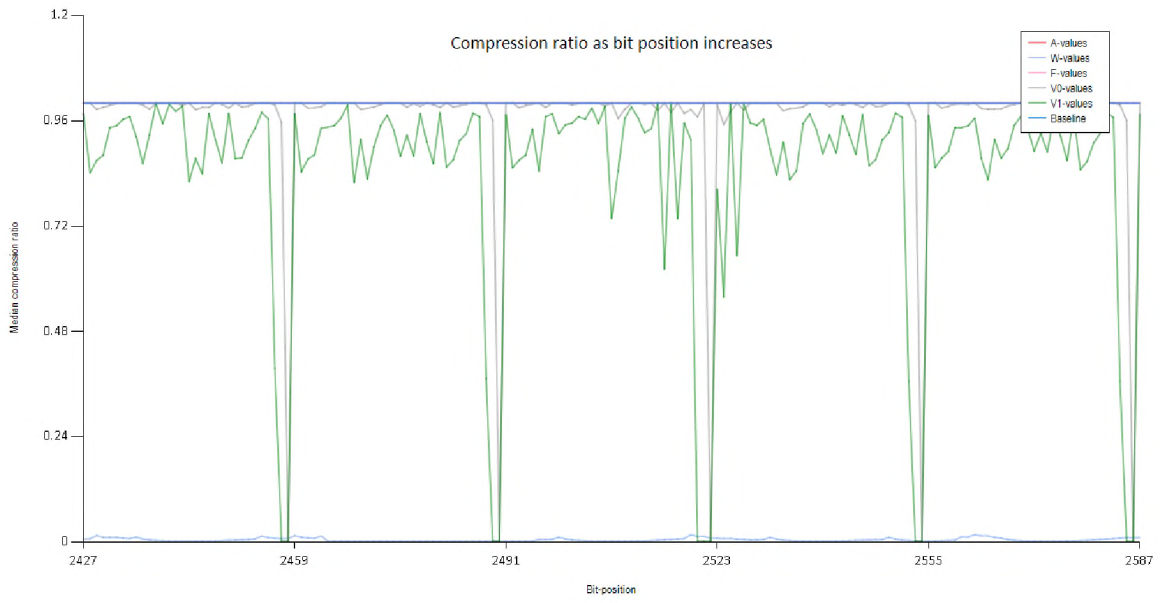

Figure 6.4: Compression ratio vs bit position (last five rounds)

6.3 Worthwhile representations

A representation which is worthwhile is one which is likely to reveal some information about SHA-1 preimages, or which may make it feasible to attack the SHA-1 preimage problem in some way. Not all representations are worthwhile; this will be demonstrated in the next section. Some characteristics of a worthwhile representation are given below.

- A representation must be able to represent the the entire SHA-1 calculation; it must therefore be able to represent a functionally complete set of boolean operators and the operands for those operators.

  The truth table representation is able to represent the entire SHA-1 calculation.

- A representation must be able to represent the links between bits. Some of the best preimage research to date (see Espitau *et al.* (2015), for example) does not represent all the links between rounds, but represents a "forward" and "backward" link and attempts to match them in order to find a preimage. However, the approach in this thesis is predicated on attempting to understand the connections between input and output; this is not possible if all links are not faithfully represented.

  The truth table representation is not able to represent the links between bits. Information about the operands is entirely lost and cannot be recovered by examination of the final output.

- It must be possible to analyse a representation. One possibility is to create lambda functions for each operation which "wrap" other functions, with a base-case function simply returning "true" or "false"; executing a particular function would execute sub-functions until a result is obtained. This sort of representation represents the links between rounds, but hides them within opaque lambda-function "boxes". It is therefore of limited use for preimage research.

  The truth table representation makes it impossible to analyse a representation because the links between bits are lost.

- The encoding of the algorithm should be separate from the encoding of an instance. Instance data can reveal interesting things about an algorithm — see, for example, the statistical analysis in Chapter 5. However, the focus of this thesis is on the algorithm itself and not on any particular instance. A representation that encodes an instance without encoding the algorithm is not worthwhile.

  The truth table representation represents all the possible encodings of ω bits. However, it does not encode the algorithm itself; only the results of the algorithm are represented.

In addition to the above points, it is not worthwhile to examine two or more representations which provide the same insight. The representations must therefore be carefully selected in an attempt to understand different aspects of the SHA-1 preimage problem. Five different representations which meet the above criteria have been selected for examination.

Conjunctive Normal Form (Chapter 7). This canonical representation is amenable to SAT-solving, which is a form of logical cryptanalysis.

Disjunctive Normal Form (Chapter 8). This canonical representation makes it exceptionally easy to identify preimages by inspection. However, it is also difficult to represent some functions in this form, and function minimization is necessary for larger inputs.

Reduced Ordered Binary Decision Diagrams (Chapter 9). This canonical representation is arc-consistent, which makes it very easy to see the relationships between variables. Furthermore, certain functions have an exceptionally small ROBDD representations, and many Binary Decision Diagram variants exist.

And-Inverter Graphs (Chapter 10). This non-canonical representation has stimulated great interest in recent years due to its scalability, flexibility, and ability to make local changes that have larger global effects.

Constraint Satisfaction Problem (Chapter 11). This is likely the most general of the representations, and is the only representation that is unsuitable for representation using the described framework. A language Γ (see Section 5.1) maps naturally to a set of constraints that a sophisticated solver can attempt to solve.

6.4 Other excluded representations

There are innumerable ways in which data can be modeled, structured, and represented, and the field of data structures is almost as old as the field of Computer Science itself. It is therefore true that this chapter, although it examines the representation of SHA-1 using very different structures, cannot be as comprehensive as a researcher might want it to be. Some of the data representations that were not discussed, but may be interesting to explore in a future work, are:

Integer representation Boole (1854) originally represented boolean logic using integers as a multivariate algebra over GF(2), and it is simple to convert any boolean formula into this algebra (Brown, 2011):

- $\neg x \mapsto 1 - x$
- $x \wedge y \mapsto x \cdot y$
- $x \vee y \mapsto x + y - x \cdot y$

The result is a multivariate polynomial can be manipulated by powerful mathematical software. Note, however, that the translations of $\wedge$ and $\vee$ both involve multiplication. Although there are fast and efficient ways to multiply polynomials, there are no known efficient ways to multiply *multivariate* polynomials, and it is especially difficult to multiply polynomials with a large number of different variables (Bodrato, 2007; van der Hoeven and Lecerf, 2013; Popescu and Garcia, 2016). If an algorithm is found to make this computationally feasible, then an integer representation would be worth investigating.

Propositional Directed Acyclic Graphs These represent the basic operations $\wedge$, $\vee$, and $\neg$ as nodes in a directed acyclic graph (Wachter and Haenni, 2006b). In general, the number of nodes required to represent a function as a PDAG is greater than the number of nodes required to represent a function as a ROBDD, and PDAGs do not have a canonical form (Wachter and Haenni, 2006a). Two very different graph-based representations (AIG and BDD), as well as one variant (ZBDD), will be considered in this work, and the literature on other graph-based representations does not indicate that additional graph-based representations would provide additional insight. Despite this indication, the only way to be sure would be to experiment with such representations, and this is therefore an additional avenue that could be investigated.

Algebraic Normal Form An equation in Algebraic Normal Form (ANF) uses only the $\{\oplus, \wedge\}$ operations and the constants 1 and 0. ANF, like CNF and DNF, is a canonical form. The mapping of the $\neg$ and $\vee$ operations to ANF is straightforward: $a \vee b \mapsto a \oplus b \oplus (a \wedge b)$, and $\neg x \mapsto 1 \oplus x$. An ANF equation distributes both $\wedge$ and $\vee$ operations algebraically, and is thus even less suitable than DNF as a representation. It is possible to ignore $\vee$ operations entirely by rephrasing the "choice" and "majority" f functions as ANF; however, distributing $\wedge$ is still very difficult and is akin to the difficulty of multiplication for multivariate polynomials (Samajder and Sarkar, 2013).

Sparse function representations Ternary trees (for example) can be used to represent a particular boolean formula in terms of variables, irrelevant variables, and negated variables which are assigned to "left", "mid", and "right" subtrees respectively. This representation is useful for minimising functions by moving variables between branches as operations are applied. However, it has been established (see Section 5.2) that SHA-1 is *not* such a function: each output bit relies upon every input bit. Therefore, ternary trees and other representations suitable for use with sparse boolean functions are not considered.

6.5 Summary

This chapter has laid out the framework that will be used by the majority of the representations that will be discussed in the next chapters. Not all representations are equally suitable; some characteristics that make a representation worth examining from a preimage perspective have been discussed, and the unsuitable representation of truth tables has been shown to reinforce the importance of this. Different categories of excluded representations, and the reasons for their exclusion, have been discussed. This sets the stage for discussing each of the worthwhile representations in Chapters 7 through 11 in an essentially self-contained way.

Chapter 7

Conjunctive Normal Form

This chapter examines a representation that is widely-used in the boolean satisfiability community: conjunctive normal form (CNF). The chapter begins be defining the form and describing how arbitrary formulas can be converted into it. Some time is then spent to discuss the best way to encode SHA-1 in such a form, taking into account the relevant literature on the subject. As it so happens, the CNF representation of any formula is easily consumed by powerful boolean satisfiability software; such software is therefore applied to the SHA-1 preimage problem.

A "normal form" for an equation is a canonical form for that equation. If two equations, which appear to be different upon visual inspection, have the same normal form, then they are equivalent. Conjunctive normal form represents an equation as a conjunction of disjunctions. Each set of disjunctions is called a *clause* (or *covering*), and each variable is called a *literal.* Clauses cannot be negated, but individual literals may be. The order of clauses is irrelevant.

Example 7.1. *CNF.* Consider the function represented by Table 6.1: $(x_0 \wedge x_1) \vee (x_2 \vee x_3) \wedge \neg(x_2 \wedge \neg x_0)$. It can be converted to CNF by using De Morgan's law to remove all negated clauses, distributing $\vee$ operations over $\wedge$ operations so that $a \vee (b \wedge c) \mapsto (a \vee b) \wedge (a \vee c)$, and then simplifying.

$$\begin{aligned}
&(x_0 \wedge x_1) \vee (x_2 \vee x_3) \wedge \neg(x_2 \wedge \neg x_0)\\
&= (x_0 \wedge x_1) \vee (x_2 \vee x_3) \wedge (\neg x_2 \vee x_0)\\
&= (x_0 \vee ((x_2 \vee x_3) \wedge (\neg x_2 \vee x_0)) \wedge (x_1 \vee ((x_2 \vee x_3) \wedge (\neg x_2 \vee x_0))\\
&= ((x_0 \vee (x_2 \vee x_3)) \wedge (x_0 \vee (\neg x_2 \vee x_0)) \wedge (x_1 \vee (x_2 \vee x_3)) \wedge (x_1 \vee (\neg x_2 \vee x_0))\\
&= (x_0 \vee x_2 \vee x_3) \wedge (x_0 \vee \neg x_2 \vee x_0) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_0)\\
&= (x_0 \vee x_2 \vee x_3) \wedge (x_0 \vee \neg x_2) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_0)\\
&= (x_0 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee x_0)
\end{aligned}$$

The clauses of a minimal CNF function can be viewed as a set of "filters" which exclude 0-valued rows of the truth table. Therefore, finding a preimage of a CNF function necessarily involves evaluating each of the clauses of the function, and discarding those rows which do not match. For functions where the number of variables is large, and the number of 1-valued rows is small, this process of elimination can take a great deal of computational effort.

Converting a function to CNF can be difficult; see Example 7.1. Though mechanical, the process involves computationally-expensive distribution of terms and application of boolean identities, and could result in an exponential increase in the size of the resulting CNF. For example, the formula $(a \wedge b) \vee (c \wedge d) \vee (e \wedge f)$ leads to the CNF $(a \vee c \vee e) \wedge (a \vee c \vee f) \wedge (a \vee d \vee e) \wedge (a \vee d \vee f) \wedge (b \vee c \vee e) \wedge (b \vee c \vee f) \wedge (b \vee d \vee e) \wedge (b \vee d \vee f)$. Note that the 3 initial clauses have become $2^3 = 8$ clauses, each of which is more complex than the initial clauses. In fact, this expansion of n clauses to 2^n clauses is not unusual, and many parts of the SHA-1 compression function — such as calculations involving $\oplus$ — lead to an exponential expansion of the resulting CNF when $\vee$s are distributed over $\wedge$s.

However, one of the great advantages of CNF is that any function can be converted to an *equisatisfiable* CNF via Tseitin encoding (Tseitin, 1983) with, at most, a linear increase in the number of terms. An equisatisfiable formula has additional variables, but is only satisfiable whenever the original formula is satisfiable. Tseitin encoding involves replacing each operation x_0 **op** x_1 with a corresponding CNF sub-expression that uses a new variable x_2:

- $x_0 \wedge x_1 \mapsto (\neg x_0 \vee \neg x_1 \vee x_2) \wedge (x_0 \vee \neg x_2) \wedge (x_1 \vee \neg x_2)$

- $x_0 \vee x_1 \mapsto (\neg x_0 \vee \neg x_1 \vee \neg x_2) \wedge (x_0 \vee x_2) \wedge (x_1 \vee x_2)$
- $\neg x_0 \mapsto (\neg x_0 \vee \neg x_2) \wedge (x_0 \vee x_2)$
- $x_0 \oplus x_1 \mapsto (\neg x_0 \vee \neg x_1 \vee \neg x_2) \wedge (\neg x_0 \vee x_1 \vee x_2) \wedge (x_0 \vee \neg x_1 \vee x_2) \wedge (x_0 \vee x_1 \vee \neg x_2)$

The new variable takes the value of the output of the expression, and can therefore be used in further expressions as a proxy for that output. Once a solution has been obtained, the new variables can be discarded and the values of the original variables substituted into the the original equation, with the same result being obtained.

A CNF formula can be stored compactly as a vector of clauses, where each clause may have a maximum of three terms. The index of the clause in the vector identifies it uniquely. This representation makes it easy to store a large formula with minimal overhead. It also makes it simple to create files in the *de facto* standard DIMACS format, in which a simple header line is followed by a single clause per line.

Finding a preimage for a CNF formula is equivalent to finding a set of inputs which will satisfy the formula. The ease of converting boolean formulae into CNF via the Tseitin transformation has led to CNF being accepted as the *de facto* standard for use with satisfiability (SAT) solvers, which exist to solve the well-known (and NP-complete) boolean satisfiability problem (Cook, 1971). Despite being theoretically unsolvable in computationally-feasible time, many practical boolean satisfiability problems can be solved heuristically using modern SAT solvers (Malik and Zhang, 2009). In the SAT literature, clauses are often called *constraints* since they constrain the possible solution space.

SAT algorithms may be split into two broad categories:

DPLL-based Davis-Putnam-Logemann-Loveland (Davis *et al.*, 1962) (DPLL) solvers (either backtracking or non-backtracking) try to find logical contradictions between variable assignments, and thus eventually derive an implication graph which contains no contradictions. Such an implication graph is a solution. Powerful modern solvers such as MiniSat (Eén and Sörensson, 2003) are DPLL-based. Conflict-Driven Clause Learning (CDCL) solvers are advanced variants of DPLL solvers.

Stochastic Local Search Whereas DPLL-based solvers work by proving relationships between the variables, stochastic local search (SLS) solvers – also known as “random walk” solvers – start with an assignment of random values to each variable. A limited number of variables is then flipped, and the solution which has the least number of unsatisfied clauses is chosen. Using this solution as the new assignment, the same procedure is repeated until a solution where *all* clauses are satisfied is obtained. (Biere, Heule, and van Maaren, 2009, Ch. 8)

The split is, in some instances, more theoretical than practical since there is a great cross-pollination of ideas and techniques between different families of solvers; indeed, solvers such as SATzilla (Xu, Hutter, Hoos, and Leyton-Brown, 2008) choose between what they believe to be the most "appropriate" method for a given problem. In the case of finding preimages, it is known that the formula *is* satisfiable (SAT) instead of unsatisfiable (UNSAT). Where a solution is known to exist, SLS solvers may significantly outperform DPLL-based solvers (Kautz and Selman, 1996; Selman, Kautz, Cohen *et al.*, 1993). However, when applied to a hash function, SLS solvers have performed much worse than their DPLL-based counterparts (Massacci, 1999).

A big drawback of SAT-solving solutions is that they are of limited generality. Although the basic algorithms used during DPLL-solving stay the same, some of the choices made by a solver are effectively arbitrary. After how many conflicts should a solver stop pursuing a branch of reasoning? Which variable should a solver try to find a value for? How much of an abandoned branch of reasoning might it be useful to keep for future reference? These questions are decided in different ways by different solvers (and different algorithm variants). A SAT solver, applied to a particular problem, is very sensitive to the parameters which govern its behaviour. The advantage of this is that there may be a particular set of parameters — Lingeling (Biere, 2013), for example, has approximately 340 tunable parameters! — which *could* find a SHA-1 preimage within a relatively short span of time. The disadvantage of such a solution is that it tells the computer scientist a great deal about the behaviour of various SAT-solving algorithms, but very little about the problem domain for which those particular parameters happened to work. It is therefore unlikely to lead to a cognitive breakthrough or generalisable understanding. Even worse, there is a possibility that a SAT solver will only find a solution for a particular problem case, and not for problems of a particular type, since "it is exactly the non-adversarial nature of practical instances that is exploited by SAT solvers" (Malik and Zhang, 2009). One practical consequence of this is that the time taken to solve a particular problem – even if the problem may be considered "similar" to a previously-solved problem – is very difficult to predict.

7.1 CNF encoding of SHA-1

The most relevant works in this regard are by Eén and Sörensson (2006); Nossum (2012); Legendre *et al.* (2012, 2014). Nossum's exceptionally comprehensive work discusses a minimal encoding of SHA-1 in order to find preimages, and conducts numerous experiments to identify the best ways to SAT-solve the resulting CNF.

The thesis of Nossum (2012) represents some of the most recent SHA-1 preimage research and focuses specifically on the application of SAT-solvers to the SHA-1 preimage problem. It begins, as this work has done, by attempting to find alternative ways to understand and encode the message expansion phase, and then tackles the equation for a_i. Most of the work was done on a reduced-round variant of SHA-1, focusing on the first 20-23 rounds; interestingly, this range ends just before the SAC is satisfied. While this work has focused on multiple representations, of which CNF is one, the primary reference for SAT-solving of SHA-1 remains Nossum (2012).

A SAT solver uses the technique of *unit propagation* to simplify a problem. A unit clause is a clause consisting of a single literal (which may be negated). When such a clause is encountered, all clauses containing the literal may be removed: if the problem is satisfiable, then the literal will make all clauses containing it satisfiable as well. In addition to this, if a clause containing the *negation* of the literal is encountered, then the literal may be removed from this clause: it cannot possibly be true in any solution, and can therefore not contribute to the meaning of the clause.

Eén and Sörensson (2006) highlight the importance of *arc consistency* as a worthwhile property of an encoding; an arc consistent encoding allows unit propagation to be used much more effectively, often to the extent of solving a problem entirely. A definition of arc consistency that is specific to CNF encoding is given by Eén and Sörensson (2006):

> **Definition.** Let $x = (x_1, x_2, \ldots, x_n)$ be a set of constraint variables, $t = (t_1, t_2, \ldots, t_m)$ a set of introduced variables. A satisfiability equivalent CNF translation $\varphi(x, t)$ of a constraint $\mathcal{C}(x)$ is said to be *arc-consistent* under unit propagation *iff* for every partial assignment σ, performing unit propagation on $\varphi(x, t)$ will extend the assignment to σ' such that every unbound constraint variable x_i in σ' can be bound to either True or False without the assignment becoming inconsistent with $\mathcal{C}(x)$ in either case.

More informally, a formula has constraints that must be satisfied by any solution; an equisatisfiable formula should try to represent these constraints faithfully; and an arc consistent equisatisfiable formula is one where an assignment leads unambiguously to constraint satisfaction or unsatisfiability. Thus, arc consistent representations of a formula may be efficiently solved for satisfiable solutions.

XOR constraints occur during message-expansion, during the encoding of addition, and during the parity function (used as the f-function in 40 of the 80 SHA-1 rounds). Unfortunately, they also involve the longest clauses (four 3-term clauses) and the largest number of these clauses (four, as opposed to a maximum of three to encode other operations). A naïve encoding of a XOR constraint such as $p \oplus q \oplus r$ thus involves taking any two of the variables, applying the Tseitin transformation, and then using the resulting additional Tseitin variable as a participant in another Tseitin transformation; viz.

$$\begin{aligned} p \oplus q \oplus r &= ((\neg p \vee \neg q \vee \neg t_0) \wedge (\neg p \vee q \vee t_0) \wedge (p \vee \neg q \vee t_0) \wedge (p \vee q \vee \neg t_0)) \oplus r \\ &= (\neg p \vee \neg q \vee \neg t_0) \wedge (\neg p \vee q \vee t_0) \wedge (p \vee \neg q \vee t_0) \wedge (p \vee q \vee \neg t_0) \wedge \\ &\quad (\neg t_0 \vee \neg r \vee \neg t_1) \wedge (\neg t_0 \vee r \vee t_1) \wedge (t_0 \vee \neg r \vee t_1) \wedge (t_0 \vee r \vee \neg t_1) \end{aligned}$$

Thus does a constraint involving n terms expand to 2^{n-1} 3-clause constraints. Bard (2007), observing that n causes problems of scale, suggests a pre-processing of n-term XOR-clauses to break them down into c-term XOR-clauses, $c < n$, where each new clause introduces an additional variable; c is called the *cutting number*.

Example 7.2. *Pre-processing XORclauses (Bard, 2007).* Assume that the 5-term formula $p \oplus q \oplus r \oplus s \oplus t$ must be converted to CNF. As above, this would require $2^{5-1} = 32$ 3-term clauses in a naïve Tseitin encoding. If we apply Bard's procedure using $c = 4$, however, we end up with the equisatisfiable clauses $p \oplus q \oplus r \oplus s \oplus x_0$ and $t \oplus x_0 \oplus x_1$. The single clause has been split into two clauses, each having a maximum of $c + 1$ terms, and the corresponding number of clauses when converted to CNF is $2^4 + 2^2 = 16 + 4 = 20$; 12 clauses fewer than would be obtained from a naïve encoding. When $c = 3$, the clauses $p \oplus q \oplus r \oplus x_0$ and $s \oplus t \oplus x_0 \oplus x_1$ would be obtained, for a total number of $2^3 + 2^3 = 8 + 8 = 16$ clauses. The cost of each clause is the introduction of an additional "dummy" variable, denoted by x_i above. According to Bard (2007), the optimal cutting number is 6.

It is also known that the clauses-to-variables ratio of a CNF formula is correlated with the difficulty of solving that formula, with ratios between 4 and 6 being most difficult to solve and ratios below 3.5 being relatively easy to solve; larger ratios are more often found in unsatisfiable formulae (Van Harmelen, Lifschitz, and Porter, 2008, p. 110-111). Nossum (2012) therefore attempts to reduce this ratio, encoding the addition step of a SHA-1 round by using the Espresso heuristic logic minimizer (Rudell and Sangiovanni-Vincentelli, 1987) to express formulae using the least number of terms; using the CryptLogVer toolkit (Morawiecki and Srebrny, 2013) for a similar reason; and using a straight Tseitin transformation (for comparison purposes).

By comparison, Legendre *et al.* (2014) hand-crafts the SHA-1 CNF encoding in an attempt to decrease the complexity of solving the formula, from the perspective of a DPLL-based SAT solver. The clauses-to-variables ratio is ignored in favour of simplifications and creating "logical bridges" (Legendre *et al.*, 2014, p. 16) — clauses containing only two variables — that may help during solving. Unfortunately, while improved results for the MD5 algorithm are demonstrated, improved results for SHA-1 are absent. The importance of this approach in the context of SHA-1 is therefore uncertain.

Table 7.1: CNF encodings, 80 rounds of SHA-1

Reference	Encoding	Clauses	Variables	Ratio	2-clauses
(Nossum, 2012)	Espresso	478,476	13,408	35.69	unknown
(Nossum, 2012)	CryptLogVer	248,220	44,812	5.54	unknown
(Nossum, 2012)	Simple	223,551	56,108	3.98	unknown
(Legendre *et al.*, 2014)	Hand-crafted	491,791	12,779	38.48	259
(Legendre *et al.*, 2014)	Hand-crafted, simplified	375,195	12,771	29.38	908

Both Legendre *et al.* (2014) and Nossum (2012) call out the addition step as being worthy of special effort when encoding. Nossum (2012) notes that the difficulty of the problem stems from the fact that "a binary k-bit ($k \geq 2$) ripple-carry adder encoded using the Tseitin transformation has $4 \times k - 4$ 'hidden' clauses, i.e. clauses which must be learnt through conflict propagation". In the case of 32-bit numbers, this would mean that $4 \times 32 - 4 = 124$ such hidden clauses exist. Nossum (2012) therefore try several approaches: using the Espresso heuristic logic minimizer (Rudell and Sangiovanni-Vincentelli, 1987) to express formulae using the least number of terms; using the CryptLogVer toolkit (Morawiecki and Srebrny, 2013) for a similar reason; and using a straight Tseitin transformation. The clauses, variables, and ratios for various 80-round encodings are presented as Table 7.1.

Table 7.2: The effect of Glucose 4.0 simplification options on a 210,121-variable, 629,597-clause CNF encoding of SHA-1

Simplification options	Clauses	Variables	2-clauses
`-elim`	356,395	66,457	81,446
`-elim -asymm`	350,673	64,901	85,185
`-elim -grow=50 -asymm`	482,577	23,653	16,476
`-elim -grow=100 -asymm`	486,735	16,730	5,687
`-elim -grow=200 -asymm`	470,176	12,700	1,314
`-elim -grow=500 -asymm`	506,669	11,587	751
`-elim -grow=10000 -asymm`	2,486,170	9,277	534

No such encoding optimisations have been enacted in this work. A simple, naïve encoding of the SHA-1 algorithm, as described by Equation 4.7, results in 210,121 variables and 629,597 clauses, giving a clause-to-variable ratio of 2.996 — which is a "better" ratio than any listed in Table 7.1. Legendre *et al.* (2014) focuses on longer clauses with more 2-clause bridges, giving a higher ratio. When this naïve encoding is passed to the Glucose 4.0 solver[1] (Audemard and Simon, 2009; Eén and Sörensson, 2003) along with suitable simplification options, the result is a CNF encoding with 470,176 clauses, 12,700 variables, and 1,314 2-clauses — arguably a "better" encoding than the hand-crafted one, under the assumption that simplification works towards making solving easier. Table 7.2 shows the effect of using various simplification options on the simple CNF encoding that has already been described.

Table 7.3: The effect of Glucose 4.0 simplification options on a 267,603-variable, 857,477-clause CNF encoding of SHA-1

Simplification options	Clauses	Variables	2-clauses
`-elim`	582,924	98,119	81,093
`-elim -asymm`	577,616	96,653	85,542
`-elim -grow=50 -asymm`	983,082	38,184	16,690
`-elim -grow=100 -asymm`	1,185,483	28,789	5,323
`-elim -grow=200 -asymm`	1,186,745	24,930	1,247
`-elim -grow=500 -asymm`	2,304,868	20,423	659
`-elim -grow=10000 -asymm`	14,422,346	14,835	448

For comparison purposes, Table 7.3 presents figures for an encoding of SHA-1 that uses the bitpattern w-calculation method explained in Section 4.1. The figures demonstrate that the trends caused by simplification are similar, even when the initial encoding of

[1]Glucose version 4.0, with Glucose Syrup

the problem is quite different: the number of variables decreases, the number of clauses increases, and the number of 2-clauses decreases. The 2-clauses column, in particular, is interesting because of how similar it is between the encodings.

Not directly evident from Tables 7.2 and 7.3, but worthwhile to note, is the time and space costs of different initial encodings. More effort spent on simplification requires more computational power, and the final rows of Tables 7.2 and 7.3 took 7 minutes and 120 minutes respectively, and resulted in simplified DIMACS files that were 113Mb and 858Mb respectively.

No encoding tricks, such as those already described, have been used to simplify the CNF encoding in this work. Taking into consideration the previous work on this topic, the experimental results detailed above, and the possible advantages and disadvantages, the conclusion reached is that the built-in simplification routines used by modern solvers (see, for example, Eén and Biere (2005)) are likely to be powerful enough for all practical purposes; there is little to be gained by hand-tweaking a SHA-1 encoding.

There are basic encodings which appear to be objectively worse to begin from; a comparative examination of Tables 7.2 and 7.3 appears to show that the bitpattern w-formulation leads to one such encoding. However, the results of the following section will demonstrate that this appearance is deceptive.

7.2 SAT-solving

An exploration of CNF as a representation would be incomplete without some attempt to find preimages using SAT solving: the DIMACS format for CNF has become synonymous with SAT solving. The Glucose (Audemard and Simon, 2009; Eén and Sörensson, 2003), YalSAT[2] (Biere, 2014), Plingeling[3] (Biere, 2013), and CryptoMiniSat[4] (Soos and Lindauer, 2015) SAT solvers were chosen as being representative of a cross-section of SAT-solving approaches. Glucose is a modern, state-of-the-art CDCL solver; YalSAT is a modern take on a Stochastic Local Search solver; Plingeling is a SAT solver that attempts to exploit multi-core architectures; and CryptoMiniSat is a well-regarded open-source CDCL solver, originally targeted towards solving cryptographic problems, which supports a XOR-clause extension to the DIMACS format. All solvers were run in their default configurations and

[2]YalSAT version 03l

[3]Plingeling version bbc-9239380-160707

[4]CryptoMiniSat version 5.0.0 (with gaussian elimination)

solvers which did not find a solution within 10 minutes were terminated. This methodology is similar to that employed in Nossum (2012, Chapter 4).

Table 7.4: SAT-solving to find a preimage

Input bits	Solver	Time taken (s)	
		standard	bitpattern
6	Glucose	1.3	1.3
	YalSAT / ProbSat / CSCCSat14	–	–
	Plingeling	1.0	1.4
	CryptoMiniSat	2.5	2.2
12	Glucose	16.4	7.1
	YalSAT / ProbSat / CSCCSat14	–	–
	Plingeling	15.9	7.6
	CryptoMiniSat	3.6	16.6
16	Glucose	135.6	252.0
	YalSAT / ProbSat / CSCCSat14	–	–
	Plingeling	25.8	250.0
	CryptoMiniSat	256.1	18.5
18	Glucose	403.6	138.2
	YalSAT / ProbSat / CSCCSat14	–	–
	Plingeling	82.7	463.2
	CryptoMiniSat	–	181.9
20	Glucose	227.8	132.0
	YalSAT / ProbSat / CSCCSat14	–	–
	Plingeling	–	–
	CryptoMiniSat	–	–
22	Glucose	–	–
	YalSAT / ProbSat / CSCCSat14	–	–
	Plingeling	–	–
	CryptoMiniSat	–	–

SAT solver results are presented in Table 7.4. YalSAT, in line with the reported results of Massacci (1999), was unable to solve any of the problems. To check whether this was a problem with YalSAT or with the SLS approach, two other independently-developed SLS solvers (ProbSat (Balint and Schöning, 2012) and CSCCSat14 (Luo, Cai, Wu, and Su, 2013, 2014)) were applied to the smallest (6-bit input) problem. Both failed to arrive at a solution within 10 minutes, and this confirms that the problem is likely to be the

SLS approach rather than the YalSAT solver itself. CDCL-based solvers worked somewhat better, with Glucose outperforming both Plingeling and CryptoMiniSat. The time taken to find solutions varied significantly and unpredictably, and neither the standard w-encoding nor the bitpattern w-encoding showed itself to be definitively superior. Glucose seemed to find the bitpattern w-encoding to be easier to solve for bit-lengths 18 and 20, but more difficult for bit-length 16; Plingeling found the bitpattern w-encoding to be uniformly harder to work with; and CryptoMiniSat seemed to find the bitpattern w-encoding to be uniformly easier to work with. This unpredictability casts some doubt on the weight that should be given to "second-guessing" a solver by simplifying or hand-tweaking a CNF encoding.

No solver could find a preimage for more than 20 bits of input. With that being said, finding a preimage for hash inputs of ≤ 20 bits on consumer-level hardware is no mean feat; if anything, it demonstrates the enormous advances that have been made in the field of SAT-solving over the past decades. The raw speed of the computation is not what is impressive: brute-force search is many times faster. However, SAT-solvers resort to brute-force when all other heuristics fail, and reduced times such as Glucose's $\approx$ 227s 20-bit solution vs. its $\approx$ 403s 18-bit solution indicate that heuristics are being used to reduce the time spent. It is the increased *semantic* understanding of the underlying problem that provides some hope for a breakthrough on this front. Another two decades of progress in the field may make SAT-solving for larger bit-lengths much easier. If a solver is able to find preimages for inputs that are $\approx$ 160 bits, then it is reasonably sure to find a preimage for most SHA-1 hashes, given their balance (Bellare and Kohno, 2004), strict avalanche criterion characteristics (see Section 5.2), and the formalisation of aPre laid out by Rogaway and Shrimpton (2009).

These results serve to update some of the results of Nossum (2012), and demonstrate the interaction between different SAT-solvers and the encoding of the problem. Note that where that work focused equally on the encoding of SHA-1 and the heuristics employed by SAT-solvers, this work has not examined the heuristics in any detail due to the already-described difficulty in generalising any results since the interactions between multiple heuristics are not well-understood. Interested readers who wish to discover more about SAT-solving and CNF as they relate to SHA-1 will find much of interest in the work of Nossum (2012).

7.3 Summary

As a representation for SHA-1, CNF is reasonably compact and, after conversion to DIMACS format, facilitates the use of powerful SAT-solving software. However, a major drawback is that a CNF formula is inevitably equisatisfiable, and the added Tseitin variables add little but noise. Furthermore, finding a set of variable assignments which satisfy a particular set of constraints is the goal of both preimage-finding using a CNF representation, and boolean satisfiability. Reducing the SHA-1 preimage problem to a known NP-complete problem domain offers one way to solve smaller problems, but provides neither the insight nor the ability to approach larger problems.

Chapter 8

Disjunctive Normal Form

The conjunctive normal form discussed in the previous chapter represents formulae as a conjunction of disjunctions. It is equally possible to represent formulae as a disjunction of conjunctions instead. This chapter begins by considering the cost of the operations required to represent a SHA-1 preimage, and then segues into a discussion of the disjunctive normal form (DNF) and its relative advantages and disadvantages (as compared to CNF). The difficulties of manipulating such a representation are discussed, and state-of-the-art minimization software is used to automate the process.

The representation that is chosen can greatly affect the cost of operations to be carried out using that representation. Calculation of most a_r^i equations involves a large number of $\wedge$, $\vee$, $\oplus$, and $\neg$ operations, most of which are performed upon equations themselves. For example, assuming that dependent a-representations already exist, the calculation of a_{30}^{10} involves

- at least four $\oplus$ operations to calculate w,
- two further $\oplus$ operations to calculate f,
- thirty-nine $\wedge$ operations, one $\oplus$ operation, twenty $\vee$ operations, and six $\neg$ operations to calculate $v0$,
- thirteen $\wedge$ operations, and nineteen $\vee$ operations to calculate $v1$,
- five $\oplus$ operations and a $\neg$ operation to calculate a.

This gives a total of 110 operations to obtain a single equation. Some equations (notably a_r^{26} and a_r^{25} equations) will, of course, involve significantly fewer operations. Nevertheless, there are $32 \times 80 = 2560$ a_r^i equations to calculate, so if an average of 90 operations per equation is assumed then it can be estimated that approximately 230,000 operations will be required. In fact, the actual number obtained for a reference implementation is 218,564 operations (103,840 $\wedge$'s, 72,500 $\vee$'s, 21,664 $\oplus$'s, and 20,560 $\neg$'s). Various micro-optimisations might be applied to reduce this number slightly, but it would be surprising if this number could be drastically reduced. Therefore, it can be assumed that there are upwards of 200,000 operations required to get to a_{80}^{27}.

Disjunctive normal form represents an equation as a disjunction of conjunctions. Each set of conjunctions is called a *clause* (or *covering*), and each variable is called a *literal*. Clauses cannot be negated, but individual literals may be. The order of clauses is irrelevant.

The clauses of a minimal DNF function can be viewed as expressing sets of 1-valued rows of the truth table. Therefore, each clause of a DNF function is equivalent to one or more preimages. This makes DNF an ideal form for finding a preimage very quickly and easily: each clause represents a different preimage. However, it is difficult to determine the values which do *not* satisfy a DNF function, for much the same reason that it is difficult to determine the values which *do* satisfy a CNF function.

Unfortunately, constructing a DNF is much more difficult than constructing a CNF. Recall that a CNF is relatively quick and easy to construct, using the Tseitin encoding, but is susceptible to an exponential increase in size without using the encoding. Converting a function to DNF may entail a similar exponential increase in size, but there is no analogous encoding trick for a DNF.

This situation leaves the researcher with a conundrum: CNF is easy to construct, but makes it difficult to find preimages; DNF is difficult to construct, but makes it easy to find preimages. It would therefore be best to construct using CNF, but find preimages using a DNF. The difficulties of this approach are best illustrated by an example.

Example 8.1. *Converting CNF to DNF.* Assume that one wishes to find all satisfying assignments for the function $f(x_{0..3}) = (x_0 \wedge x_1) \vee (x_2 \vee x_3) \wedge \neg(x_2 \wedge \neg x_0)$, which has the truth table shown in Table 6.1 on page 89. The minimal DNF is $(x_0 \wedge x_1) \vee (x_0 \wedge x_2) \vee (\neg x_2 \wedge x_3)$. It can readily be seen that each clause of this DNF is equivalent to one or more sets of satisfying assignments (or preimages).

The minimal CNF for the function is $(x_0 \vee \neg x_2) \wedge (x_0 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$. By De Morgan's Law, the negation of a DNF is a CNF (and vice versa). Therefore, it seems natural to try to construct a DNF is by constructing a CNF of a function's negation, and then negating that CNF. In effect, this is simply an assertion of the fact that $\neg\neg f(x) = f(x)$, with different forms being used to make the computation easier.

The negation of the original function is $(\neg x_0 \vee \neg x_1) \wedge (\neg x_2 \wedge \neg x_3) \vee (x_2 \wedge \neg x_0)$. The minimal CNF of this is $(\neg x_0 \vee \neg x_1) \wedge (\neg x_0 \vee \neg x_2) \wedge (x_2 \vee \neg x_3)$, and by applying de Morgan's law again, the resulting function becomes $(x_0 \wedge x_1) \vee (x_0 \wedge x_2) \vee (\neg x_2 \wedge x_3)$ — which is exactly the form that is sought.

Example 8.1 shows how easily CNF may be converted to DNF. However, the difference between "equisatisfiable" and "equivalent" means that this approach is not viable: recall that the Tseitin-encoded CNF is equisatisfiable, and not equivalent. When the Tseitin encoding is used, the resulting CNF is

$$(\neg x_0 \vee \neg t_0) \wedge (x_0 \vee t_0) \wedge (\neg x_1 \vee \neg t_1) \wedge (x_1 \vee t_1) \wedge (\neg x_2 \vee \neg t_2) \wedge (x_2 \vee t_2) \wedge (\neg x_3 \vee \neg t_3) \wedge (x_3 \vee t_3) \wedge (\neg t_0 \vee \neg t_1 \vee \neg t_4) \wedge (t_0 \vee t_4) \wedge (t_1 \vee t_4) \wedge (\neg t_2 \vee \neg t_3 \vee t_5) \wedge (t_2 \vee \neg t_5) \wedge (t_3 \vee \neg t_5) \wedge (\neg x_2 \vee \neg t_0 \vee t_6) \wedge (x_2 \vee \neg t_6) \wedge (t_0 \vee \neg t_6) \wedge (\neg t_4 \vee \neg t_5 \vee t_7) \wedge (t_4 \vee \neg t_7) \wedge (t_5 \vee \neg t_7) \wedge (\neg t_5 \vee \neg t_6 \vee \neg t_8) \wedge (t_5 \vee t_8) \wedge (t_6 \vee t_8)$$

Tseitin variables have been given the names $t_{0..8}$. Note that this form is significantly larger than the minimal CNF given before. However, as has been shown, the maximum increase in formula size is linear, and the Tseitin encoding is therefore more generally applicable than a direct translation. Negating the Tseitin-encoded CNF gives the DNF

$$(x_0 \wedge t_0) \vee (\neg x_0 \wedge \neg t_0) \vee (x_1 \wedge t_1) \vee (\neg x_1 \wedge \neg t_1) \vee (x_2 \wedge t_2) \vee (\neg x_2 \wedge \neg t_2) \vee (x_3 \wedge t_3) \vee (\neg x_3 \wedge \neg t_3) \vee (t_0 \wedge t_1 \wedge t_4) \vee (\neg t_0 \vee \neg t_4) \vee (\neg t_1 \wedge \neg t_4) \vee (t_2 \wedge t_3 \wedge \neg t_5) \vee (\neg t_2 \wedge t_5) \vee (\neg t_3 \wedge t_5) \vee (x_2 \wedge t_0 \wedge \neg t_6) \vee (\neg x_2 \wedge t_6) \vee (\neg t_0 \wedge t_6) \vee (t_4 \wedge t_5 \wedge \neg t_7) \vee (\neg t_4 \wedge t_7) \vee (\neg t_5 \wedge t_7) \vee (t_5 \wedge t_6 \wedge t_8) \vee (\neg t_5 \wedge \neg t_8) \vee (\neg t_6 \wedge \neg t_8)$$

Each term contains at least one Tseitin variable, and the ability to obtain preimages by inspection has effectively been removed.

A DNF formula can be represented easily using bits: a bit to indicate whether a variable is present, and a bit to indicate whether the variable is complemented or not.

Example 8.2. *Representing a DNF formula using bits.* The function in Example 8.1 uses four variables and three clauses. Therefore, two bit-vectors (*present* and *complemented*) of four bits each would be used to represent it:

present	1100	1010	0011
complemented	0000	0000	0010

Since there is no Tseitin-analogous encoding for DNF functions, size must be kept in check by attempting to minimise (or reduce) the number of terms in the formula. Minimizing a DNF formula is a matter of applying certain identities to reduce the number of terms. Some identities in this regard are listed below.

Elimination $(a \wedge b) \vee (\neg a \wedge b) = b$

Simplification $(a \wedge b \wedge c) \vee (a \wedge \neg b) = (a \wedge c) \vee (a \wedge \neg b)$

Absorption (positive) $a \vee (a \wedge b) = a$

Absorption (negative) $\neg a \vee (a \wedge b) = \neg a \vee b$

Retention $a \vee (b \vee c) = a \vee b \vee c$

Building up an equation using bits to represent clauses in a normal form is straightforward, but expensive. Whether a top-down or bottom-up approach is used, a clause added to an equation may need to be distributed over all the clauses already existing in that equation. Since there may be 2^n clauses in a minimal representation, this quickly becomes very expensive. Furthermore, distributing a new clause over the existing clauses does not result in a minimal representation; for this to happen, identities such as those mentioned above must be applied. Since the application of one identity may make it possible to apply a different identity, the minimization process is recursive and very time-intensive.

In fact, the minimization of a DNF formula has been shown to be an NP-complete problem (Umans, 1998), and circuit minimization in general is NP-complete as well (Buchfuhrer and Umans, 2011). These results show that it is impractical to obtain an exactly-minimized formula, given a non-trivial input size; however, many NP-complete problems

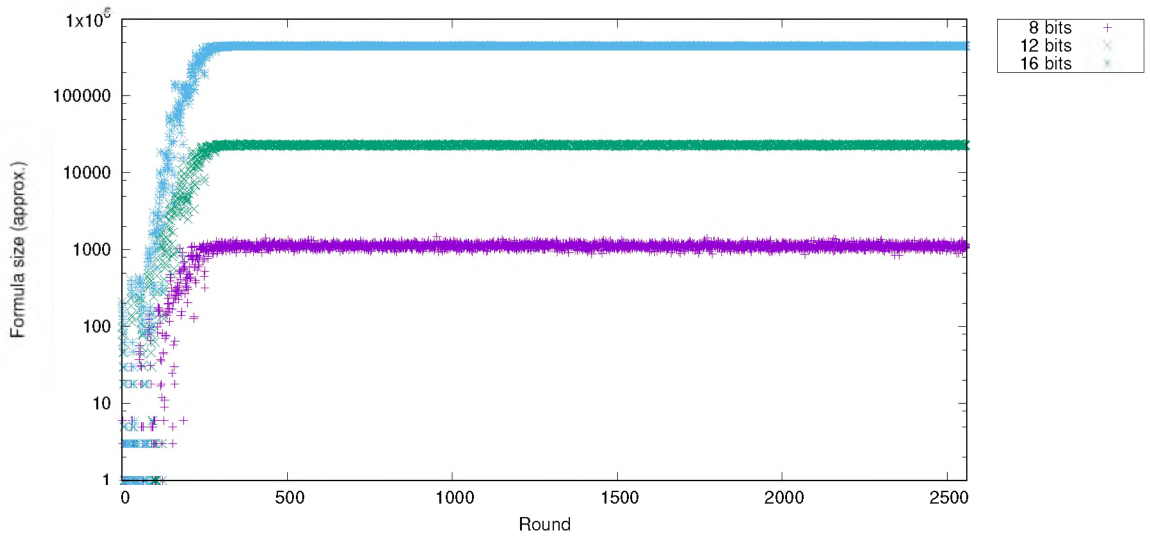

Figure 8.1: Espresso's diminishing returns

admit to heuristic "good enough" solutions and the effectiveness of these remains to be evaluated.

Well-known heuristic tools such as Espresso (Rudell and Sangiovanni-Vincentelli, 1987) are able to minimize an equation adequately, but grow increasingly expensive to apply as the number of variables and the number of terms increases. Figure 8.1 shows the size that Espresso reduces a 8-, 12-, and 16-variable SHA-1 inputs to, as the equation is being built up in a bottom-up manner. The formula size is approximate, measured as the number of characters in a string-form equation using Espresso's `eqntott` output, but correlates very strongly with the number of clauses and terms in the minimized equation. It is evident that, even before the data-word rounds are exhausted, the minimization has failed to have any significant effect.

An alternative, and in some instances superior, method of minimization is implemented by the BOOlean Minimizer II (BOOM-II)[1] (Fišer and Kubátová, 2004, 2006). Although this method results in smaller DNF representations, the cost (in time) of using it is much more unpredictable, ranging from ≈ 1 second to ≈ 25 seconds. It is also unable to handle even the limited input sizes that Espresso does: 16-variable inputs can take between 10 minutes and 2 hours to minimize, and this time does not appear to be entirely dependent on the round of the input. BOOM-II's algorithm does make more use of randomness than Espresso's, and it is hypothesised that it is this increased randomness that leads to the increased unpredictability. As a consequence, few 16-variable SHA-1 inputs are shown in Figure 8.2.

[1]BOOM-II version 2.7 (`http://ddd.fit.cvut.cz/prj/BOOM/`)

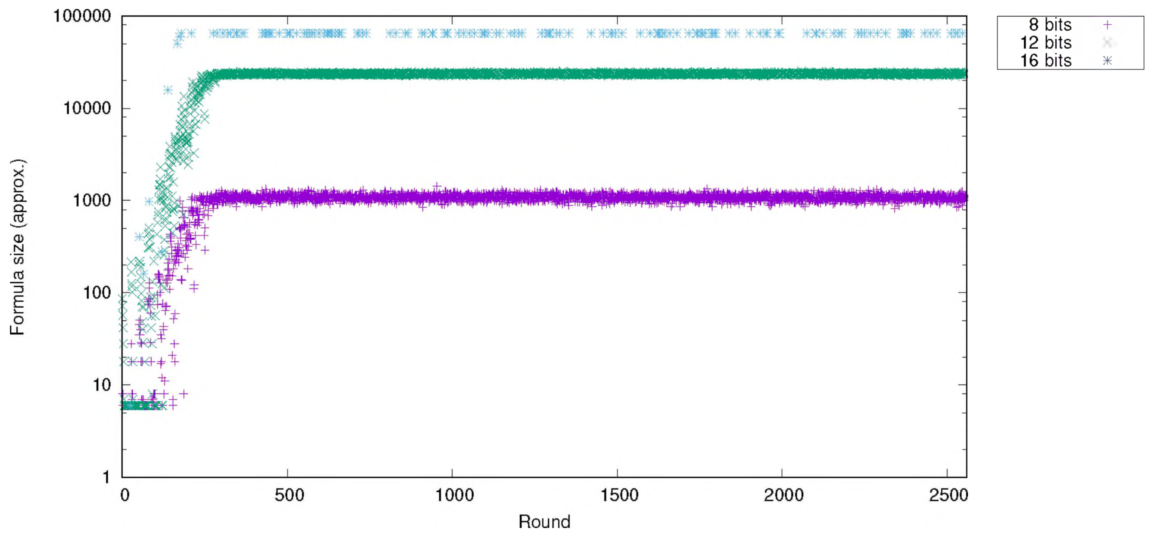

Figure 8.2: BOOM-II's diminishing returns

It is clear that BOOM-II does a better job than Espresso, but at a higher computational cost, when $\omega \geq 12$. For smaller input sizes ($\omega = 8$), BOOM-II does a much better job at a comparable computational cost to Espresso.

8.1 Summary

DNF is an ideal representation for the final preimage. However, constructing a DNF is exceptionally difficult since there is no Tseitin-equivalent encoding. Minimizing a DNF exactly is computationally infeasible; it has been shown to be an NP-complete problem. Unfortunately, the experiments conducted in this chapter demonstrate that heuristic approaches to achieve "good enough" results are not feasible as the input size grows: they either fail to produce the desired result (in the case of Espresso) or take an exorbitant and unpredictable amount of time (in the case of BOOM-II).

Chapter 9

Reduced Ordered Binary Decision Diagram

CNF and DNF are both “typical” representations of a formula, in the sense that they both represent a formula using variables and operations. However, it is possible to represent a formula — sometimes very compactly — using *choices* instead, and the *decision diagram* representation discussed in this chapter does exactly that. The chapter begins by introducing the representation and describing its salient properties. A powerful software package that can manipulate the representation is used to examine the scalability of the representation in the context of the SHA-1 preimage problem. Lastly, decision diagram variations are discussed, and an attempt is made to find a more suitable way of encoding the preimage problem using decision diagrams.

A reduced ordered binary decision diagram (ROBDD, though conventionally shortened to BDD) is a directed, rooted, acyclic graph which represents a boolean formula (Bryant, 1986; Wegener, 1994; Knuth, 2011). Each node in a BDD represents a variable v and has two children: the “high” or “then” child, which represents the “decision” of setting $v = 1$, and the “low” or “else” child, which represents the “decision” of setting $v = 0$ – hence the terminology of “decision diagram”. Many other types of decision diagram exist, such as MDD (Multi-Valued Decision Diagram), ADD (Algebraic Decision Diagram), and ZBDD (Zero-Suppressed Binary Decision Diagram). The BDD representation is both relatively simple and applicable to the problem domain, and will therefore be discussed first.

More formally, each level of a BDD splits the problem space into two based on Boole’s expansion theorem (Boole, 1854) $f(v_0, v_1, ..., v_n) = (v_0 \wedge f(0, v_1, ..., v_n)) \vee (v_0 \wedge f(1, v_1, ..., v_n))$, more commonly known as the Shannon expansion (Shannon, 1949b). One branch represents the result when the variable is 1-valued, and the other branch represents the result when the variable is 0-valued.

A BDD has the desirable property of being able to represent certain formulae with surprising brevity; however, the brevity of the BDD is largely dependent on the formula and the variable ordering that is chosen (Langberg, Pnueli, and Rodeh, 2003). The *size* of a BDD is the number of nodes in the BDD, and could be up to 2^{n-1} for an n-variable formula in a worst-case scenario. Such a scenario can arise if an inefficient variable ordering is chosen; however, choosing a suitable variable ordering is an NP-hard problem (Bollig and Wegener, 1996; Sieling, 2002). Heuristics exist to guess at a suitable variable ordering with a fair degree of success, and research on this topic is proceeding at a rapid pace (Minato, 2013). With that said, there are particular functions that are known to have approximately 2^n nodes when represented as a BDD, no matter which variable ordering is chosen (Bryant, 1991).

To create a ROBDD, a variable ordering must be chosen. A particular variable ordering results in a canonical BDD: in other words, equivalent functions will result in the same BDD, no matter what their original formulation was. The first variable serves as the root of the diagram, as described above, with the "then" and "else" edges both leading to the next variable in the ordering. After all variables have been ordered, the leaves of the graph will be either 0 or 1, and the graph is now an OBDD. To reduce the OBDD, the following rules are applied repeatedly:

1. Merge nodes which have identical (i.e. isomorphic) subgraphs; and
2. Remove nodes which have identical subgraphs.

Implementation optimizations which make the reduction of a OBDD more efficient are possible, but not necessary to understand at the level of semantics. The first rule reduces the number of nodes by allowing node functionality to be shared in the graph. The second rule eliminates duplicate nodes.

Example 9.1. *ROBDD creation.* Assume that a function $f(a, b, c, d) = (a \wedge b) \vee (b \wedge \neg c) \vee d$ is used. Using the variable ordering d, c, b, a, the resulting OBDD is shown as Figure 9.1.

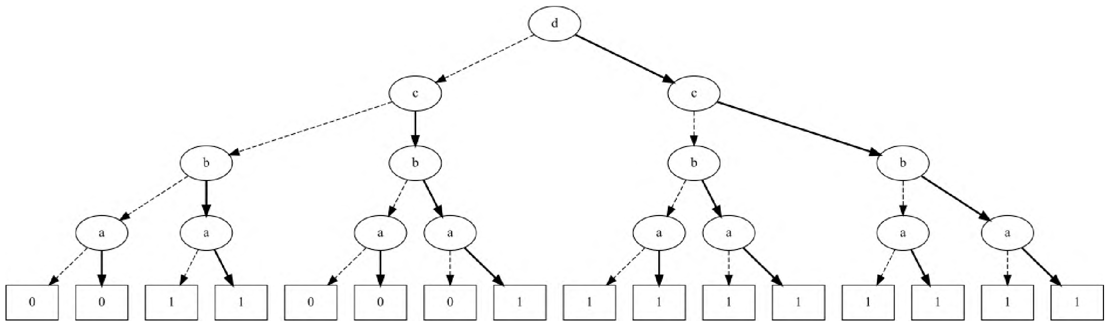

Figure 9.1: Example: ordered binary decision diagram

After applying rule (1) and merging identical subgraphs, the graph is partially reduced and has a size of 11. A fully-reduced graph, which is the final form of the ROBDD, has a size of 7. Partially-reduced and fully-reduced graphs are shown as Figure 9.2. Any equivalent formulation of the formula (such as $f(a, b, c, d) = d \vee (b \wedge (a \vee c))$) will have exactly the same ROBDD representation, given the same variable ordering.

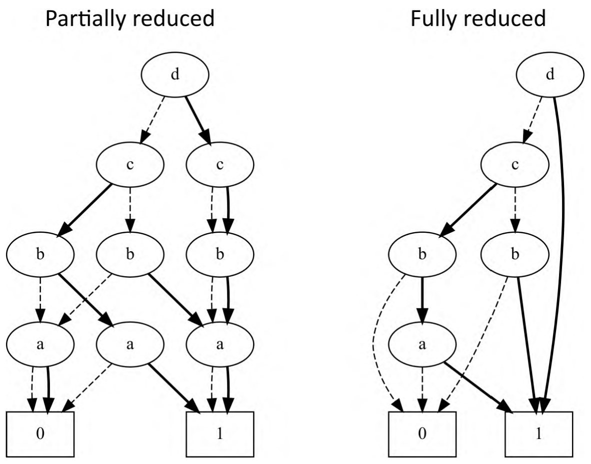

Figure 9.2: Example: reduction of an ordered binary decision diagram

Boolean operations may be performed between BDDs, resulting in another BDD. Such operations are typically recursive, involving the modification of subtrees within the graph, but an optimised implementation can reduce the cost of recursive operations through memoization (Bryant, 1992). In an optimised implementation, the cost of an operation that combines BDDs x an y is $|x| \cdot |y|$. Since a particular variable-ordering of the graph is canonical, the only way to reduce the size of the tree is by rearranging the order of the variables.

The state-of-the-art CU Decision Diagram (CUDD)[1] (Somenzi, 2015) package was used

[1]CUDD version 3.0.0

to create and manipulate BDDs. Importantly, the software includes nineteen different reordering heuristics (and heuristic variants) that aim to reduce the size of a BDD. The size of a BDD is the primary factor that affects the efficiency of operations, and consequently the scalability of BDDs as a representation.

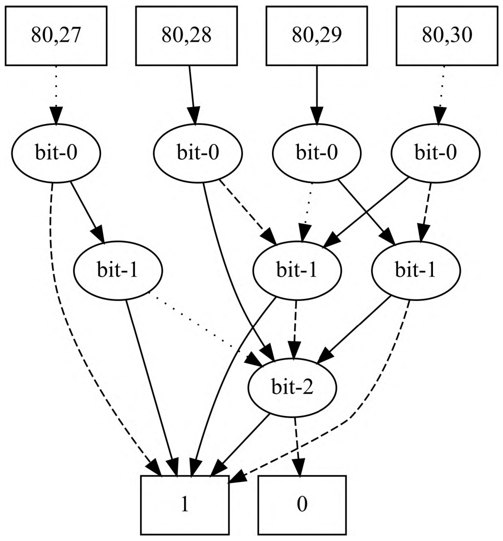

Figure 9.3: BDD for 3-input SHA-1

CUDD's BDDs are, strictly speaking, an extended form of ROBDD called a Complemented Reduced Ordered Binary Decision Diagram (CROBDD, more commonly shortened to CBDD). While a ROBDD has two possible types of edges – the "then" and "else" edges a CBDD has an additional type, the "complement" edge, which indicates logical negation (Drechsler and Sieling, 2001). A CBDD is equivalent to a ROBDD, and can easily be translated into an ROBDD (Madre and Billon, 1988); however, a CBDD may have a somewhat smaller representation than a ROBDD and can be manipulated more efficiently (Somenzi, 2001). Figure 9.3 shows such a CBDD, where dotted lines indicate complemented edges, dashed lines indicate "else" edges, and solid lines indicate "then" edges. The boxes at the top of the figure show the outputs $a_{80}^{27..30}$, with the boxes at the bottom being the constant terminal nodes.

Example 9.2. *Interpreting a ROBDD.* Using Figure 9.3 as an example, assume that one wanted to find the value of a_{80}^{27}, given the 3-bit input vector 100. Let $\bullet$ be an as-yet-unresolved result; initially, $a_{80}^{27} = \bullet$. Starting at the a_{80}^{27}, one encounters a complementation edge, giving $a_{80}^{27} = \neg(\bullet)$. The value of the 0th bit is 1, so the "then" edge is taken. This brings us to the 1st bit, which is 0; and so the "else" edge, which is a complemented edge, is taken; $a_{80}^{27} = \neg(\neg(\bullet))$. Finally, the 2nd bit is 0, so the "else" edge is taken, giving the final value of 0; therefore, $a_{80}^{27} = \neg(\neg(0)) = 1$ when the input vector is 100.

The importance of arc-consistency has already been covered in Chapter 7. One useful property of a BDD is that it is always arc-consistent (Eén and Sörensson, 2006); indeed, Example 9.2 would not be possible if the BDD was *not* arc-consistent.

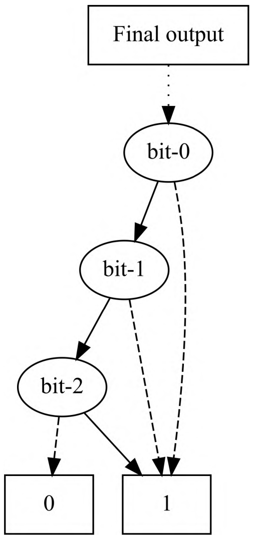

Figure 9.4: Preimage for a 3-input SHA-1 BDD

Finding a preimage using a BDD is similarly simple: by forcing the output of the BDD to be a constant value via Equation 4.9 or 4.10, a preimage is found by tracing the transitions down the tree that lead to a non-constant node, whenever the option is given to do so. Figure 9.4 illustrates this for a 3-input BDD that has had the final output fixed to 1, using the hash value `4754c111532732b8c54c97538a735fdc20cc4568`. Using the same notation as the previous example, and beginning at the output, $1 = \neg(\bullet)$. The "else" child leads to the constant node 1, and hence a contradiction; therefore, the "then" child is taken. Following the same logic until the terminal node 0 gives the identity $1 = \neg(0)$. The transitions taken to reach the terminal node are "then", "then", and "else", from bits 0, 1, and 2 respectively. The corresponding mappings for bits v_n are $v_0 = 1$, $v_1 = 1$, and $v_2 = 0$; and, indeed, the 3-bit input 110 results in the hash value `4754c111532732b8c54c97538a735fdc20cc4568`.

The importance of each output bit during preimage creation is shown by Figure 9.5, which examines the size of the BDD as each output bit is combined using $\wedge$. It can readily be seen that the number of nodes that need to be combined to find a preimage is $\approx \omega$. If the trend continues, and there is little reason to suspect that it would not, then it would take ≈ 160 combined output bits to find a preimage for a 160-bit input.

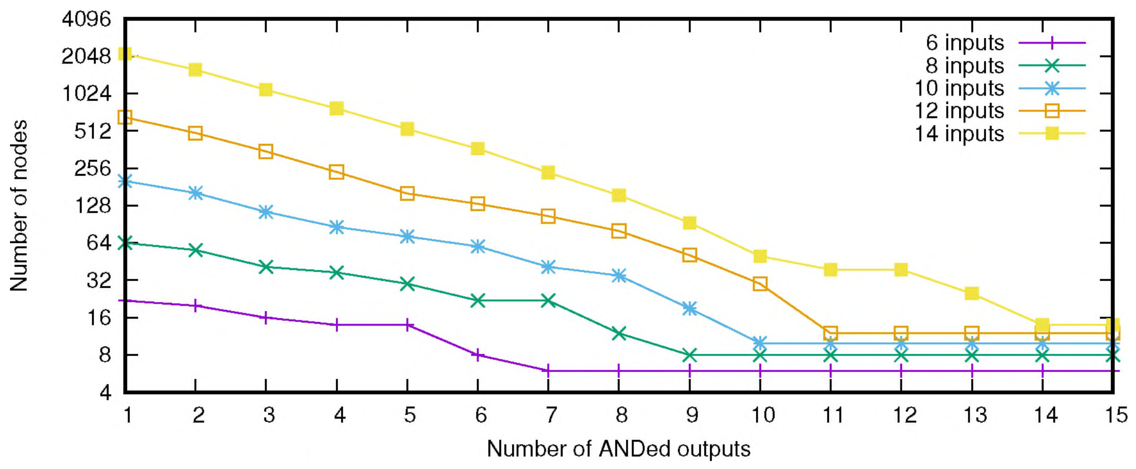

Figure 9.5: Node reduction due to output node combination

An important point to note is that the number of inputs strongly influences the impact of each output bit. Schnorr (1989) points to multiplicative complexity, which can be thought of as the sum of the degrees of each product term in the function's ANF, as a measure of boolean function complexity. This result implies that the multiplicative complexity of SHA-1 is $min(\omega, 160)$.

Such complexity is less than the upper bound for multiplicative complexity, shown by Boyar, Peralta, and Pochuev (2000) to be $\approx 2^{\frac{\omega}{2}-1}$. It can nevertheless be said that the SHA-1 compression function has the maximum possible multiplicative complexity for a collision-resistant hash function. This is because a function which achieved a greater multiplicative complexity would need to have multiple product terms in its ANF — and multiple product terms entail a collision.

Example 9.3. *Exceeding ω multiplicative complexity necessitates a collision.* Consider a function $f(x) = \mathbb{Z}_2^n \to \mathbb{Z}_2^m$, and an inverting function $f^{-1}(x) = \mathbb{Z}_2^m \to \mathbb{Z}_2^n$. By definition, $f^{-1}(f(x)) = x'$, where x' is either preimage or second-preimage. If $x \neq x'$, then there are at least two inputs that collide.

Now consider the multiplicative complexity of f^{-1}. If it is n, then every bit in a vector $\{0,1\}^n$ must be combined using $\wedge$ to produce the output: the ANF must be equivalent to $x_0 \wedge x_1 \wedge \cdots \wedge x_n$, where each x_i may be negated. If it is greater than n, then a second product term must exist since repeating a variable x_i with the same polarity would be redundant and repeating a variable x_i with an inverted polarity would cause the term to be 0.

Assume that two product terms exist in the ANF of f^{-1}, such that $f^{-1} = a \oplus b$ where $a, b \in \{0,1\}^n$. It is evident that there must be at least two inputs which, when XORed together, produce this result. The maximum possible multiplicative complexity that can be achieved while remaining collision-free is therefore n.

As a more concrete example, let $f^{-1} = (x_0 \wedge x_1 \wedge x_2 \wedge x_3) \oplus (\neg x_0 \wedge x_1 \wedge \neg x_2 \wedge x_3)$. The two colliding vectors can be determined, by inspection of the ANF, to be 1111 and 0101.

A consequence of the above discussion for a general preimage-finding algorithm is that it will not be sufficient to concentrate on a subset of the output bits that numbering less than $\approx \omega$ to obtain a preimage: at least $\approx \omega$ output bits must be considered.

9.1 Scalability

To understand the scalability characteristics of various reordering heuristics, a simple experiment was performed. For a particular number of input bits n, starting from $n = 1$, a SHA-1 BDD representation was created using each of the nineteen heuristics. The amount of time that it took to apply each of the logical operations $\{\wedge, \vee, \oplus\}$ was recorded. The $\neg$ operation for BDDs is implemented in CUDD as a simple complement of a pointer's least significant bit, and is therefore very fast, non-recursive, and guaranteed to create no new nodes; no statistics were gathered for this operation. A six-figure statistical summary (minimum, lower quartile, median, geometric mean, upper quartile, maximum) of the recorded times, per-operation and across all operations, was generated. Furthermore, the number of nodes in the final SHA-1 representation as recorded to ensure that the heuristics resulted in reasonable representations. The number of input bits n was increased, and the same procedure was repeated again; however, if the total time taken to generate a SHA-1 representation using a heuristic exceeded two minutes, then that heuristic was excluded from consideration for future values of n. The experiment was considered to be complete when no usable reordering heuristic remained.

Figure 9.6 plots the sizes of the resulting BDDs for each number of inputs. It can readily be seen that the difference in size is entirely negligible, no matter what reordering method is used; the data points overlap in all cases. Note that the y-axis is logarithmic, and the line (from $n = 3$ onwards) is reasonably straight. This allows us to approximate the size of a BDD representation of SHA-1, based on the gathered data, to be $16 \cdot 2^{1.02n}$, for an n-bit input.

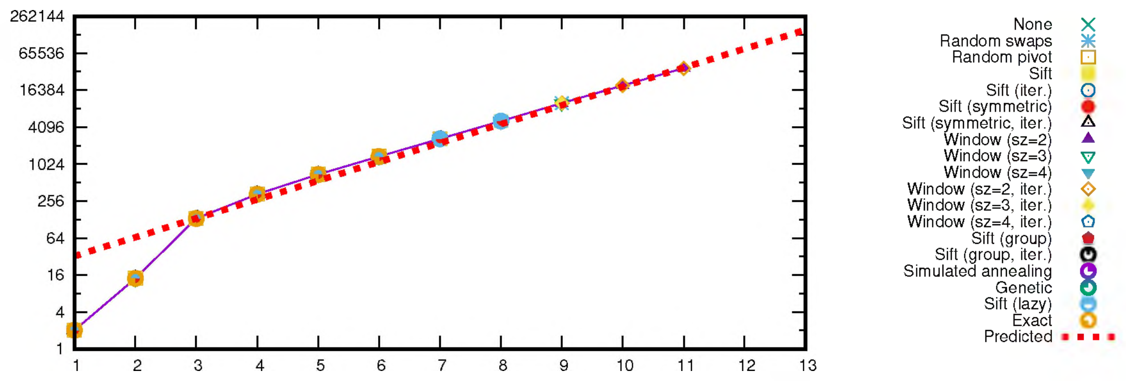

Figure 9.6: Size of SHA-1 BDD representation under different reordering heuristics

Table 9.1: BDD orderings and related size variance

Input bits	Heuristics	Unique orderings	Min. size	Max. size	Size variance
2	19	1	14	14	0%
3	19	1	133	133	0%
4	18	5	330	334	1.2%
5	18	8	695	696	0.1%
6	16	6	1351	1361	0.7%
7	15	8	2633	2641	0.3%
8	10	4	5136	5136	0.0%
9	5	2	10068	10071	0.0%
10	2	1	19612	19612	0.0%
11	2	1	37464	37464	0.0%

It was initially thought that the similar sizes were due to the quality of the reordering heuristics, but further investigation proved this hypothesis to be incorrect. Table 9.1 shows that while all heuristics converge on a set ordering when very few input bits ($n \leq 3$) or very few heuristics ($n < 5$) are being used, there are multiple different competing orderings for the majority of the runs seen. Nor were these permutations only slightly different: for example, two 7-bit orderings were 0-1-2-3-4-5-6 and 5-1-0-4-3-6-2, with the resulting sizes being 2633 and 2634 respectively. This is a very surprising result, given that the existing literature makes it clear that the ordering of variables typically results in very different sizes of BDD (Bryant, 1992; Bollig and Wegener, 1996; Krause, Savický, and Wegener, 1999; Sieling, 2002). However, the result does make some sense in the light of Section 5.2, which implies that all variables are equally important.

The data collected on the time taken by various reordering heuristics is largely irrelevant if all heuristics result in equally-bad outcomes. If all are equally bad then the best solution is to simply not apply *any* reordering heuristic at all: any heuristic has a non-zero cost and not applying any heuristic is the only solution that adds no cost at all.

Bryant (1992, p. 6) separates functions into three typical classes: *symmetric*, *integer addition*, and *integer multiplication*. The last class is the worst, with the BDD representation invariably requiring an exponential number of bits. Given the failure of all the attempted heuristics to find a non-exponential representation for any set of input bits, it is likely that SHA-1 happens to fall into this class. Wegener (1994, p. 368) defines the concept of *sensitivity* as "the quotient of the size of a reduced OBDD for [a function] f with respect to a worst ordering of the variables and the size of a reduced OBDD for f with respect to an optimal ordering of variables", and states that symmetric functions have a sensitivity of 1. It appears that SHA-1 shares this sensitivity, but is not symmetric since no heuristic was able to find a non-exponential representation.

An argument could be made that the heuristics are at fault, and that a non-exponential BDD representation is possible. To investigate this, three heuristics which utilised randomness were selected from the available set of CUDD heuristics, under the assumption that a random ordering is more likely to happen upon a "better" ordering (if one exists). These heuristics were examined in detail, informed by the official CUDD documentation (Somenzi, 2015) and by an examination of the CUDD 3.0.0 source code.

The RandomSwaps heuristic (called `CUDD_REORDER_RANDOM` in CUDD) randomly chooses n pairs of variables, where n is the number of inputs, and swaps the order of adjacent variables between the pairs. The most reduced order is the one which is used.

Example 9.4. *RandomSwaps heuristic.* Assume that a 5-variable BDD has the ordering (a, b, c, d, e). A random swap may choose the variables b and e to swap. This occurs by enacting the following adjacent swaps: $(b \Leftrightarrow c)$, $(d \Leftrightarrow e)$, $(b \Leftrightarrow e)$. At the end of the process, there are three *moves* which took place; the size of the tree after each move is retained, and the tree with the smallest size becomes the new BDD. Since there are five variables, this process is repeated five times, with the swapped variables being randomly chosen each time.

The RandomPivot heuristic (called `CUDD_REORDER_RANDOM_PIVOT` in CUDD) is similar, but chooses the variables to swap more deterministically. The variable which has the largest number of nodes is selected as a target; in case of a tie, the tied variable closest to the root becomes the target. The first variable to swap is chosen randomly from variables closer to the root than the target, and the second is chosen randomly from variables further away from the root than the target. If there are no variables closer to the root than the target, or further away from the root than the target, then the target is chosen as the appropriate variable to swap. The swapping sequence (as detailed above) occurs using the chosen variables, with the smallest size being selected. The RandomPivot heuristic thus ensures that the layer that takes up the most space is targeted.

The Genetic heuristic (called `CUDD_REORDER_GENETIC` in CUDD) uses a genetic algorithm to attempt to find a better order, inspired by the work of Drechsler, Becker, and Gockel (1996). It uses a deterministic heuristic to ensure that at least one "reasonable" order exists in a population, and then proceeds to randomly generate other members of the population. Crossover in the algorithm is implemented by a Partially Matched Crossover operation, which attempts to "construct the children by choosing the part between the cut positions from one parent and preserve the position and order of as many variables as possible from the second parent". The "best" children, as determined by size, are chosen for use in subsequent generations.

Neither the random nor the deterministic heuristics achieve better than exponential representations. This does not rule out the possibility, but it does make it much more unlikely that a non-exponential representation exists.

A binary decision diagram can provide a compact representation for many functions, but the SHA-1 compression function is not among this set. Operations performed on a BDD are reasonably fast, and the representation makes it easy to find a preimage. However, a set of 64 inputs would require $\approx 2^{64}$ nodes to represent; this makes it infeasible to use for the purposes of this work.

9.2 BDD Variants

As mentioned in the introduction of this chapter, there are many different variants of BDD. To over-simplify somewhat, a BDD variant (hereafter *DD) does two things: it makes a decision at each node to go "left" or "right" (or "high"/"low", "then"/"else", etc), and it ends up at a particular constant value when it runs out of non-terminal children. Most *DDs, such as Algebraic Decision Diagrams (ADD) (Bahar, Frohm, Gaona, Hachtel, Macii, Pardo, and Somenzi, 1997) or Biconditional Binary Decision Diagrams (BBDD) (Amarú, Gaillardon, and De Micheli, 2013), change one (or, in more extreme variants, both) of these things. For example, an ADD uses a wider range of terminal values, and a BBDD makes decisions using the biconditional expansion $f(v_0, v_1, ..., v_n) =$

$((v_0 \oplus v_1) \wedge f(\neg v_1, v_1, ..., v_n)) \vee (\neg(v_0 \oplus v_1) \wedge f(v_1, v_1, ..., v_n))$ instead of Boole's expansion theorem. In this way, and by complementary changes to the reduction rules, a variant can support more scenarios and/or increase the scalability and applicability of the *DD.

Although the improvement is welcome, it does not fundamentally change the difficulty of representing SHA-1 using a *DD. Amarú *et al.* (2013), for example, claim a reduction in the size of a ROBDD of between 28-50%. Assuming that a 64-input BBDD were to be created, and a reduction of 50% were to be achieved, the diagram would still require $\approx 2^{63}$ nodes to be represented. A more fundamental change of the representation is required, and this subsection therefore examines one of the most unusual *DDs that may be applicable to the problem: zero-suppressed BDDs.

A zero-suppressed binary decision diagram (ZBDD) (Minato, 1993) is a specialized *DD which represents *combination sets* instead of bits. A combination set expresses a set of solutions to a combinatorial problem. Technically speaking, any BDD can represent a combination set: the transitions that lead to a 1-terminal are the transitions that make the function represented by the BDD (called the *characteristic function*) true, and the ones that lead to a 0-terminal make the characteristic function false.

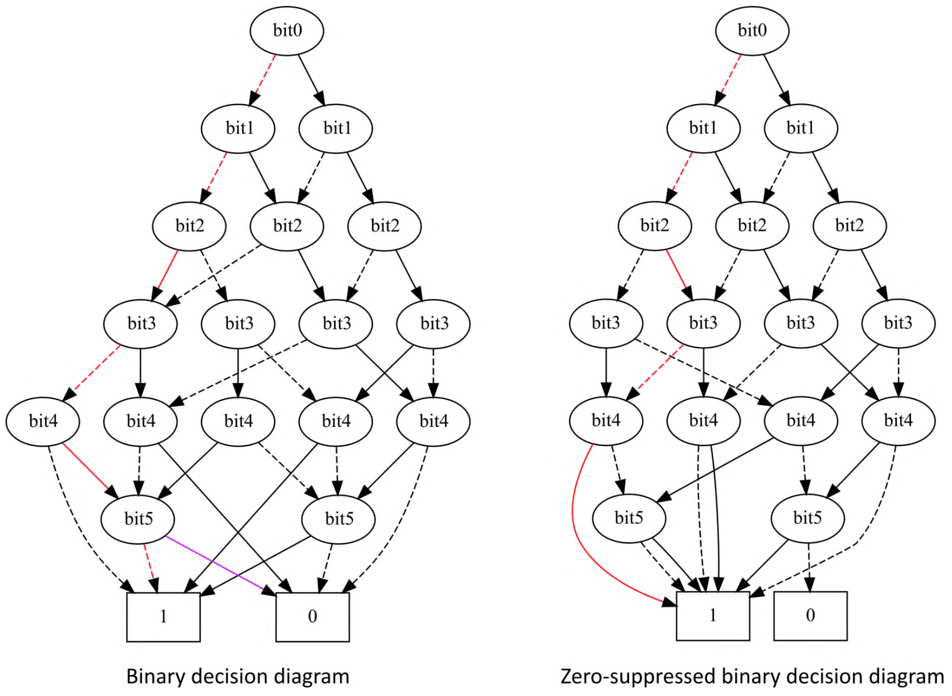

Figure 9.7: Example: BDD representation of carry calculation

A ZBDD replaces the second reduction rule of a BDD with the rule *Remove nodes with a "then" child that leads to 0, rerouting the node's input to the "else" child.* As a result, the interpretation of the diagram changes: when a node is not present at a particular level, it is *assumed to be zero-valued for the terminal value to be 1.* This means that a diagram does not need to explicitly record 0-valued nodes: it is "zero-suppressed".

Consider a combinatorial problem such as the "carry" problem described in Section 4.2.1. In that problem, the v_0 carry-value is set if exactly 1, 2, 5, or 6 bits (assuming $k = 1$) happen to be set. Note that the combination of the bits matters, but the order does not. There are then $\binom{6}{1} + \binom{6}{2} + \binom{6}{5} + \binom{6}{6} = 28$ possible ways for v_0 to equal 1. Figure 9.7 shows what this would look like in a ROBDD (*sans* complement edges) and a ZBDD, both of which represent each bit as a node.

Example 9.5. *Difference in interpretation of BDD and ZBDD.* Consider the red path in Figure 9.7, which represents the vector 001010 and should end at the 1-terminal. The paths between the nodes form a combination set, mapping out the combinations where the value will be 0 or 1. In the BDD, all paths are explicit. In the ZBDD, the final path is *implicit*: the last bit of the vector is implicitly 0 for the terminal to be 1. Now consider the vector 001011, which should end at the 0-terminal. In the BDD, the path taken diverges onto the magenta path that leads to 0. In the ZBDD, the same path is taken as before; however, since the *bit5* value which was assumed to be 0-valued is 1, the result is 0.

One consequence of this is that while the paths of the BDD shown in the figure can handle 6-bit numbers, the ZBDD can handle n-bit numbers and will follow the implicit 0-terminal path for 001011, 0001011, 1001010, and so on. Another consequence is that the number of "then" paths is exactly equal to the smallest number of set combinations that lead to the 1-value (in this case, 15); this will always be true of a ZBDD (Minato, 1993).

A ZBDD can be seen as representing all of the set combinations that can lead to the characteristic function being true. As a result, ZBDDs cannot be used with the usual boolean manipulations described by Bryant (1986) and others. Set operations $\{\cap, \cup, \setminus\}$ must be used instead. The advantage of this, however, is that ZBDDs are excellent for representing *sparse* combinatorial problems which have very few 1-valued solutions. Unfortunately, SHA-1 representation is not such a problem; the balance (Bellare and Kohno, 2004) and SAC characteristics (see Section 5.2) of SHA-1 ensure that every output bit has a very large potential number of inputs that may cause it to be 1-valued. When restricted to a particular preimage, the situation changes drastically; however, as Figure 9.4 shows, a BDD representation of a preimage is very small, and little would be gained by a ZBDD representation.

9.3 Intermediate representation size reduction

Bradley and Davies (1998) describes a technique to lazily construct a BDD in a top-down fashion, using a maximum space complexity of $\mathcal{O}(|t|)$ where $|t|$ is the number of nodes in the final tree. Since the final tree is very small, this approach is very attractive. However, lazy construction takes place from the root downwards, and terminal nodes are therefore not available without being explicitly created via a traversal down the tree. This means that the path that leads to the 1 terminal is unclear, and finding it involves an extensive search of the state space; the nodes that lead to 0 are not shared or eliminated in the un-decomposed lazy representation. The result is a typical time-space trade-off that is comparable to the trade-off made by SAT solvers.

One way of investigating the size increase in more detail is by looking at the per-round sizes of generated BDDs. The calculation of each individual a_i^r value can be represented using six variables, assuming that f is expanded and w is not expanded: a, b, c, d, e, and w. If calculation proceeds from $i = 26$ and continues downwards, wrapping around towards $i = 27$, then $v0$ and $v1$ can be represented in terms of previously-defined a variables. Each a-BDD can be referenced separately.

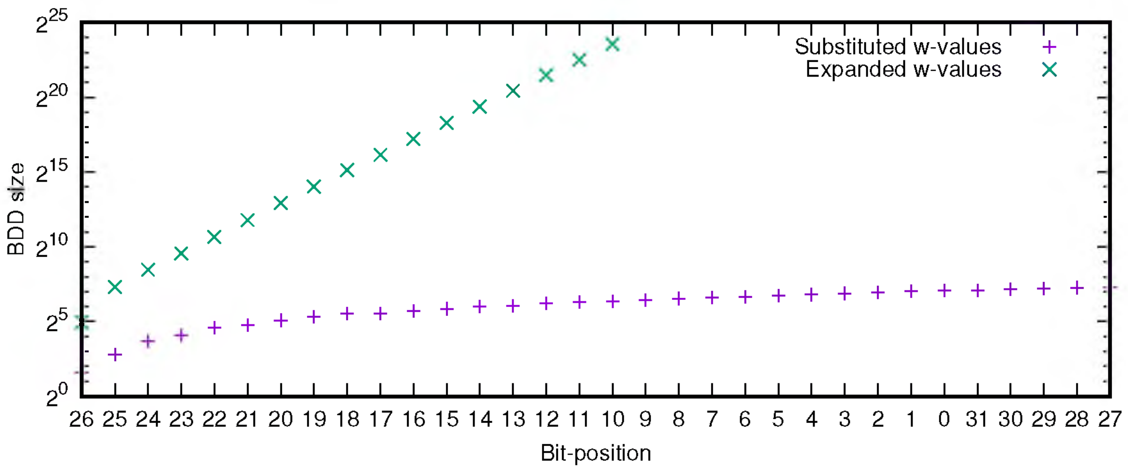

Figure 9.8: BDD size increase (substituted w vs expanded w)

Although it is tempting to expand the w term, the consequences of doing so are dire. Figure 9.8 demonstrates the difference in BDD size for 128 bits of input. As each separate a-BDD is created, it is combined with the previously-created a-BDD. This should, theoretically, reduce the size of the BDD since the solution space is being increasingly constrained. The reduction is not sufficient to overcome the increase in size due to the number of variables, and this points to the importance of keeping the number of variables to an absolute minimum when BDDs are used.

Let the calculation begin from $r = 80$ and move towards $r = 1$. The initial BDD calculated, a^{26}_{80}, is obviously the smallest and comes in at 11 nodes. Subsequent a-BDDs increase in size since they must make use of carry calculations. The maximum size of the BDD for each row is not exorbitant, coming in at ≈ 1245 nodes when $i = 27$. However, combining BDDs between rows involves an increase in the number of variables, and a correspondingly large increase in the size of BDDs; the situation quickly devolves into one similar to that shown by Figure 9.8.

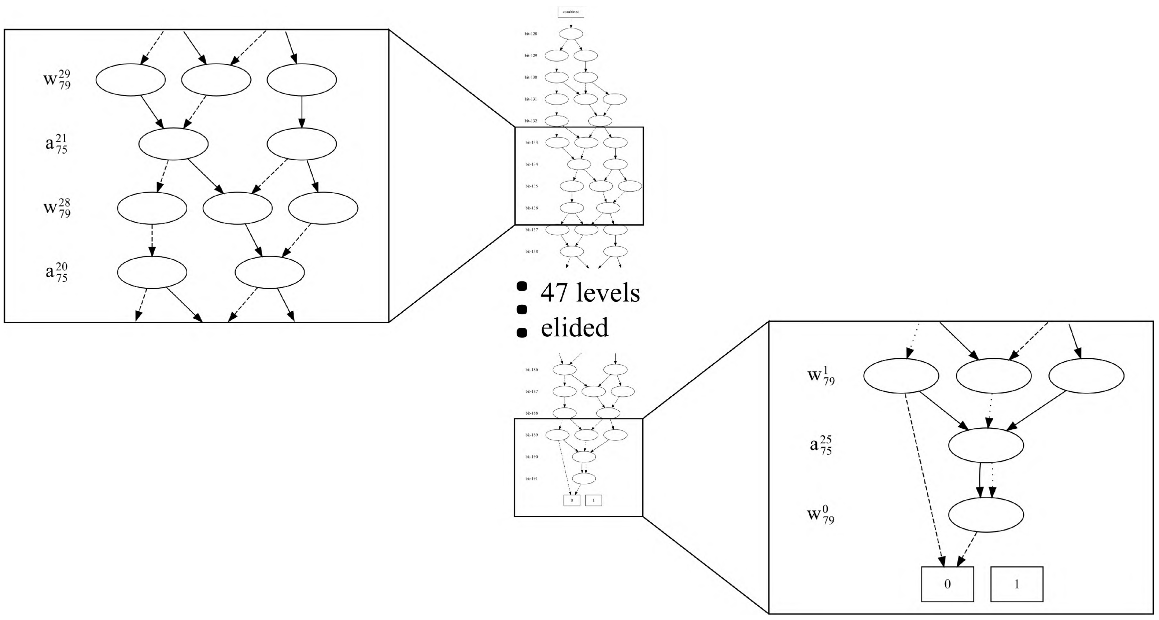

Figure 9.9: Pattern of divergence/convergence (left), solution to w^0_{79} (right)

Since it is possible to start with the output a-BDDs, it is also possible to combine them using Equation 4.10. The resulting graph has ≈ 155 nodes and is deep, with divergence and convergence between a_{75} and w nodes. This is expected since the $a_{\geq 76}$ BDDs have fixed values. The very final w variable, w^0_{79}, has a definite solution since there can be no possible carry from the related a^{27}_{80} variable. Note that the w variables are expansion-words, not data-words, and therefore represent sets of data-words. This is necessary to be able to scale up to a larger number of inputs. Figure 9.9 is an example BDD that shows the pattern of divergence and convergence, highlights the w^0_{79} solution, and gives some indication of the depth of the tree. Interestingly, this BDD also shows that both w^0_{79} and a^{25}_{75} are irrelevant, as long as a particular path is followed that leads to the leftmost node of w^1_{79}. However, there is no guarantee that this path would be taken — and, in fact, the SAC characteristics of SHA-1 make it unlikely for $\omega < 160$.

9.4 Summary

ROBDDs are a versatile representation, but this research was unable to discover any way to reduce their intermediate size and *DD variants are unlikely to offer solutions. The SHA-1 preimage problem cannot be efficiently encoded without such a reduction. The experiments conducted illustrate the importance of each output bit (up to $\approx \omega$ output bits) in finding a preimage, imply the multiplicative complexity of the SHA-1 compression function, and reestablish the inability of heuristics to overcome the fundamental difficulty in efficient representation.

Chapter 10

And-Inverter Graphs

Representations examined up to this point share the trait of being difficult to optimize in a *local* context. In other words, a short segment of the function cannot be "lifted" from the rest, optimized, and then dropped back into the global picture; instead optimization must take place globally or not at all. This chapter examines a representation that is both scalable and very amenable to local optimizations: the and-inverter graph (AIG). The representation is first described and discussed, and a powerful state-of-the-art tool for manipulating the representation is introduced. Various local optimizations are described and applied to the SHA-1 preimage problem. A more focused optimization is then attempted, and the results of this are presented.

An and-inverter graph is a directed acyclic graph (DAG) which represents a boolean formula using only $\wedge$ and $\neg$ operations (Kuehlmann, Ganai, and Paruthi, 2001). The set $\{\wedge, \neg\}$ is functionally complete.

- $a \wedge b \mapsto a \wedge b$
- $a \vee b \mapsto \neg(\neg a \wedge \neg b)$
- $\neg a \mapsto \neg a$
- $a \oplus b \mapsto \neg(a \wedge b) \wedge \neg(\neg a \wedge \neg b)$

Most nodes in the graph have two inputs (called *fan-ins*) and one output (called a *fan-out*); each fan-in may be inverted such that 'true' becomes 'false' and vice-versa. Given fan-ins a and b, the fan-out is $a \wedge b$. If a node does not have any fan-ins then it is called a *primary* *input*. Each variable is represented as a primary input. AIG representations are very useful for electronics engineering work, where the area of a circuit and the delay between circuit input and output are important. The number of nodes in an AIG is correlated with the area of the graph, and the maximum number of nodes between primary input(s) and final fan-outs is correlated with the delay. In other representations, these would be more commonly be called the *size* and *depth* of a graph.

An AIG represents formulae in a compact and convenient way: the representation of a formula correlates linearly with the number of variables, and nodes in an AIG are cheap to refer to and reuse (Mishchenko, Chatterjee, and Brayton, 2006). Furthermore, interactive logic synthesis software such as ABC[1] is able to minimise an AIG heuristically via DAG-aware transforms (Berkeley Logic Synthesis and Verification Group, 2016).

Although AIGs have existed for decades, it is only within the past decade that they have become important as tools for logic re-synthesis and minimisation. Most other representations rely on a global view of the entire function to minimise or analyse the function, and this can lead to complex functions and/or many variables being a source of scalability problems. By contrast, AIGs are typically manipulated in terms of local transformations, and therefore do not suffer from these issues. An AIG representation of a function is not canonical: two identical functions may have different AIG representations. This is both an advantage and a disadvantage: AIGs can be modified more easily and flexibly than many other representations, but verifying that two AIG representations have the same semantics requires additional (and potentially expensive) computation.

Another advantage of AIGs is the ability to easily use *structural hashing*. Structural hashing ensures that elements of the graph are not duplicated by checking the inputs of a graph. Inversion of inputs is considered during the check. If two nodes have the same inputs, then one may be replaced entirely by the other.

At runtime, each node in an AIG can be represented as a pair of 32-bit unsigned integers (or, equivalently, a single 64-bit unsigned integer). The graph itself consists of multiple nodes, and is represented by a list of nodes. This design follows on directly from the work of Biere (2007) on the AIGER format, which is the default input and output format of ABC. The main points of this representation are as follows:

1. Each index in the list represents the output of a node.
2. The constants true and false are 1 and 0 respectively.

[1]ABC version 1.01 (compiled July 7 2016)

3. Inputs are numbered sequentially from 1 onwards.
4. An even integer n represents the input $\frac{n}{2}$, and an odd integer n represents the inverted input $\frac{n-1}{2}$.
5. The number of primary inputs is specified explicitly.

Given the above rules, the value stored at a particular index of a list can be unambiguously interpreted as either a fan-out or a primary input.

Example 10.1. *AIGER format.* The compact in-memory AIG representation and the AIGER format are closely related. The following example of an AIGER file represents the function $\neg(a \oplus b)$, and may help to understand this representation more intuitively.

```
aag 6 2 0 1 3
2
4
11
6 2 4
8 3 5
10 7 9
```

This example is also represented visually as Figure 10.1. The file begins with a format identifier string `aag`. This is followed by five variables. For our purposes, the first and third variables are unimportant. The second variable (`2`) is the number of primary inputs; the fourth (`1`) is the number of outputs; and the last (`3`) is the number of non-primary-input nodes in the graph.

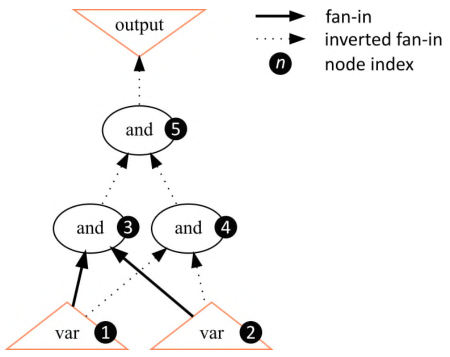

Figure 10.1: And-inverter graph example (derived from ABC's dotfile output)

The next two lines declare the two inputs (`2` and `4`), which are positive and therefore even. Inputs are considered to be nodes in the AIG, and therefore take up space in the list. The index of the first input is $\frac{2}{2} = 1$, and the index of the second is $\frac{4}{2} = 2$. The output (`11`) follows the input, and indicates the index $\frac{11-1}{2} = 5$ from which the output value can be read. Since `11` is an odd value, the actual output is the inverted value found at index 5.

The next three lines have the format $[idx]$ $[i0]$ $[i1]$, where idx is the doubled index of the node in the list, and $i0$ and $i1$ are the inputs to that node. Therefore, `6 2 4` will perform an AND of the values at $\frac{2}{2} = 1$ and $\frac{4}{2} = 2$, storing the result at $\frac{6}{2} = 3$; and `10 7 9` performs an AND of the inverted value at $\frac{7-1}{2} = 3$ and the inverted value at $\frac{9-1}{2} = 4$, storing the result at $\frac{10}{2} = 5$. The AIG that is thus described has two inputs, and performs an XNOR of those inputs via the logic $f(i_1, i_2) = \neg(\neg(i_1 \wedge i_2) \wedge \neg(\neg i_1 \wedge \neg i_2))$.

It requires approximately 221,500 nodes and a graph depth of approximately 18,140 to represent a SHA-1 calculation. The computational cost of generating an AIG representation is minimal since the addition of a node involves no changes to any other node. Evaluation of the representation, given particular input values, is likewise quick and easy: each node needs to be evaluated only once, and the result can be cached for future use. The cost of evaluation is therefore $\mathcal{O}(n)$.

An AIG with fewer nodes has fewer logical operations separating the final fan-outs and the primary inputs, and is consequently easier to work with to find a preimage. Fortunately, one of the primary uses of an AIG is to reduce the area of a circuit which is represented as an AIG, and this goal is often achieved by reducing the number of nodes in the AIG. Therefore, existing state-of-the-art tools such as ABC contain many algorithms for reducing the node count, including

Rewriting This is "a fast greedy algorithm for minimizing the AIG size by iteratively selecting AIG subgraphs rooted at a node and replacing them with smaller precomputed subgraphs, while preserving the functionality of the root node" (Mishchenko *et al.*, 2006; Mishchenko and Brayton, 2006). The pre-computed subgraphs, located in a statically-compiled look-up table (LUT), are exhaustive and have 4 inputs each;

rewriting should therefore always be able to find the smallest possible representation of a 4-input AIG. If the best alternative representation has the same number of nodes, then it is considered to be a "zero-cost" replacement; the user decides, using a `-z` switch, whether zero-cost replacements are substituted.

Refactoring "A *cut* C of node n is a set of nodes of the network, called *leaves*, such that each path from a [primary input] to n passes through at least one leaf. Node n is called *root* of cut C. The cut *size* is the number of its leaves." (Mishchenko and Brayton, 2006). Refactoring attempts to find, using various heuristics, a large cut for each node which can be replaced by a smaller, or equally-sized, set of nodes. Similarly to rewriting, zero-cost replacements can be used at the user's option by specifying a `-z` switch.

Collapsing This effectively transforms an AIG into a BDD; however, all the disadvantages of BDD representation remain, and larger functions may be effectively unrepresentable as a BDD.

Rebalancing This is the process of reducing the number of levels between inputs and outputs — otherwise known as the depth of the graph — without duplicating logic. It is possible only because AIG representation is non-canonical.

FRAIGing Using a SAT-solver and simulation, this set of functionality attempts to convert an AIG into a Functionally Reduced And-Inverter Graph (FRAIG) (Mishchenko, Chatterjee, Jiang, and Brayton, 2005). FRAIGing attempts to identify sets of nodes which perform the same function in an AIG, and remove redundant sets of nodes which are no longer needed.

Perhaps more interestingly, the choices made during construction of a FRAIG can be accumulated for later examination and reevaluation. This allows future transformations to examine multiple possible representations of functionality and choose between them.

Each algorithm is used on an AIG by invoking the appropriately-named ABC command. Other algorithms included in ABC, such as *redundancy removal* and removal of extraneous nodes (*cleanup*), are largely not applicable to SHA-1 since there is no obviously-redundant or extraneous logic in the algorithm (see Chapter 4). The above algorithms are often interleaved and iterated, with each transformation of the AIG giving rise to new opportunities for additional improvement. They are also typically used in conjunction with algorithms which affect the graph structure, but do not decrease the number of nodes in the graph. An example of such an algorithm is *balance*, which attempts to reorganise the AIG to reduce the number of nodes between primary-inputs and final outputs. In doing so, additional opportunities for subsequent area improvement may be created.

In the context of SHA-1, most of the local transformations described act to decrease the amount of diffusion that is present in the algorithm. It is particularly noteworthy that in AIG form, all internal state (i.e. a-variables) is last since the notion of individual variables is replaced by the abstraction of $\wedge$ gates and inverters — and these are represented exclusively in terms of their structure, and without names.

ABC comes with script aliases which can execute commands sequentially, and additional script aliases can be defined very easily. The default scripts have been shown to reduce the size of AIG representations of problems from various domains (Mishchenko *et al.*, 2006). For example, one of the more useful scripts, `resyn`, executes the commands `balance; rewrite; rewrite -z; balance; rewrite -z; balance`. This takes approximately a minute to run, but reduces the size needed to represent a 447-input single-chunk SHA-1 calculation from $\approx 221,500$ to $\approx 179,200$, and reduces the depth from $\approx 18,770$ to $\approx 18,140$. Although these figures are still too high to be useful, it is a distinct improvement at a reasonable cost and clearly demonstrates the utility of AIG representation and transformations. Additional time spent using rewriting, refactoring, rebalancing, and FRAIGing reduces the size to $\approx 165,900$ and the depth to $\approx 16,630$.

10.1 Pushing variables up the graph

Choose a non-input, non-output node in an AIG. It is connected to at least one input and at least one output, perhaps via other nodes. In what follows, the paths leading towards outputs at the "top" are considered to be "up", and the paths leading towards inputs at the "bottom" are considered to be "down".

It is possible to restructure the AIG to push particular variables up or down the graph; the case of pushing up will be discussed first. Assume that, is shown by Equation 4.10, the graph has a single output. By pushing a particular variable up to the top of the graph, the only node that refers to that variable ends up being the topmost variable in the graph. This effectively partitions the AIG into two sub-graphs, one of which represents the function when the variable is true, and the other which represents the function when the variable is false. In essence, this partitioning is the same as that accomplished by Boole's expansion theorem.

Boole (1854) showed that, given a boolean function $f(v_0, v_1, ..., v_n) = 0$, the following identity holds:

$$f(v_0, v_1, ..., v_n) = f(0, v_1, ..., v_n) \wedge f(1, v_1, ..., v_n) = 0$$

Chapter 9 briefly discussed this in the context of BDD decision functions. For the purposes of this chapter, it is significant to understand the *expansion* that occurs: the remaining terms in the original equation are doubled so that a single variable can be removed. While this would ordinarily be a bad idea, it is hoped that the structural hashing of nodes in an AIG would compensate for this doubling, and that local rewriting and refactoring after each "push" would be able to find more opportunities for minimization when fewer variables exist in each subgraph.

Example 10.2. *Pushing a variable up (simple case).* Consider the formula $a \wedge b \wedge c$, where a is the variable that should be pushed up. Figure 10.2 shows the necessary transformation.

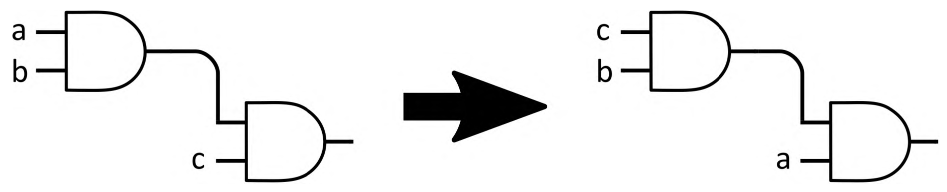

Figure 10.2: Example: simple push upwards

Although the literals b and c are used in this example, the non-a inputs can just as easily be non-primary nodes.

In the absence of clause negation, the process of pushing a variable upwards is simple (see Example 10.2): a transposition of inputs is all that is required.

Example 10.3. *Pushing a variable up (more complex case).* Consider the formula $\neg(a \wedge b) \wedge c \wedge d$, where a is the variable to be pushed up. Figure 10.3 shows the necessary transformation.

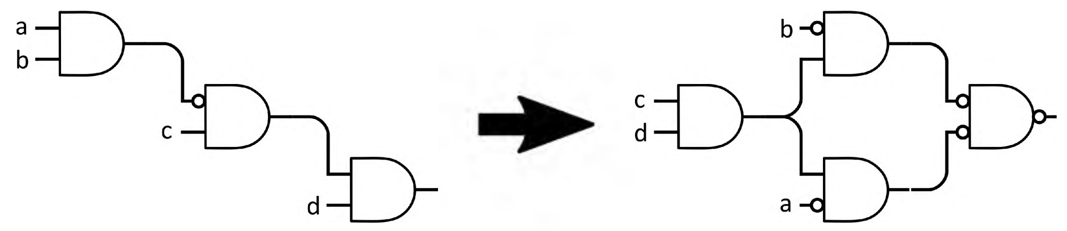

Figure 10.3: Example: more complex push upwards

Notable features of this transformation are that an additional node in the graph is necessary; it is necessary to operate on a larger cut of the graph in order to achieve the transformation; and the advance is limited. If the original node that takes a d input was not present, it would be impossible to advance the a node by a level.

Other examples can be constructed for additional cases, but this is a very time-consuming and error-prone process. Let the desired variable be α. The following steps can then be used to automate such construction:

1. **Identify a cut which includes α on the lowest level**. A cut comprising a maximum of 7 nodes and 8 variables across 3 levels of the AIG was chosen (see Figure 10.4). Some non-root nodes may be primary inputs and the size of the cut may therefore be less than 7.

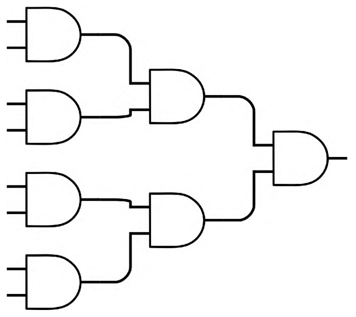

Figure 10.4: Illustration of chosen cut

2. **Assign a variable for each non-constant, unique terminal of the cut**. A terminal is an input to the cut.

3. **Generate a truth-table for the cut**. Since there are a maximum of 8 variables for the chosen cut, the maximum size of the truth table is $2^8 = 256$.

4. **Split the truth table based on the value of α.** This is trivially done if, in the previous step, the α variable was used as the "leftmost" variable in the truth table. The resulting truth tables will contain all variables *except* for α.

5. **Generate a minimal representation.** Non-heuristic methods such as Karnaugh maps (Karnaugh, 1953) or the Quine-McCluskey algorithm (McCluskey, 1956) and Petrick's method (Petrick, 1956) are reasonably efficient for small numbers of variables. Using the cut shown in Figure 10.4 and after the table split, the maximum number of variables to be handled is 7, which is well within the bounds of computational feasibility.

6. **Convert the minimal representations into replacement cuts.** One of the representations gives the function value when $\alpha = 0$ and the other gives the function value when $\alpha = 1$. The variables extracted in step (2) are reintegrated into the cuts in this step.

7. **Join the representations appropriately, with α at the top.** The identity $x \vee y = \neg(\neg x \wedge \neg y)$ can be used to give the correct gates. An implementation must be sure to catch all the "edge cases", such as when one of the representations is always true or always false, since these edge cases could result in a simpler representation.

An iterative approach that keeps pushing a variable upwards will, with a cost that is linear with reference to the size of the AIG, succeed in pushing the desired variable to the top of the graph. The most expensive step is (5), but this can be greatly reduced by maintaining a cache of minimal representations that is indexed by truth table; conveniently, step (5) is also one in which the truth table matters and the variables do not.

Optimization opportunities arise during the rewriting process when duplicated variables are found in a cut. This is more likely to happen with larger cuts, but the rewriting of larger cuts is correspondingly more expensive since there are many more possible variations to consider.

Figure 10.5 shows a representative sample of the results of this approach. The overall algorithm works in a single pass over the AIG and is therefore linearly related to the size, but is dominated by the exponential time taken by the Quine-McCluskey algorithm that is used. It can readily be seen that the size of the AIG almost doubles each time despite the use of minimization techniques. As expected, the situation changes dramatically as soon as the remaining number of variables fall to 4 or 5 since, due to the built-in lookup

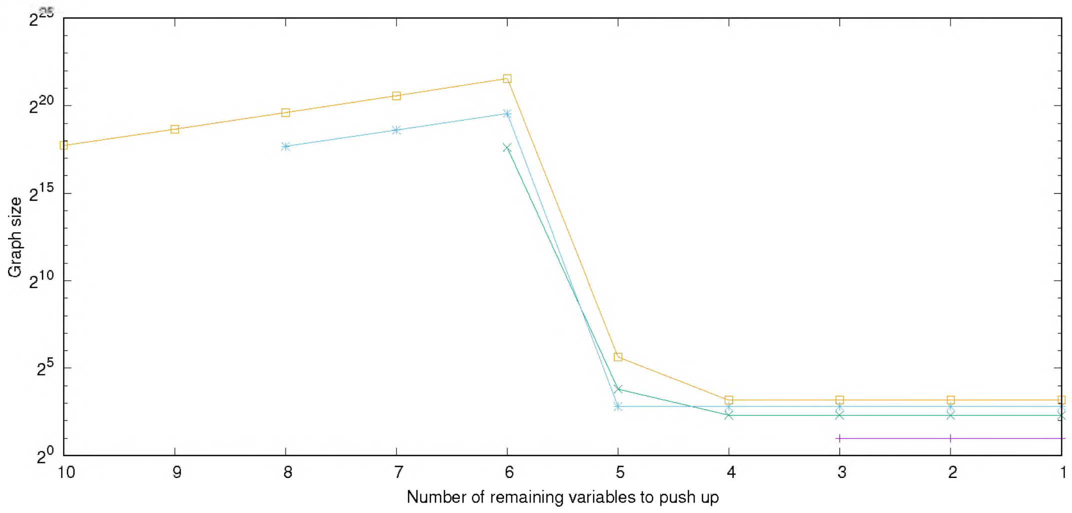

Figure 10.5: AIG doubling

tables, any graph with 4 or fewer variables can be immediately reduced to a minimal form — in other words, a preimage.

Several different minimization techniques were attempted, including FRAIGing and lookup table creation. Unfortunately, these techniques did not improve the size of the AIG by an appreciable amount and, furthermore, they led to runtimes that were more difficult to predict. Rewriting, refactoring, and rebalancing were therefore exclusively used via the `resyn2` alias, which expands to the following sequence of commands: `balance`; `rewrite`; `refactor`; `balance`; `rewrite`; `rewrite -z`; `balance`; `refactor -z`; `rewrite -z`; `balance`.

As an optimization, the doubled graph can be split into two branches and each of these can be processed independently. The overall CPU time spent remains the same since both graphs must ultimately be processed, but the time can be split across multiple processors and/or machines. Figure 10.6 demonstrates that the majority of the time spent is taken up by rewriting and refactoring; therefore, another possibility is to push a number of variables upwards before incurring the expense of rewriting and refactoring. A classic time-space trade-off is incurred in the process.

No optimization changes the fundamental issue: an AIG representation where $\omega \geq 6$ cannot be minimized to a degree that offsets the space increases incurred by pushing variables upwards.

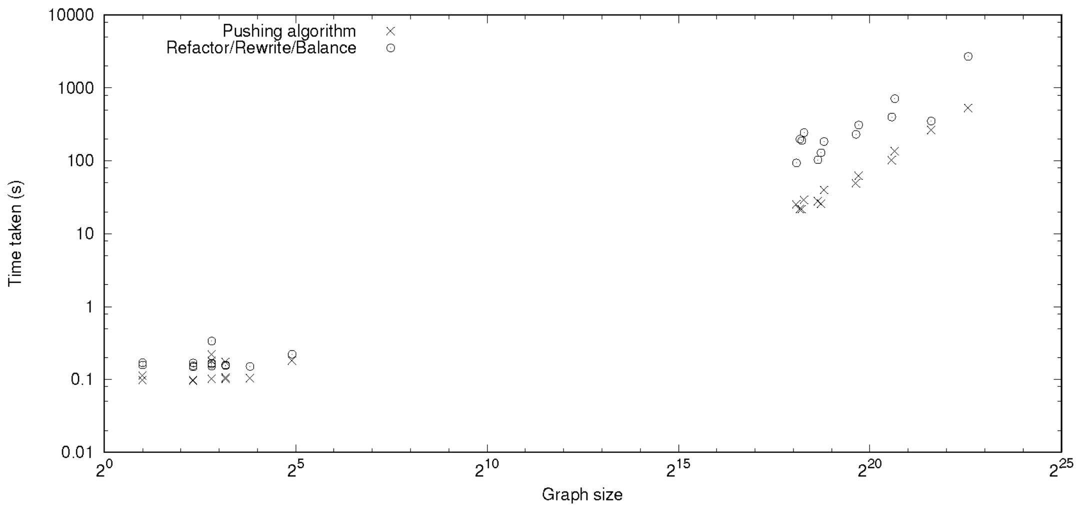

Figure 10.6: Performance: pushing-up algorithm vs. rewrite/refactor/balance

10.2 Summary

An AIG is a versatile, compact, and scalable representation that is well-suited to expressing any boolean formula. The strength of AIGs is their flexibility: nodes can be added inexpensively, functions can be represented in many different ways, and DAG-aware rewriting of the graph is a low-cost operation that can have larger cumulative effects. However, this flexibility is also the weakness of AIGs since it is difficult to arrive at the simplest representation of a function when a "function" cannot be recognized as such. The ABC package partially addresses this through the use of lookup tables during rewriting, and can always recognize the simplest representation of any 4-input function via a look-up table. However, this approach does not scale: a 4-input LUT must represent $2^{2^4} = 65,536$ functions, but a 5-input LUT to do the same for 5-input functions would need to represent $2^{2^5} = 4,294,967,296$ functions. As a representation for SHA-1, an AIG is able to scale to larger inputs and perform (limited) minimization operations. In the final analysis, however, local transformations must eventually rely on heuristics, and those heuristics are (at present) unable to reduce the SHA-1 preimage problem to its most minimal form.

The fact that it is difficult to reduce the size and depth of the graph, despite structural hashing, demonstrates that there is inherent complexity in the diffusion. By contrast, the $a..e$ variables of the traditional formulation (see Section 2.4) are merely renamed a variables; the fact that they have different names in the specification does not lead to any increased inherent complexity, though it may lead to the appearance of such.

Chapter 11

Constraint Satisfaction Problem

This chapter describes a representation that considers the SHA-1 preimage problem as a search problem. The preimage problem is encoded as a set of constraints on each variable, putting the theory described in Section 5.1 into practice. This aspect of the representation makes it unsuitable for use with with the structure described in Section 6.1. A naïve implementation provides fertile ground for discussion of the difficulties of this course, and a sophisticated implementation provides results for further analysis.

A Constraint Satisfaction Problem (CSP) must be considered as both an abstract problem to model and a practical problem to implement. At an abstract level, the problem must be modeled appropriately since the manner in which it is modeled can have a big impact on how difficult it is to solve. The difficulty of solving the problem is, in this case, expected to be directly tied to the difficulty of finding a valid partial solution — or, to phrase it another way, the difficulty of finding a partial solution which can be *shown to be valid.* On the practical side, an efficient way of propagating constraints — a process which is crucial to efficiently solving CSPs — must be implemented. The latter problem is made much easier by the existence of well-justified, efficient CSP toolkits such as Gecode[1] (see Tack (2009) for details), and will therefore not be addressed here.

Using the constraints previously described in Equation 4.8 (page 63), SHA-1 can be modeled as additions using w variables and q variables, with the remainder of the variables (a, v, f, and k) being either derived from these or constant. A q variable is particularly useful because it contains sufficient information to identify both a and v values, as well as the carry behaviour from one position to another.

[1]`http://www.gecode.org/`

Example 11.1. *The sufficiency of q and w variables.* Assume that the values for a round r are expressed entirely in terms of q and w variables, viz.

q_r-values	`574476423`**`4`**`3411344443345333256335`
w_{r-1}-values	`01100010011101001001001100010000`

The q_r^{18} value (**boldfaced** above) has been singled out for examination; however, the reasoning in this example applies to any index. The equation for q_r^{18} is

$$q_r^{18} = a_{r-1}^{23} + f_{r-1}^{23} + a_{r-5}^{16} + w_{r-1}^{i+5} + v_r^{18} + k_{r-1}^{23}$$

The k value is constant and can therefore be disregarded. The w value is known. Any a_r^i value is equivalent to q_r^i mod 2, and can therefore be calculated when q values are available. Since f values rely on a values, they are also calculable. Lastly, the v_r^{18} value is equivalent to $\frac{q_r^{19}}{2}$ (using integer division), and is 1 in the case of the example data since $q_r^{19} = 3$. Tracking q and w variables is therefore sufficient for all calculations.

This is not the only way of modeling SHA-1. On a practical level, a CSP can be defined by the 3-tuple (constraints, domains, variables), and there are often many variations of these that are possible for any given problem. The way in which a problem is modeled, and the algorithm used to find a solution, can have a very large impact on the time taken to find a solution (Rossi, van Beek, and Walsh, 2006).

Two kinds of algorithms that are used in practice to search for a solution to a CSP are *forward checking* (Brailsford, Potts, and Smith, 1999) and *arc consistency* (Régin, 2005). Both of these approaches attempt to eliminate values from domains based on an extension of a partial solution. A forward checking algorithm will tentatively make an assignment to a variable x, and check whether all other variables are consistent with that assignment. Arc consistency enforces consistency between the values of all variables such that each variable is consistent with each other variable when all constraints between variables are taken into account. An arc consistency algorithm will therefore tentatively make an assignment to a variable x, and check whether all *pairs* of variables are consistent with that assignment. This involves more checking and is therefore more expensive, but will result in more implications between variables being determined.

It is possible to use a more "traditional" formulation of binary variables, or to convert the q-variable representation to the binary domain using either the hidden-variables or dual-graphs technique as described in (Bacchus, Chen, Van Beek, and Walsh, 2002). However, Brailsford *et al.* (1999) suggest that “a few variables with large domains are in general preferable to many variables with small domains” since the former lead to a smaller search space, and they specifically state that “zero-one variables are not desirable and should be avoided if possible”.

The work of Bacchus *et al.* (2002) empirically examines the advantages and disadvantages of using binary/non-binary constraints, as well as the effect of particular translations of non-binary constraints into binary constraints. Using examples and proofs, Bacchus *et al.* (2002) examine dual-graph, hidden-variables, and non-binary formulations and show that it is possible for any formulation of a problem to be exponentially worse than other formulations — and that this is a feature of the problem itself. However, they provide no guidance about how to determine whether a particular problem will be exponentially worse when using a particular formulation.

Bacchus *et al.* (2002) do show that if a problem is arc consistent in a non-binary formulation, then it will be arc consistent when converted to a binary domain using the hidden variables technique. Furthermore, they show that the computational cost of maintaining arc consistency is never more than a polynomial factor worse when considering the non-binary and hidden-variables binary formulations. They also show that the hidden-variables formulation is never more than a polynomial factor worse than the dual-graphs formulation. As a result, there is no reason to avoid a non-binary formulation: the performance of such a model is not necessarily worse than the performance of binary formulations, and when it is worse, it is guaranteed to not be much worse.

Another reason to not model the SHA-1 CSP in binary terms is because this has been done — unsuccessfully — before, usually in the context of SAT-solving. Nossum (2012) models the SHA-1 problem domain in binary terms in order to represent it in CNF; Jovanović and Janičić (2005) shows a generic encoding for any hash function; and Nossum (2013) have submitted a tunable instance generator for SHA-1-based CNF formulae. It therefore seems worthwhile to take the path less-travelled and use a non-binary model, in the absence of any reason to avoid it.

Example 11.2. *An advantage of a non-binary model.* Consider the input `0xdeadbeef-cafeb4be`. The last five rounds of this, using the coordinate system described in Equation 4.8, are given as side-by-side q and a values.

Round	q-values	a-values
76	`44555544253441333244234344`**`2`**`44335`	`00111100011001111000010100`**`0`**`00111`
77	`45346412453323445463443366`**`2`**`54666`	`01100010011101001001001100`**`0`**`10000`
78	`57447642343411344443345333`**`2`**`56335`	`11001000101011100001101111`**`0`**`10111`
79	`44434556442544566653325355`**`2`**`55447`	`00010110000100100011101111`**`0`**`11001`
80	`44346452355455644568645222`**`1`**`33243`	`00100010111011000100001000`**`1`**`11001`

Index 26 is equivalent to index 0 for carrying purposes and has been indicated by **boldface** in the rows above. Carries occur to the left and wrap around. It can readily be seen that the q-values contain sufficient information to identify v-values, a-values, and carry behaviour.

As discussed in Section 5.1, majority is a polymorphism of SHA-1, and this means that the CSP has an n-extension property for $n \geq 2$. Since a valid partial solution should be extendable in this fashion, ensuring that a partial solution is valid is of paramount importance.

11.1 Naïve implementation

A naïve non-binary implementation utilizing arc consistency was created to explore the CSP representation. Three kinds of constraints could be applied immediately:

1. A constraint that reduces the domain of known w values such as w_r^i when $r \times 32 + i >= n$ for an n-bit input; and
2. A constraint that reduces the domain of known q values by restricting them to be all-odd or all-even, depending on the known a-values of $a_{76..80}^i$; and
3. A constraint that reduces the domain of q values, consistent with the maximum and minimum possible contributing values.

Assume that the CSP is modeled with only these three constraints. As a result, $|D(q_{80}^i)| = 2$, $\max(|D(q_{77..79}^i|) = 3$, $\max(|D(q_{76}^i|) = 4$, and $\max(|D(q_{1..75}|) = 9$. A partial solution cannot be found by manipulating these values, however, because it is always possible to manipulate earlier-round q-values to obtain a particular outcome. This is because the q_{80}^i equations are typically of the form

$$q^i_{80} \;=\; a^{i+5}_{79} \;+\; f^{i+5}_{79} \;+\; a^{i+30}_{75} \;+\; w^{i+5}_{79} \;+\; v^i_{80} \;+\; k^{i+5}_{79}$$

Typical domains associated with such an equation might be

$$\{4,6\} \;=\; \{1\} \;+\; \{1\} \;+\; \{0,1\} \;+\; \{0,1\} \;+\; \{2,3\} \;+\; \{0\}$$

The possibilities make it easy to obtain any value. This manipulation remains possible until the initial rounds are reached and fixed a-values are encountered. Therefore, it is difficult to check whether a set of values constitutes a valid partial solution without extensive effort.

Instead of starting to search for solutions from round 80, it is possible to start the search from round 1. For most of the round $|D(q_1)| = 3$, with the exception being $|D(q_1^{26})| = 2$. Although this is a larger domain than is present in round 80 (where $|D(q^i_{80})| = 2$), the domains associated with the equations in round 1 make them much less amenable to manipulation. A typical example of such domains is

$$\{2,3,4\} \;=\; \{0\} \;+\; \{0\} \;+\; \{0\} \;+\; \{0,1\} \;+\; \{2,3\} \;+\; \{0\}$$

As values are fixed for q-values closer to $i = 26$, the possible choices for other domains tends to reduce since v-values become increasingly constrained. Thus, although the number of choices is ostensibly 3, it is 2 in practice if values are fixed from $i = 26$ onwards. However, it is still very difficult to check whether a chosen value is a partial solution or not since there is no direct link to the known $a^i_{76..80}$ values. Due to the manipulation that is possible during later rounds, any w-value assignment is acceptable and can be adjusted for by modifications to earlier q-values. Once again, this manipulation remains possible until the earlier rounds are encountered.

The aforementioned constraints can be strengthened by the addition of constraints over w, which constrain those values to be in accordance with the linear system that generates those values. This system can, when used in conjunction with Gauss-Jordan elimination (Anton, 2010), be used to indicate that an erroneous assignment has occurred at some point by obtaining an insoluble system. This kind of detection is most useful when working from later rounds towards earlier rounds; in earlier rounds, w values are being assigned directly and simply asserted to be correct. However, two issues arise from the use of such a system of constraints.

1. The number of equations must be equal to the number of variables being examined.
2. No degenerate equations are permitted.

A system where the number of equations is less than the number of variables is underdetermined, and Gauss-Jordan elimination will not be able to find values for variables. Conversely, a system where the number of equations is greater than the number of variables is overdetermined, and Gauss-Jordan elimination will result in an inconsistent set of assignments. Issue (1) relates to the number of equations that must therefore be chosen. Ideally, each equation would cover the same set of variables; however, this is not possible in practice. The particular expanded-word w-values must be carefully chosen to overlap while introducing as few additional variables as possible. In addition, it would be useful for the chosen w-values to be as close to $q_{76..80}$ as possible since, given their already-constrained domains, this makes it easier to obtain empty domains. Unfortunately, the later w-values are also those which have the largest number of terms in their expansion-word equations.

Issue (2) is about which equations are included in the system. A *degenerate* equation in a linear system is one which can be formulated by the linear combination of other equations in the system. Therefore, it provides no additional information about variable values and, if not replaced by a non-degenerate equation, will result in a solution which is not unique (and, in our context, therefore likely to be incorrect) or a solution which is incorrect. A degenerate system can be detected by calculating the determinant of the system's coefficient matrix. If the determinant is non-zero, then the system does not contain any degenerate equations. Determining which equations are linear combinations of other equations can be done via exhaustive testing, though it may be simpler to "swap out" equations until a non-degenerate system is found. For example, given a pool of 13 equations and 8 variables, there are 5 equations which should be excluded from the linear system; by permuting or shuffling the pool and taking the first 8 equations, a non-degenerate system may be found. A more structured, but much more computationally intensive, option may be to pre-calculate which equations are degenerate in relation to which other equations, since this depends purely on the equations themselves and not on the data.

A significant problem with relying on the w-equation constraints to detect errors is that although an error indicates that an incorrect assignment has been made, the location of the assignment is unknown. Assuming that n variables are considered in the linear system, this means that a search space of 2^n possibilities has to be considered.

No solution to these issues could be found. It therefore appears that a non-binary representation results in similar computationally infeasible results to the binary representations mentioned in the literature.

11.2 Sophisticated implementation

The CSP may be more tractable if implemented in a more sophisticated way; it may be the case that more sophisticated propagators would be able to find relationships that are not immediately obvious, or to pare domains down more efficiently than an ad-hoc implementation. To this end, the state-of-the-art Gecode[2] (Tack, 2009) framework was used to represent the non-binary model that has been described, using q and w variable arrays respectively. Branching "defines the shape of the search tree" (Schulte, Tack, and Lagerkvist, 2016) in Gecode, and requires two things: the selection of a variable to branch on, and a means of splitting the values in the domain to create a choice. In all cases, domains with a smaller number of variables were selected (the `INT_VAR_SIZE_MIN` option in Gecode), and domains were split using the mean of largest and smallest domain values, weighted towards larger values (the `INT_VAL_SPLIT_MAX` option in Gecode).

Branching defines the shape of the search tree, and search algorithms define the order in which the tree is explored. A custom search algorithm which took into account the probability of carries made no difference to any result. The depth-first search algorithm (`DFS` in Gecode) was used in all cases; the more recent Least Discrepancy Search (LDS) occasionally returned false-negative results, and the Branch-And-Bound (BAB) search is more geared towards finding the best possible solution.

For reproducibility, it should be stated that the C++ code of the sophisticated implementation needed to be compiled with the `clang++`[3] compiler; both the Visual Studio 2015 C++ toolchain (`cl`[4] and `link`[5]) and the GNU `g++`[6] compiler spent close to an hour attempting to compile the code before giving up. The exact reasons for this are unknown, but it is hypothesised that such long compilation times could be due to Gecode's extensive use of C++ template functionality. Since the fundamental design of Gecode is unlikely to change, it is likely that researchers wishing to replicate or extend these results will run into similar issues.

[2]Gecode version 5.0.0
[3]clang version 3.8.1
[4]cl version 19.00.24210
[5]link version 14.00.24210
[6]g++ version 6.2.0

The model admitted the possibility of several variations:

1. The ω, as with other representations, could be altered.
2. Branching could occur on w variables, q variables, q-then-w variables, or w-then-q variables.
3. The number of constraints could be reduced by excluding a certain percentage of constraints; however, to make false-positives less likely, the last five rounds would always be fully-constrained.
4. In the case of a second-preimage, the exact q values would be known; this, in turn, would decrease the search space by fixing the final values instead of simply restricting them to be even, if $a^i_{r\geq 76} = 0$, or odd otherwise.
5. The number of rounds could be reduced.

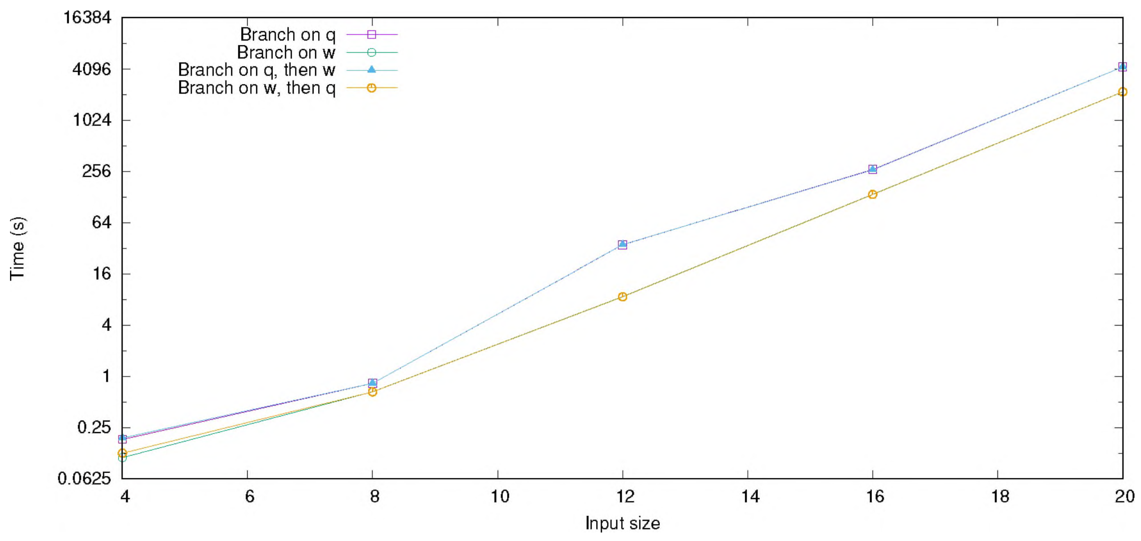

Figure 11.1: Tractability of non-binary formulation, using a sophisticated implementation

Variations (1) and (2) provide a baseline for how tractable a more sophisticated implementation can make the preimage CSP representation. Variation (2) is particularly significant since the shape of the search tree determines the overall scalability of the approach. Figure 11.1 shows that the tractability of the problem remains the same despite the shape of the search tree, although the shape of the tree does have a significant effect on the time taken to obtain a preimage. This result implied that the more sophisticated implementation is unable to find any additional inferences to constrain the model, and that implication was verified through an interactive exploration of the search tree using Gecode's Gist user interface (Schulte *et al.*, 2016). Figure 11.1 makes it clear that branching on w is the most efficient way to approach the problem, and the following variations will therefore restrict themselves to this branching method. A depth-first search through the q variables involved a larger number of choices to be made and this accounts for the increased time taken.

Gecode, by default, uses the concept of "no-goods": paths that can be shown to be impossible are excluded from the list of paths to try in future. The exponential curve shown in Figure 11.1 implies that this no-goods list is not being used effectively: failed paths give no information about correct paths.

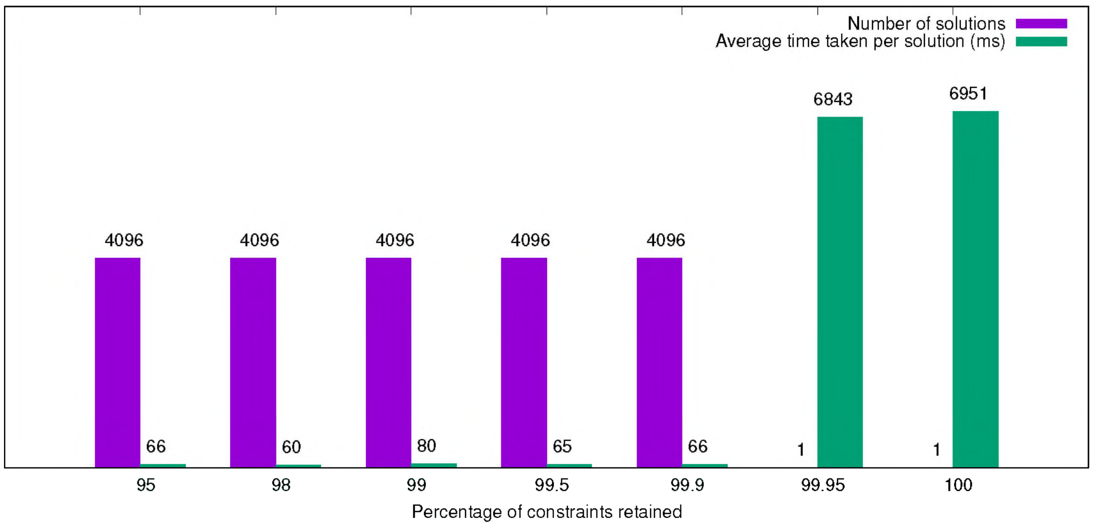

Figure 11.2: Number of solutions generated for reduced constraints ($\omega = 12$)

Variation (3) is premised on the idea of reducing the number of constraint checks to be made and thus increasing the efficiency of the system. Performance is, indeed, increased: each solution takes mere milliseconds to find. However, the false positive rate is $2^{\omega} - 1$ (see Figure 11.2), which means that the solutions generated are simply an enumeration of the search space. Although Figure 11.2 shows results for $\omega = 12$, the same pattern was obtained for $\omega \in \{4, 8, 16, 20\}$. Removing a single constraint results in the same performance characteristics, and the same single solution, as removing no constraints; removing more than one constraint results in an exhaustive enumeration of the search space. It is hypothesised that the reduced number of checks makes it more difficult to prune values efficiently and an exhaustive search through unpruned values must therefore be undertaken. This result demonstrates the importance of every constraint in the system; no simplification of the system through constraint removal is possible. It also demonstrates the importance of determining constraints for the system since more constraints lead to fewer values that need to be exhaustively tested.

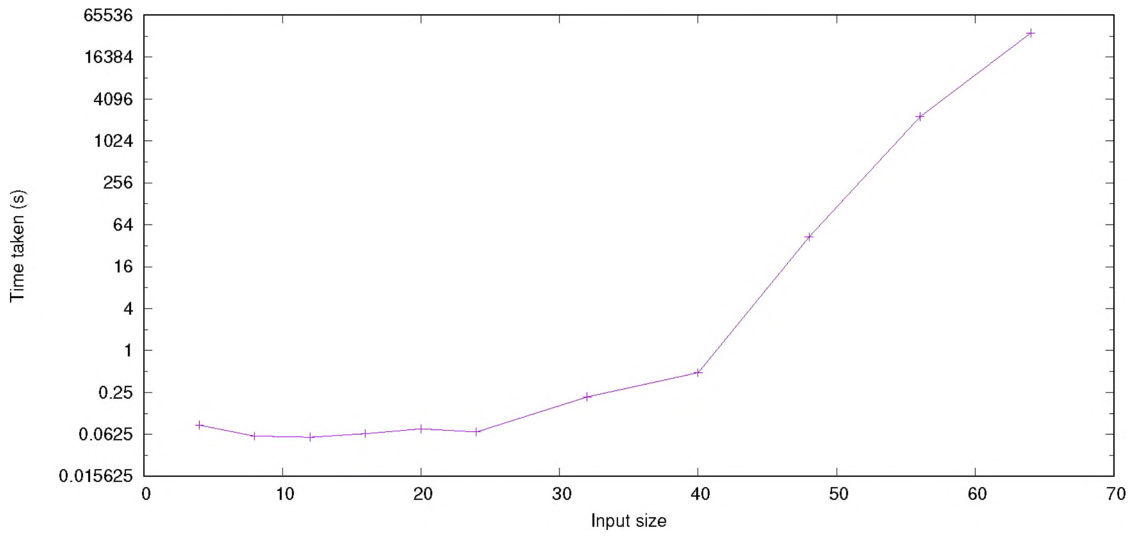

Figure 11.3: Reducing the output search space

Variation (4) can be expected to have a significant positive effect on performance since it reduces the search space. Such an effect can be seen in Figure 11.3, which is worth comparing to the baseline shown in Figure 11.1. The results indicate that the positive effect is sharply curtailed as ω increases. This is likely due to the fact that, although the search space is greatly reduced, it is not entirely eliminated — and a larger input size necessarily entails a larger space to search through. Although the amount of time taken is drastically reduced, the result is of limited utility since fixing the output in a non-binary formulation converts the compression function into an expansion function.

Recall that the pigeonhole principle, first mentioned in Chapter 2, states that if n objects are distributed among k compartments where $n > k$, then at least one compartment must contain more than one object. A non-binary formulation increases k by a factor of 5: a 0-value is mapped to $\{0, 2, 4, 6, 8\}$ and a 1-value is mapped to $\{1, 3, 5, 7, 9\}$. This means that the 512 (or $\approx$ 456, for valid single-block SHA-1) inputs are mapped to $160 \times 5 = 800$ possible compartments and second-preimages for an arbitrary compression function output are no longer guaranteed to exist. Since the non-binary output can only be fixed to exact values if an input is known, but exact values do not necessarily result in a second-preimage, the utility of variation (4) is limited. Nevertheless, if some way can be devised to guess q-values from the pattern of a-values, then restricting q-values appropriately can provide a very significant reduction in solving time.

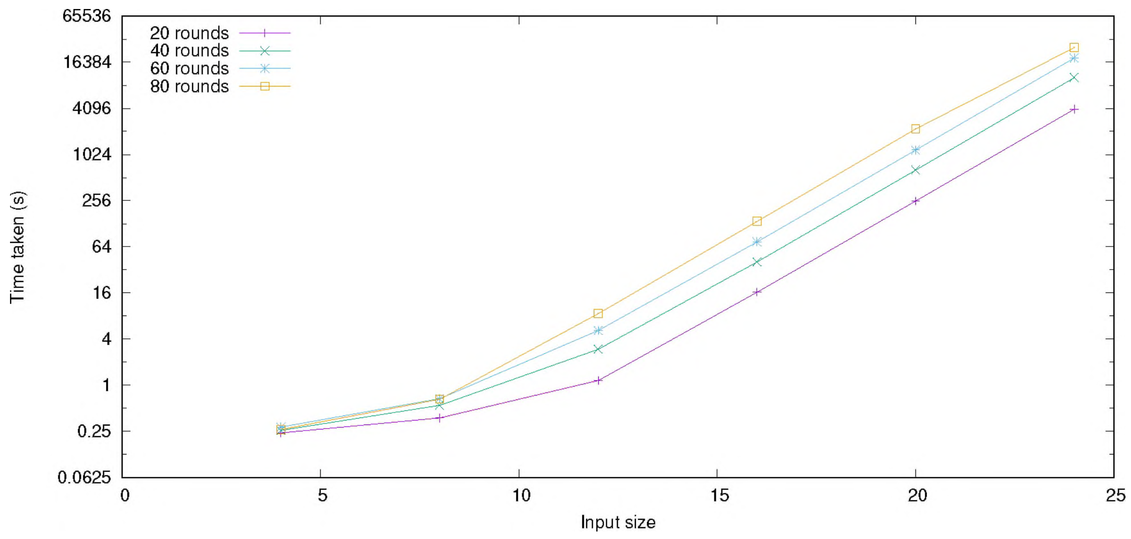

Figure 11.4: Reducing the depth of the tree

Variation (5) reduces the depth of the tree that must be explored; see Figure 11.4 shows that every 20 rounds, starting from $\omega = 12$, adds a relatively constant factor to the time required to find a solution. Taking the geometric mean of the differences across input sizes from $\omega = 12$ onwards, 40 rounds takes $\approx$ 2.53x more time than 20 rounds; 60 rounds takes $\approx$ 1.80x more time than 40 rounds; and 80 rounds takes $\approx$ 1.68x more time than 60 rounds.

11.3 Summary

The choice of whether to model the SHA-1 preimage problem as binary or non-binary CSP leads to an interesting theoretical discussion of trade-offs, but has little impact on the final result: no model, either in the literature or implemented in this research, provided a better-than-brute-force way to find a solution. Branching on w was better than branching on q, and the non-binary model made it easier to fix both the carry and the final value and understand the impact of doing so. An important result from this section is that adding more rounds to the problem becomes increasingly ineffective.

Part IV

Discussion & conclusions

Chapter 12

Discussion

This chapter reflects on the design decisions that lead to the remarkable preimage resistance of SHA-1, notes certain novel contributions of this research, and discusses general lessons learned from the experiments conducted in Part III. Future research directions are proposed to end off the chapter.

By way of summarising the progress towards finding a preimage, it is worthwhile to once again look at the framework of Rogaway and Shrimpton (2009). This framework presents the aPre formalisation included in this work as Equation 3.1 in Section 3.2 (see page 24). A simplified formalisation that does not take the chaining value into consideration is

$$\mathbf{Adv}_H^{\mathrm{aPre}[m]}(A) = \Pr_{\max}\left[M \xleftarrow{\$} \{0,1\}^m ; Y \leftarrow H(M); M' \xleftarrow{\$} A(Y) : H(M') = Y\right]$$

No representation, algorithm, or approach that has been considered is likely to give an attacker any advantage over the brute-force approach; no $A(Y)$ has been discovered.

12.1 Design decisions

In terms of hash function evolution, SHA-1 follows on from MD5 (Rivest, 1992b), which in turn followed on from MD4 (Rivest, 1992a). MD4 was originally created in 1990 as a cryptographic hash with an emphasis on performance; it seems likely that this emphasis could have been intended to overcome the objection that adding security made systems "slow". However, MD4 was quickly broken (Dobbertin, 1998), and the signs of this impending collapse were evident soon after the algorithm was created. MD5 took a different approach (Rivest, 1992b):

> MD5 is slightly slower than MD4, but is more "conservative" in design. MD5 was designed because it was felt that MD4 was perhaps being adopted for use more quickly than justified by the existing critical review; because MD4 was designed to be exceptionally fast, it is "at the edge" in terms of risking successful cryptanalytic attack. MD5 backs off a bit, giving up a little in speed for a much greater likelihood of ultimate security.

SHA-1 follows in this tradition of prioritising security over speed. The diffusion of SHA-1, as investigated in Section 5.2, is critical to preimage-resistance of the compression function since it ensures that there are no clues in the output that could indicate which input was used. The way in which linear and nonlinear functions are mixed is essential to SHA-1's excellent diffusion characteristics; the properties of each function have been discussed in Section 4.2.2. Of particular interest, as far as preimage resistance is concerned, is the fact that all f-functions are balanced and almost all component-functions are either 0th-order or 1st-order correlation-immune.

The message-expansion phase (Section 4.1) of SHA-1 is purely linear, and most of the spliffling functions — i.e., the majority, choice, and carry sub-functions — are nonlinear (Section 4.2.2). Recall that the essential difference between a linear and nonlinear function is that the output of the former always depends on *all* the inputs. By using a linear function in message-expansion to distribute the bits of the input, and a nonlinear function in spliffling to process them, the SHA-1 function ensures that each bit has multiple opportunities to participate in the hash function output, even if any particular bit happens to be ignored by a nonlinear function. This aspect of the design is an improvement over SHA-1's predecessor MD5 (Rivest, 1992b), which does not diffuse the bits in a separate message-expansion phase.

The theoretical investigation of SHA-1 as a CSP (Section 5.1) showed that SHA-1 is a decision problem in the L complexity class, solvable in $\mathcal{O}(2^{\log n})$ time. However, it should be noted that the complexity class of a decision problem relates the size of the input to the time/space requirements of that problem, *ignoring* any constant factor. The results of Section 11.2 relate the tree depth to the time taken to find a preimage. It is this tree depth of 80 rounds that adds a constant factor, making the average case difficult to solve. Section 11.2 demonstrated that the increased difficulty imposed by every additional 20-round block decreases. It is therefore plausible that similar experiments on the part of SHA-1's designers caused them to settle on 80 as a number of rounds that balances preimage resistance and computational cost.

12.2 Contribution

Much of the research described in this work is not found in any published literature. The analysis of message-expansion and spliffling (Chapter 4) resulted in an alternative w-formulation (Section 4.1.1), an alternative q-variable formulation (Section 4.3). An algorithm to efficiently find valid expansion-equations (Section 4.1.3) and an exploration of bit-pattern subexpressions (Section 4.1.2) are also presented. The treatment of SHA-1 as a CSP (Section 5.1) is also novel, and the result — that SHA-1, assuming a partial solution, is theoretically solvable in $\mathcal{O}(2^{\log n})$ time — was not found in any of the literature surveyed. The statistical analysis of SHA-1 to determine its adherence to the Strict Avalanche Criterion (Section 5.2) is also novel, and is based on previously-published work (Motara, 2016).

Part III comprises many different representations of the SHA-1 preimage problem. To the best of my knowledge, such practical experiments have not been described in the literature, with the exception of some aspects of Chapter 7. The experiments confirmed the difficulty of the SHA-1 preimage problem, and in many cases they represent the very first results of their type. Certain experiments, such as those relating to intermediate representation size reduction (Section 9.3) and those relating calculation method to solving-time (Section 7.2), serve to quantify the difficulty in the context of a particular representation. Selected examples of novel experiments and results are the application of a modern heuristic minimizer such as BOOM-II (Chapter 8), the irrelevance of heuristic reordering algorithms for ROBDDs (Section 9.1), and the use of AIGs and related tools to attempt to simplify a directed acyclic preimage graph (Chapter 10).

12.3 Preimage representation

Existing representations make it possible to do one of two things: to describe the constraints of a search space (CNF, AIG and CSP), or to describe a search space (DNF and BDD). The former is inevitably time-inefficient, and the latter is inevitably space-inefficient. A representation must reflect all the necessary constraints, and must therefore fully model the SHA-1 compression function. Any representation that does not do this makes it impossible to check whether a partial solution has been found.

An ideal representation of SHA-1 does not exist. An ideal data structure would be compactly and flexibly represented (as is the case with an AIG); efficient and minimal to represent in preimage form, and retaining arc-consistency, as is the case with a BDD; and easily manipulable and traversed using state-of-the-art tools, as is the case with both CNF and AIG forms. None of the representations has all of these qualities. A BDD expands far too much before a preimage can restrict it; an AIG is flexible, but difficult to reduce to a minimal form; and the difficulties of solving for a solution using CNF or a CSP representation have been experimentally demonstrated.

The research into representations makes it possible to draw conclusions about properties of the SHA-1 compression function, and potentially about preimage research in general.

It is difficult to establish a “good” way (in the context of the preimage problem) to generate a SHA-1 representation. Section 7.2 demonstrates that different SAT solvers found different w-calculation methods to be optimal, despite the *a priori* recommendations in the relevant literature (see Nossum (2012); Legendre *et al.* (2012)) about what *should* be optimal. This points to the importance of experimental verification when heuristic methods are employed.

SHA-1 has excellent diffusion properties. The Strict Avalanche Criterion results (Section 5.2 and results such as those shown in Figure 9.5 indicate the importance of considering every output bit when attempting to find a preimage; an attacker’s algorithm cannot simply consider a subset of the output bits.

Finding a SHA-1 preimage is, as expected, a very hard problem although it is not among the *hardest* known problems (Section 5.1). This research has verified that the SHA-1 problem is not easily solvable via heuristic minimization (Chapter 8; Section 10.1), nor by decomposition (Chapter 9); nor is any advance in these fields (apart from a solution to P=NP) likely to make the problem easier.

The diffusion properties and minimization resistance of SHA-1 make probability-based attacks difficult to conceptualize: every path appears to be equally possible and all paths are approximately the same length. Since the input is fixed in advance and the output cannot be modified, the “standard” probability-based attacks used to find collisions are inapplicable.

12.4 Future research directions

Reflecting on the work performed during this research has led to consideration of the research directions mentioned in this chapter. These possibilities are not exhaustive: it may be that a breakthrough in an “unrelated” field, such as multivariate polynomial multiplication (see the excluded “Integer representation” in Section 6.4), would lead to an easier way to find a preimage. Instead of enumerating such possibilities, of which there are many, this section suggests work that can build directly upon the foundation laid in earlier chapters.

The AIG representation, out of all the representations explored in this work, is particularly interesting as a target for future preimage research. It is scalable and amenable to optimization at micro- and macro-levels, and benefits from research conducted in the fields of graph theory and electronic design, automation, and optimization.

The SHA-1 preimage problem is complicated, but not necessarily complex. Recalling the definition of “partial solution” (Chen, 2009), the issue becomes one of representation. Each preimage has a simple, and typically small, definition; however, the intermediate representation that is created to find the preimage ends up representing the solution space of all possible solutions before being “pared down” to the much-simpler preimage definition. Although different representations have been considered, a hybrid representation and accompanying search algorithm may be worth investigating.

SAT solvers are one of the most powerful tools available for solving combinatorial problems, and are widely applicable to a vast range of problem domains. However, they are also unable to explore the generically-represented problem space efficiently, taking into consideration all the constraints. SAT solvers are geared towards solving tractable problems via heuristic methods and, as mentioned in Chapter 7, they typically have an inordinate number of variables that may be tweaked. It would be interesting to build on the research of Legendre *et al.* (2012) or Nossum (2013) to create a SAT solver that is specifically tuned towards the structure of the SHA-1 preimage problem. The use of an interactive visualiser, or a non-binary formulation, may also be useful in this regard.

Lastly, this work focused on a single-threaded algorithmic approach. A multi-threaded approach to the preimage problem is quite possible, especially in the cases of CSP search and AIG decomposition, and may be worth pursuing. This is a brute-force style of solving the problem, and unlikely to be successful (see Section 3.5). However, if combined with some algorithmic insight, it may be good enough for all practical purposes: if an input where $\omega \approx 160$ can be solved by brute force, then the balance and SAC characteristics of SHA-1 suggest that a preimage can be found for (almost) any output.

12.5 Summary

This chapter has discussed and related the results of the experiments and analysis detailed in Part II, and proposed additional research questions that could fruitfully be explored. The impact of this research is, as already expressed in Section 1.2, limited. At the theoretical level, SHA-1 is a dedicated hash function and identifying a flaw in the function may be difficult to generalize to any other area. At the practical level, modern (and maintained) systems should eventually switch to a hash function such as SHA-3, and this will limit the practical applicability of any SHA-1 preimage research. Outdated, unmaintained, or security-ignorant software will continue to use SHA-1, and further research into SHA-1 preimages will be able to target these. Such software, and the software developers which develop it, can take comfort in the knowledge that inasmuch as their software relies on the preimage-resistance of SHA-1, it should be secure for the foreseeable future.

The next chapter builds on the discussion of this chapter and lays out the primary conclusions of this research.

Chapter 13

Conclusions

The goal of this research was to explore and understand the SHA-1 compression function, focusing on the problem of finding preimages. The specific contributions, as noted in Section 1.2, are:

1. an in-depth analysis of, and alternative valid formulations of, the SHA-1 compression function and its components (Chapter 4);
2. a statistical analysis of diffusion characteristics (Chapter 5);
3. multiple representations of the SHA-1 compression function, with preimage-focused practical experiments (Chapters 7, 8, 9, 10, and 11);
4. a reflection on representations and SHA-1 design choices (Chapter 12).

This chapter recounts interesting results and conclusions from Parts II and III of this work and presents them here in summarised form.

As message expansion progresses, more data-words and bits from each data-word tend to be used in each round, but this trend is not strict: for example, round 78 uses no bits from w_1 (Section 4.1 and Table 4.1). Different bits from a single data-word affect the calculation of an expansion-word, and this can be modeled using the idea of bitpatterns (Section 4.1.1); see Algorithms 4.1 and 4.2, as well as Table 4.1 and Appendix A. Certain common sub-sequences occur during message-expansion, but become much rarer as the length of the sub-sequence increases (Section 4.1.2).

The carry calculation during spliffling is most naturally represented using a ripple-carry design, whether in this research or the work of others (see (Legendre *et al.*, 2012; Nossum, 2013)), but does admit to some small optimization (Section 4.2.1). The component functions used for spliffling have properties summarised in Table 4.7 on page 59. The inputs and outputs for each of the component functions can be unpacked for further analysis (Section 4.2.4) and a non-binary formulation of spliffling is possible (Section 4.3). Finally, a single equation can be used to represent the preimage: either Equation 4.9 or Equation 4.10 (page 64).

When modeled as a CSP(Γ), the SHA-1 preimage problem is a majority-polymorphism CSP(Γ) which consequently has a 2-extension property and is in the NL complexity class (Section 5.1). Statistical analysis of the diffusion of SHA-1 reveals that every bit from round 24 onwards meets, or very closely approximates, the Strict Avalanche Criterion; *all* input bits contribute equally to each output bit and there is no way to determine a relationship between input and output bits (Section 5.2).

Not all representations are suitable for representing the SHA-1 preimage problem (Sections 6.2 and 6.4). The characteristics of a worthwhile representation have been discussed (Section 6.3).

The assumptions made to find a “better” CNF representation of SHA-1 may be unfounded (Section 7.1) since different solvers find different encodings to be more or less tractable (Section 7.2). SLS SAT-solvers find the SHA-1 preimage problem to be entirely intractable and DPLL SAT-solvers are not superior to a brute-force approach; Tseitin variables add little but noise (Section 7.2).

DNF is an entirely unsuitable representation and heuristic methods are unable to overcome the NP-hardness of the boolean minimization problem (Chapter 8).

The SHA-1 preimage problem expands to an unmanageable size when using a ROBDD representation, irrespective of variable ordering (Section 9.1). The multiplicative complexity of SHA-1 is the greatest that it can be while preserving collision resistance (Example 9.3 on page 122). A way to reduce the ROBDD size could not be discovered (Section 9.3) and it is unlikely that *DD variants would provide the necessary reduction (Section 9.2).

Local transformations of the SHA-1 DAG, in AIG form, fail to reduce the size of the graph by a significant amount (Chapter 10), and the structural hashing of nodes is insufficient to compensate for the doubling that occurs as variables are pushed “upwards” in the graph (Section 10.1). Although 4-input functions can always be minimized due to precalculated LUTs, the technique does not scale to larger input sizes (Section 10.2).

Whether modeled in binary or non-binary form, naïvely or using a sophisticated framework, the SHA-1 preimage problem remains intractable (Sections 11.1 and 11.2). Any significant constraint removal causes the system to collapse into a simple enumeration of all possible values in the search space; failed paths give no information about correct paths; and the difficulty added by additional rounds decreases as more rounds are added (Section 11.2). A q-variable formulation which could be predicted could lead to significant performance increases during solving, but no way is known to obtain such a prediction (Section 11.2).

References

Adinetz, A. V. and Grechnikov, E. A. Building a collision for 75-round reduced SHA-1 using GPU clusters. In *European Conference on Parallel Processing*, pages 933–944. Springer, 2012.

Allender, E., Bauland, M., Immerman, N., Schnoor, H., and Vollmer, H. The complexity of satisfiability problems: Refining Schaefer's theorem. *Journal of Computer and System Sciences*, 75(4):245–254, 2009.

Amarú, L., Gaillardon, P.-E., and De Micheli, G. Biconditional BDD: a novel canonical BDD for logic synthesis targeting XOR-rich circuits. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1014–1017. EDA Consortium, 2013.

Anton, H. Elementary linear algebra. John Wiley & Sons, 10^{th} edition, 2010. ISBN 978-0470432051.

Aoki, K. and Sasaki, Y. Meet-in-the-middle preimage attacks against reduced SHA-0 and SHA-1. In *Advances in Cryptology (CRYPTO 2009)*, pages 70–89. Springer, 2009.

Audemard, G. and Simon, L. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st international joint conference on Artifical Intelligence*, pages 399–404. Morgan Kaufmann Publishers Inc., 2009.

Babbage, S. On the relevance of the strict avalance criterion. *Electronic Letters*, 26(7):461, 1990.

Bacchus, F., Chen, X., Van Beek, P., and Walsh, T. Binary vs. non-binary constraints. *Artificial Intelligence*, 140(1):1–37, 2002.

Bahar, R. I., Frohm, E. A., Gaona, C. M., Hachtel, G. D., Macii, E., Pardo, A., and Somenzi, F. Algebraic decision diagrams and their applications. *Formal methods in system design*, 10(2-3):171–206, 1997.

Balint, A. and Schöning, U. Choosing probability distributions for stochastic local search and the role of make versus break. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 16–29. Springer, 2012.

Bard, G. V. Algorithms for solving linear and polynomial systems of equations over finite fields, with applications to cryptanalysis. Ph.D. thesis, Department of Mathematics, University of Maryland, United States of America, 2007.

Bellare, M. and Kohno, T. Hash function balance and its impact on birthday attacks. In *Advances in Cryptology (EUROCRYPT 2004)*, pages 401–418. Springer, 2004. ISBN 978-3-540-24676-3.

Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. 2016. Release 160605. Accessed 1 December 2016. URL `http://www.eecs.berkeley.edu/~alanmi/abc/`

Biere, A. The AIGER And-Inverter Graph (AIG) Format Version 20071012. Technical report, Johannes Kepler University, 2007. Accessed 1 December 2016. URL `http://fmv.jku.at/aiger/FORMAT.aiger`

Biere, A. Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013. *Proceedings of SAT Competition 2013*, pages 51–52, 2013.

Biere, A. Yet Another Local Search Solver and Lingeling and friends entering the SAT competition 2014. *Proceedings of SAT Competition 2014*, pages 39–40, 2014.

Biere, A., Heule, M., and van Maaren, H. Handbook of satisfiability, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS press, 2009.

Biryukov, A. and Velichkov, V. A Method for Automatic Search for Differential Trails in ARX Ciphers. *Early Symmetric Crypto (ESC 2013)*, pages 68–97, 2013.

Bloom, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

Bodrato, M. Towards optimal Toom-Cook multiplication for univariate and multivariate polynomials in characteristic 2 and 0. In *International Workshop on the Arithmetic of Finite Fields*, pages 116–133. Springer, 2007.

Bollig, B. and Wegener, I. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.

Boole, G. An investigation of the laws of thought: on which are founded the mathematical theories of logic and probabilities, volume 2. Walton and Maberly, 1854.

Borodin, A., Diaconis, P., and Fulman, J. On adding a list of numbers (and other one-dependent determinantal processes). *Bulletin of the American Mathematical Society*, 47(4):639–670, 2010.

Bosselaers, A. Even faster hashing on the Pentium. 1997. Presented at the Rump Session of Eurocrypt '97. Accessed 1 December 2016.
URL `http://www.esat.kuleuven.ac.be/cosicart/pdf/AB-9701.pdf`

Bosselaers, A., Govaerts, R., and Vandewalle, J. Fast Hashing on the Pentium. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, pages 298–312. Springer-Verlag, 1996.

Bosselaers, A., Govaerts, R., and Vandewalle, J. SHA: a design for parallel architectures? In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 348–362. Springer, 1997.

Boyar, J., Peralta, R., and Pochuev, D. On the Multiplicative Complexity of Boolean Functions over the Basis $(\wedge, \oplus, 1)$. *Theoretical Computer Science*, 235(1):43–57, 2000.

Bradley, J. and Davies, N. Compositional BDD Construction: A Lazy Algorithm. Technical report, Department of Computer Science, University of Bristol, Bristol, UK, 1998.

Braeken, A. Cryptographic properties of Boolean functions and S-boxes. Ph.D. thesis, Electrical Engineering Department, Katholieke Universiteit Leuven, 2006.

Brailsford, S. C., Potts, C. N., and Smith, B. M. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research*, 119(3):557–581, 1999.

Brayton, R. K., Sangiovanni-Vincentelli, A. L., McMullen, C. T., and Hachtel, G. D. Logic Minimization Algorithms for VLSI Synthesis. Kluwer Academic Publishers, Norwell, MA, USA, 1984. ISBN 0898381649.

Brown, F. M. Comments on a numerical method for solving Boolean equations. *Information Sciences*, 181(3):547–551, 2011.

Brualdi, R. Introductory Combinatorics. Pearson Education International, 2012. ISBN 9780132791717.

Bryant, R. E. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986.

Bryant, R. E. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, 1991.

Bryant, R. E. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.

Buchfuhrer, D. and Umans, C. The complexity of Boolean formula minimization. In *Automata, Languages and Programming*, pages 24–35. Springer, 2008.

Buchfuhrer, D. and Umans, C. The complexity of boolean formula minimization. *Journal of Computer and System Sciences*, 77(1):142–153, 2011.

Canteaut, A., Carlet, C., Charpin, P., and Fontaine, C. Propagation Characteristics and Correlation-Immunity of Highly Nonlinear Boolean Functions. In *Advances in Cryptology (EUROCRYPT 2000)*, pages 507–522. Springer, 2000.

Carlet, C. On the propagation criterion of degree l and order k. In *Lecture Notes in Computer Science*, volume 1403, pages 462–474. Springer, 1998.

Chaitin, G. J. Exploring RANDOMNESS. Springer Science + Business Media, 2001. ISBN 1447103076.

Chen, H. A rendezvous of logic, complexity, and algebra. *ACM Computing Surveys (CSUR)*, 42(1):2, 2009.

Cochran, M. Notes on the Wang *et al.* 2^{63} SHA-1 Differential Path. *IACR Cryptology ePrint Archive*, 2007. Accessed 1 December 2016.
URL `https://eprint.iacr.org/2007/474.pdf`

Cook, S. A. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of computing*, pages 151–158. ACM, 1971.

Crama, Y. and Hammer, P. L. Boolean models and methods in mathematics, computer science, and engineering, volume 2. Cambridge University Press, 2010.

Cryptographic Technology Group. NIST Cryptographic Standards and Guidelines Development Process. Technical report, National Institute of Standards and Technology, 2016.

Damgård, I. B. A design principle for hash functions. In *Conference on the Theory and Application of Cryptology*, pages 416–427. Springer, 1989.

Davis, M., Logemann, G., and Loveland, D. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

De Canniere, C. and Rechberger, C. Finding SHA-1 characteristics: General results and applications. In *Advances in Cryptology (ASIACRYPT 2006)*, pages 1–20. Springer, 2006.

De Canniere, C. and Rechberger, C. Preimages for Reduced SHA-0 and SHA-1. In *Advances in Cryptology (CRYPTO 2008)*, pages 179–202. Springer, 2008.

Deutsch, P. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational), May 1996. IETF. Accessed 1 December 2016.
URL `http://www.ietf.org/rfc/rfc1951.txt`

Dobbertin, H. Cryptanalysis of MD4. *Journal of Cryptology*, 11(4):253–271, 1998. ISSN 1432-1378.

Drechsler, R., Becker, B., and Gockel, N. Genetic algorithm for variable ordering of OBDDs. *Computers and Digital Techniques*, 143(6):364–368, 1996.

Drechsler, R. and Sieling, D. Binary decision diagrams in theory and practice. *International Journal on Software Tools for Technology Transfer*, 3(2):112–136, 2001.

Dubuc, S. Characterization of linear structures. *Designs, Codes and Cryptography*, 22(1):33–45, 2001.

Eén, N. and Biere, A. Effective preprocessing in SAT through variable and clause elimination. In *International conference on theory and applications of satisfiability testing*, pages 61–75. Springer, 2005.

Eén, N. and Sörensson, N. An extensible SAT-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

Eén, N. and Sörensson, N. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.

Eichlseder, M., Mendel, F., Nad, T., Rijmen, V., and Schlaeffer, M. Linear propagation in efficient guess-and-determine attacks. In *Proceedings of the International Workshop on Coding and Cryptography (IWCC 2013)*. 2013.

Espitau, T., Fouque, P.-A., and Karpman, P. Higher-Order Differential Meet-in-The-Middle Preimage Attacks on SHA-1 and BLAKE. In *35th International Cryptology Conference (CRYPTO 2015)*, pages 683–701. Springer, 2015.

Ferguson, N. and Schneier, B. Practical cryptography. Wiley, 2003. ISBN 978-0471223573.

Fišer, P. and Kubátová, H. Two-level boolean minimizer BOOM-II. In *6th International Workshop on Boolean Problems (IWSBP '04)*. 2004.

Fišer, P. and Kubátová, H. Flexible two-level Boolean minimizer BOOM-II and its applications. In *9th EUROMICRO Conference on Digital System Design (DSD'06)*, pages 369–376. IEEE, 2006.

Foley, L. Analysis of an on-line random number generator. Technical report, Computer Science Department, Trinity College Dublin, 2001. Project report, The Distributed Systems Group.

Forré, R. The Strict Avalanche Criterion: Spectral Properties of Boolean Functions and an Extended Definition. In *Advances in Cryptology (CRYPTO '88)*, pages 450–468. Springer Science + Business Media, 1990.

Freedman, D., Pisani, R., and Purves, R. Statistics. W.W. Norton & Company, 2007. ISBN 9780393930436.

Gauravaram, P., Millan, W., and Nieto, J. G. Some thoughts on Collision Attacks in the Hash Functions MD5, SHA-0 and SHA-1. *IACR Cryptology ePrint Archive*, 2005. Accessed 1 December 2016.
URL `https://eprint.iacr.org/2005/391.pdf`

Gopalasubramanian, G. [PATCH] add znver1 processor. *binutils* mailing list, March 2015. Accessed 1 December 2016.
URL `https://sourceware.org/ml/binutils/2015-03/msg00078.html`

Gouget, A. On the Propagation Criterion of Boolean Functions. In *Coding Cryptography and Combinatorics*, pages 153–168. Springer, 2004.

Grechnikov, E. A. Collisions for 72-step and 73-step SHA-1: Improvements in the Method of Characteristics. *IACR Cryptology ePrint Archive*, 2010. Accessed 1 December 2016.
URL `http://eprint.iacr.org/2010/413.pdf`

Gulley, S., Gopal, V., Yap, K., Feghali, W., Guilford, J., and Wolrich, G. Intel® SHA Extensions. Technical report, Intel Corporation, July 2013. Accessed 1 December 2016.
URL `https://software.intel.com/sites/default/files/article/402097/intel-sha-extensions-white-paper.pdf`

Guo, J., Ling, S., Rechberger, C., and Wang, H. Advanced meet-in-the-middle preimage attacks: First results on full Tiger, and improved results on MD4 and SHA-2. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 56–75. Springer, 2010.

Joux, A. and Peyrin, T. Hash functions and the (amplified) boomerang attack. In *Advances in Cryptology (CRYPTO 2007)*, pages 244–263. Springer, 2007.

Jovanović, D. and Janičić, P. Logical analysis of hash functions. In *International Workshop on Frontiers of Combining Systems*, pages 200–215. Springer, 2005.

Jutla, C. S. and Patthak, A. C. Is SHA-1 conceptually sound? *IACR Cryptology ePrint Archive*, 2005. Accessed 1 December 2016.
URL `http://eprint.iacr.org/2005/350`

Karnaugh, M. The map method for synthesis of combinational logic circuits. *Transaction of the American Institute of Electrical Engineers*, 72(5):593–599, Nov 1953. ISSN 0097-2452.

Karp, R. M. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Springer, 1972.

Kautz, H. and Selman, B. Pushing the envelope: planning, propositional logic, and stochastic search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1194–1201. 1996.

Kelsey, J. and Schneier, B. Second preimages on n-bit hash functions for much less than 2^n work. In *Advances in Cryptology (EUROCRYPT 2005)*, pages 474–490. Springer, 2005.

Kenny, C. Random number generators: An evaluation and comparison of `random.org` and some commonly used generators. Technical report, Computer Science Department, Trinity College Dublin, 2005.

Khovratovich, D. and Nikolić, I. Rotational cryptanalysis of ARX. In *Fast Software Encryption*, pages 333–346. Springer, 2010.

Kim, K., Matsumoto, T., and Imai, H. A Recursive Construction Method of S-boxes Satisfying Strict Avalanche Criterion. In *Advances in Cryptology (CRYPTO '90)*, pages 565–574. Springer Science + Business Media, 1991.

Knellwolf, S. and Khovratovich, D. New preimage attacks against reduced SHA-1. In *Advances in Cryptology (CRYPTO 2012)*, pages 367–383. Springer, 2012.

Knuth, D. E. Combinatorial Algorithms, Part 1, volume 4A of *The Art of Computer Programming*. Addison-Wesley Professional, 2011.

Kojevnikov, A. and Nikolenko, S. I. New combinatorial complete one-way functions. In *Symposium on Theoretical Aspects of Computer Science 2008*, pages 457–466. 2008.

Krause, M., Savickỳ, P., and Wegener, I. Approximations by OBDDs and the variable ordering problem. In *International Colloquium on Automata, Languages, and Programming*, pages 493–502. Springer, 1999.

Kuehlmann, A., Ganai, M. K., and Paruthi, V. Circuit-based Boolean Reasoning. In *Proceedings of the 38th Annual Design Automation Conference*, pages 232–237. ACM, 2001.

Lai, X. Higher order derivatives and differential cryptanalysis. In *Communications and Cryptography*, volume 276, pages 227–233. Springer, 1994. ISBN 978-1-4615-2694-0.

Langberg, M., Pnueli, A., and Rodeh, Y. The ROBDD size of simple CNF formulas. In *Correct Hardware Design and Verification Methods*, pages 363–377. Springer, 2003.

Legendre, F., Dequen, G., and Krajecki, M. Encoding hash functions as a SAT problem. In *24th International Conference on Tools with Artificial Intelligence (ICTAI 2012)*, pages 916–921. IEEE, 2012.

Legendre, F., Dequen, G., and Krajecki, M. Logical Reasoning to Detect Weaknesses About SHA-1 and MD4/5. *IACR Cryptology ePrint Archive*, 2014:239, 2014.

Leurent, G. MD4 is not one-way. In *International Workshop on Fast Software Encryption*, pages 412–428. Springer, 2008.

Leurent, G. Construction et Analyse de fonctions de Hachage. Ph.D. thesis, Université Paris Diderot, 2010.

Leurent, G. Analysis of differential attacks in ARX constructions. In *Advances in Cryptology (ASIACRYPT 2012)*, pages 226–243. Springer, 2012.

Levin, L. A. The tale of one-way functions. *Problems of Information Transmission*, 39(1):92–103, 2003.

Li, C.-M. and Ye, B. SAT-Encoding of Step-Reduced MD5. *SAT COMPETITION 2014*, page 94, 2014.

Lien, R., Grembowski, T., and Gaj, K. A 1 Gbit/s partially unrolled architecture of hash functions SHA-1 and SHA-512. In *Cryptographers' Track at the RSA Conference*, pages 324–338. Springer, 2004.

Lipmaa, H. and Moriai, S. Efficient algorithms for computing differential properties of addition. In *International Workshop on Fast Software Encryption*, pages 336–350. Springer, 2001.

Lloyd, S. Counting Functions Satisfying a Higher Order Strict Avalanche Criterion. In *Lecture Notes in Computer Science*, pages 63–74. Springer Science + Business Media, 1990.

Lloyd, S. Balance uncorrelatedness and the strict avalanche criterion. *Discrete Applied Mathematics*, 41(3):223–233, Feb 1993.

Luo, C., Cai, S., Wu, W., and Su, K. Focused random walk with configuration checking and break minimum for satisfiability. In *International Conference on Principles and Practice of Constraint Programming*, pages 481–496. Springer, 2013.

Luo, C., Cai, S., Wu, W., and Su, K. Double Configuration Checking in Stochastic Local Search for Satisfiability. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence*, pages 2703–2709. 2014.

Madre, J.-C. and Billon, J.-P. Proving circuit correctness using formal comparison between expected and extracted behaviour. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 205–210. IEEE Computer Society Press, 1988.

Malik, S. and Zhang, L. Boolean satisfiability from theoretical hardness to practical success. *Communications of the ACM*, 52(8):76–82, 2009.

Manuel, S. Classification and generation of disturbance vectors for collision attacks against SHA-1. *Designs, Codes and Cryptography*, 59(1-3):247–263, 2011.

Massacci, F. Using Walk-SAT and Rel-Sat for Cryptographic Key Search. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 290–295. Morgan Kaufmann Publishers Inc., 1999.

Massacci, F. and Marraro, L. Logical cryptanalysis as a SAT problem. *Journal of Automated Reasoning*, 24(1-2):165–203, 2000.

McCluskey, E. J. Minimization of Boolean functions. *The Bell System Technical Journal*, 35(6):1417–1444, Nov 1956. ISSN 0005-8580.

Menezes, A. J., Van Oorschot, P. C., and Vanstone, S. A. Handbook of applied cryptography. CRC press, 1996. ISBN 0-8493-8523-7.

Merkle, R. C. Secrecy, authentication, and public key systems. Ph.D. thesis, Department of Electrical Engineering, Stanford University, 1979.

Merkle, R. C. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238. Springer, 1989.

Minato, S. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *Proceedings of the 30th Conference on Design Automation*, pages 272–277. June 1993. ISSN 0738-100X.

Minato, S. Techniques of BDD/ZDD: brief history and recent activity. *IEICE Transactions on Information and Systems*, 96(7):1419–1429, 2013.

Mishchenko, A. and Brayton, R. Scalable Logic Synthesis using a Simple Circuit Structure. In *Proceedings of the 15th International Workshop on Logic & Synthesis*, pages 15–22. ACM, 2006.

Mishchenko, A., Chatterjee, S., and Brayton, R. DAG-aware AIG rewriting: a fresh look at combinational logic synthesis. In *Proceedings of the 43rd Annual Design Automation Conference*, pages 532–535. ACM, 2006.

Mishchenko, A., Chatterjee, S., Jiang, R., and Brayton, R. K. FRAIGs: A unifying representation for logic synthesis and verification. Technical report, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 2005.

Morawiecki, P. and Srebrny, M. A SAT-based preimage analysis of reduced Keccak hash functions. *Information Processing Letters*, 113(10):392–397, 2013.

Motara, Y. M. SHA-1 and the Strict Avalanche Criterion. In *Proceedings of the 2016 Information Security for South Africa (ISSA 2016) Conference*. IEEE, 2016.

Nelson, V. Digital Logic Circuit Analysis and Design. Prentice Hall, 1995. ISBN 9780134638942.

NIST. Federal information processing standard (fips) 180-1. Secure hash standard. *National Institute of Standards and Technology*, 17, 1995.

NIST. Federal information processing standard (fips) 202. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. *National Institute of Standards and Technology*, 2015.

Nossum, V. SAT-based preimage attacks on SHA-1. Master's thesis, Department of Informatics, University of Oslo, Norway, 2012.

Nossum, V. Instance generator for encoding preimage, second-preimage, and collision attacks on SHA-1. *Proceedings of the SAT competition*, pages 119–120, 2013.

O'Connor, L. and Klapper, A. Algebraic nonlinearity and its applications to cryptography. *Journal of Cryptology*, 7(4):213–227, 1994.

Paar, C. and Pelzl, J. Understanding cryptography: a textbook for students and practitioners. Springer Science & Business Media, 2009.

Petrick, S. R. A direct determination of the irredundant forms of a Boolean function from the set of prime implicants. Technical report, Air Force Cambridge Research Center, 1956.

Pieprzyk, J. and Finkelstein, G. Towards effective nonlinear cryptosystem design. *IEE Proceedings E (Computers and Digital Techniques)*, 135(6):325–335, 1988.

Popescu, D. A. and Garcia, R. T. Multivariate Polynomial Multiplication on GPU. *Procedia Computer Science*, 80:154–165, June 2016. ISSN 1877-0509. International Conference on Computational Science 2016 (ICCS).

Preneel, B. Analysis and design of cryptographic hash functions. Ph.D. thesis, Electrical Engineering Department, Katholieke Universiteit Leuven, 1993.

Rechberger, C. Second-preimage analysis of reduced SHA-1. In *Information Security and Privacy*, pages 104–116. Springer, 2010.

Régin, J.-C. AC-*: A Configurable, Generic and Adaptive Arc Consistency Algorithm. In **van Beek, P.**, editor, *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP 2005)*, pages 505–519. Springer, October 2005. ISBN 978-3-540-32050-0.

Richerby, D. Can a k-ary relation have polymorphisms of arity greater than k? Computer Science Stack Exchange, 2016. Accessed 2 September 2016.
URL `http://cs.stackexchange.com/q/63110`

Rivest, R. The MD4 Message-Digest Algorithm. RFC 1320 (Historic), April 1992a. IETF. Obsoleted by RFC 6150. Accessed 1 December 2016.
URL `http://www.ietf.org/rfc/rfc1320.txt`

Rivest, R. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), April 1992b. Updated by RFC 6151. Accessed 1 December 2016.
URL `http://www.ietf.org/rfc/rfc1321.txt`

Rivest, R., Shamir, A., and Adleman, L. Cryptographic communications system and method. September 20 1983. US Patent 4,405,829. Accessed 1 December 2016.
URL `http://www.google.com/patents/US4405829`

Rjaško, M. Black-box property of cryptographic hash functions. In *International Symposium on Foundations and Practice of Security*, pages 181–193. Springer, 2011.

Rogaway, P. and Shrimpton, T. Cryptographic Hash-Function Basics: Definitions, Implications and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. Cryptology ePrint Archive, Report 2004/035, 2009. Updated version of original 2004 paper. Accessed 1 December 2016.
URL `http://eprint.iacr.org/2004/035.pdf`

Rossi, F., van Beek, P., and Walsh, T. Handbook of Constraint Programming. Foundations of Artificial Intelligence. Elsevier Science, 2006. ISBN 9780080463803.

Rothaus, O. S. On "bent" functions. *Journal of Combinatorial Theory, Series A*, 20(3):300–305, May 1976.

Rudell, R. L. and Sangiovanni-Vincentelli, A. Multiple-valued minimization for PLA optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(5):727–750, 1987.

Samajder, S. and Sarkar, P. Fast multiplication of the algebraic normal forms of two boolean functions. In *International Workshop on Coding and Cryptography (WCC 2013)*, pages 373–385. 2013.

Savitch, W. J. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

Schaefer, T. J. The complexity of satisfiability problems. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, pages 216–226. ACM, 1978.

Schnorr, C. P. The multiplicative complexity of boolean functions. In **Mora, T.**, editor, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 357 of *Lecture Notes in Computer Science*, pages 45–58. Springer-Verlag, 1989. ISBN 3540510834.

Schulte, C., Tack, G., and Lagerkvist, M. Z. Modeling. In **Schulte, C., Tack, G., and Lagerkvist, M. Z.**, editors, *Modeling and Programming with Gecode*. 2016. Corresponds to Gecode 5.0.0.

Seberry, J. and Zhang, X.-M. Highly nonlinear 0–1 balanced Boolean functions satisfying strict avalanche criterion. In *Advances in Cryptology (AUSCRYPT'92)*, pages 143–155. Springer, 1993.

Seberry, J., Zhang, X.-M., and Zheng, Y. Nonlinearly Balanced Boolean Functions and Their Propagation Characteristics. In *Advances in Cryptology (CRYPTO '93)*, pages 49–60. Springer, 1994.

Selman, B., Kautz, H., Cohen, B. *et al.* Local search strategies for satisfiability testing. *Cliques, coloring, and satisfiability: Second DIMACS implementation challenge*, 26:521–532, 1993.

Shannon, C. E. A mathematical theory of communication. *Bell System Technical Journal*, 4:379–423, 1948.

Shannon, C. E. Communication theory of secrecy systems. *Bell System Technical Journal*, 28(4):656–715, 1949a.

Shannon, C. E. The synthesis of two-terminal switching circuits. *Bell System Technical Journal*, 28(1):59–98, 1949b.

Siegenthaler, T. Correlation-immunity of nonlinear combining functions for cryptographic applications (Corresp.). *IEEE Transactions on Information Theory*, 30(5):776–780, September 1984.

Sieling, D. The nonapproximability of OBDD minimization. *Information and Computation*, 172(2):103–138, 2002.

Somenzi, F. Efficient manipulation of decision diagrams. *International Journal on Software Tools for Technology Transfer*, 3(2):171–181, 2001.

Somenzi, F. CUDD: CU Decision Diagram Package Release 3.0.0. 2015.

Soos, M. and Lindauer, M. The CryptoMiniSat-4.4 set of solvers at the SAT Race 2015. *SAT Race*, 2015.

Staffelbach, O. and Meier, W. Cryptographic significance of the carry for ciphers based on integer addition. In *Advances in Cryptology (CRYPT0 '90)*, pages 602–614. Springer, 1991.

Steube, J. Exploiting a SHA1 weakness in password cracking. In *Passwords^12*. 2004.

Stevens, M. New collision attacks on SHA-1 based on optimal joint local-collision analysis. In *Advances in Cryptology (EUROCRYPT 2013)*, pages 245–261. Springer, 2013.

Stevens, M., Bursztein, E., Karpman, P., Albertini, A., and Markov, Y. The first collision for full sha-1. *IACR Cryptology ePrint Archive*, 2017:190, 2017.

Stevens, M., Karpman, P., and Peyrin, T. Freestart collision for full sha-1. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 459–483. Springer, 2016.

Sung, S. H., Chee, S., and Park, C. Global avalanche characteristics and propagation criterion of balanced Boolean functions. *Information Processing Letters*, 69(1):21–24, January 1999.

Tack, G. Constraint propagation - models, techniques, implementation. Ph.D. thesis, Department of Computer Science, Saarland University, Germany, 2009.

Tseitin, G. S. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983.

Umans, C. The minimum equivalent DNF problem and shortest implicants. In *39th Annual Symposium on Foundations of Computer Science*, pages 556–563. IEEE, 1998.

van der Hoeven, J. and Lecerf, G. On the bit-complexity of sparse polynomial and series multiplication. *Journal of Symbolic Computation*, 50:227–254, 2013. ISSN 0747-7171.

Van Harmelen, F., Lifschitz, V., and Porter, B. Handbook of knowledge representation, volume 1. Elsevier, 2008.

Velichkov, V., Mouha, N., De Canniere, C., and Preneel, B. The additive differential probability of ARX. In *Fast Software Encryption*, pages 342–358. Springer, 2011.

Velichkov, V., Mouha, N., De Cannière, C., and Preneel, B. UNAF: a special set of additive differences with application to the differential analysis of ARX. In *Fast Software Encryption*, pages 287–305. Springer, 2012.

Wachter, M. and Haenni, R. Propositional DAGs: A New Graph-Based Language for Representing Boolean Functions. In *10th International Conference on Principles of Knowledge Representation and Reasoning*, pages 277–285. 2006a.

Wachter, M. and Haenni, R. Representing Boolean Functions with Propositional Directed Acyclic Graphs. In *Workshop on Inference Methods based on Graphical Structures of Knowledge (WIGSK '06)*, page 31. 2006b.

Wagner, D. The boomerang attack. In *Fast Software Encryption*, pages 156–170. Springer, 1999.

Wang, X., Lai, X., Feng, D., Chen, H., and Yu, X. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–18. Springer, 2005a.

Wang, X., Yin, Y. L., and Yu, H. Finding collisions in the full SHA-1. In *Advances in Cryptology (CRYPTO 2005)*, pages 17–36. Springer, 2005b.

Wang, X. and Yu, H. How to break MD5 and other hash functions. In *Advances in Cryptology (EUROCRYPT 2005)*, pages 19–35. Springer, 2005.

Wang, X., Yu, H., and Yin, Y. L. Efficient collision search attacks on SHA-0. In *Advances in Cryptology (CRYPTO 2005)*, pages 1–16. Springer, 2005c.

Webster, A. F. and Tavares, S. E. On the Design of S-Boxes. In *Lecture Notes in Computer Science*, pages 523–534. Springer Science + Business Media, 1986.

Wegener, I. Efficient data structures for Boolean functions. *Discrete Mathematics*, 136(1):347–372, 1994.

Wikipedia. SHA-1 — Wikipedia, The Free Encyclopedia. 2014. Accessed 20 August 2014.
URL `http://en.wikipedia.org/w/index.php?title=SHA-1&oldid=622023521`

Winternitz, R. S. A Secure One-Way Hash Function Built from DES. In *IEEE Symposium on Security and Privacy*, pages 88–90. 1984.

Xiao, G.-Z. and Massey, J. L. A spectral characterization of correlation-immune combining functions. *IEEE Transactions on Information Theory*, 34(3):569–571, 1988.

Xu, L., Hutter, F., Hoos, H. H., and Leyton-Brown, K. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, pages 565–606, 2008.

Zhang, X.-M. and Zheng, Y. Characterizing the structures of cryptographic functions satisfying the propagation criterion for almost all vectors. *Designs, Codes and Cryptography*, 7(1-2):111–134, January 1996.

Zhegalkin, I. I. On a technique of evaluation of propositions in symbolic logic. *Matematisheskii Sbornik*, 34(1):9–27, 1927.

Appendix A

Bitpattern terms

The 2-tuples in this table give the *baseEquations* input for Algorithm 4.2.

Index	Tuples
0	0,1; 2,1; 8,1; 13,1
1	1,1; 3,1; 9,1; 14,1
2	2,1; 4,1; 10,1; 15,1
3	0,2; 2,2; 3,1; 5,1; 8,2; 11,1; 13,2
4	1,2; 3,2; 4,1; 6,1; 9,2; 12,1; 14,2
5	2,2; 4,2; 5,1; 7,1; 10,2; 13,1; 15,2
6	0,3; 2,3; 3,2; 5,2; 6,1; 8,1; 8,3; 11,2; 13,3; 14,1
7	1,3; 3,3; 4,2; 6,2; 7,1; 9,1; 9,3; 12,2; 14,3; 15,1
8	0,2; 2,2; 2,3; 4,3; 5,2; 7,2; 8,1; 8,2; 10,1; 10,3; 15,3
9	0,4; 1,2; 2,4; 3,2; 3,3; 5,3; 6,2; 8,2; 8,4; 9,1; 9,2; 11,1; 11,3; 13,4
10	1,4; 2,2; 3,4; 4,2; 4,3; 6,3; 7,2; 9,2; 9,4; 10,1; 10,2; 12,1; 12,3; 14,4
11	2,4; 3,2; 4,4; 5,2; 5,3; 7,3; 8,2; 10,2; 10,4; 11,1; 11,2; 13,1; 13,3; 15,4
12	0,5; 2,5; 3,4; 4,2; 5,4; 6,2; 6,3; 8,3; 8,5; 9,2; 11,2; 11,4; 12,1; 12,2; 13,5; 14,1; 14,3
13	1,5; 3,5; 4,4; 5,2; 6,4; 7,2; 7,3; 9,3; 9,5; 10,2; 12,2; 12,4; 13,1; 13,2; 14,5; 15,1; 15,3
14	0,2; 0,4; 2,2; 2,4; 2,5; 4,5; 5,4; 6,2; 7,4; 8,3; 8,4; 10,3; 10,5; 11,2; 14,1; 14,2; 15,5
15	0,6; 1,2; 1,4; 2,6; 3,2; 3,4; 3,5; 5,5; 6,4; 7,2; 8,4; 8,6; 9,3; 9,4; 11,3; 11,5; 12,2; 13,6; 15,1; 15,2
16	0,2; 0,3; 1,6; 2,3; 2,4; 3,6; 4,2; 4,4; 4,5; 6,5; 7,4; 8,3; 9,4; 9,6; 10,3; 10,4; 12,3; 12,5; 13,3; 14,6

Index	Tuples
17	1,2; 1,3; 2,6; 3,3; 3,4; 4,6; 5,2; 5,4; 5,5; 7,5; 8,4; 9,3; 10,4; 10,6; 11,3; 11,4; 13,3; 13,5; 14,3; 15,6
18	0,7; 2,2; 2,3; 2,7; 3,6; 4,3; 4,4; 5,6; 6,2; 6,4; 6,5; 8,5; 8,7; 9,4; 10,3; 11,4; 11,6; 12,3; 12,4; 13,7; 14,3; 14,5; 15,3
19	0,4; 1,7; 2,4; 3,2; 3,3; 3,7; 4,6; 5,3; 5,4; 6,6; 7,2; 7,4; 7,5; 8,4; 9,5; 9,7; 10,4; 11,3; 12,4; 12,6; 13,3; 14,7; 15,3; 15,5
20	0,4; 0,6; 1,4; 2,4; 2,6; 2,7; 3,4; 4,2; 4,3; 4,7; 5,6; 6,3; 6,4; 7,6; 8,2; 8,5; 8,6; 9,4; 10,5; 10,7; 11,4; 12,3; 14,3; 15,7
21	0,8; 1,4; 1,6; 2,4; 2,8; 3,4; 3,6; 3,7; 4,4; 5,2; 5,3; 5,7; 6,6; 7,3; 7,4; 8,6; 8,8; 9,2; 9,5; 9,6; 10,4; 11,5; 11,7; 12,4; 13,3; 13,8; 15,3
22	0,4; 1,8; 2,6; 3,4; 3,8; 4,4; 4,6; 4,7; 5,4; 6,2; 6,3; 6,7; 7,6; 8,3; 9,6; 9,8; 10,2; 10,5; 10,6; 11,4; 12,5; 12,7; 14,3; 14,8
23	1,4; 2,8; 3,6; 4,4; 4,8; 5,4; 5,6; 5,7; 6,4; 7,2; 7,3; 7,7; 8,6; 9,3; 10,6; 10,8; 11,2; 11,5; 11,6; 12,4; 13,5; 13,7; 15,3; 15,8
24	0,4; 0,9; 2,9; 3,8; 4,6; 5,4; 5,8; 6,4; 6,6; 6,7; 7,4; 8,2; 8,3; 8,4; 8,7; 8,9; 9,6; 10,3; 11,6; 11,8; 12,2; 12,5; 12,6; 13,9; 14,5; 14,7
25	1,4; 1,9; 3,9; 4,8; 5,6; 6,4; 6,8; 7,4; 7,6; 7,7; 8,4; 9,2; 9,3; 9,4; 9,7; 9,9; 10,6; 11,3; 12,6; 12,8; 13,2; 13,5; 13,6; 14,9; 15,5; 15,7
26	0,6; 0,8; 2,4; 2,6; 2,8; 2,9; 4,9; 5,8; 6,6; 7,4; 7,8; 8,4; 8,7; 8,8; 9,4; 10,2; 10,3; 10,4; 10,7; 10,9; 11,6; 12,3; 14,2; 14,5; 14,6; 15,9
27	0,10; 1,6; 1,8; 2,10; 3,4; 3,6; 3,8; 3,9; 5,9; 6,8; 7,6; 8,4; 8,8; 8,10; 9,4; 9,7; 9,8; 10,4; 11,2; 11,3; 11,4; 11,7; 11,9; 12,6; 13,3; 13,10; 15,2; 15,5; 15,6
28	0,3; 0,6; 0,7; 1,10; 2,3; 2,7; 2,8; 3,10; 4,4; 4,6; 4,8; 4,9; 6,9; 7,8; 8,3; 8,7; 9,4; 9,8; 9,10; 10,4; 10,7; 10,8; 11,4; 12,2; 12,3; 12,4; 12,7; 12,9; 13,3; 13,7; 14,3; 14,10
29	1,3; 1,6; 1,7; 2,10; 3,3; 3,7; 3,8; 4,10; 5,4; 5,6; 5,8; 5,9; 7,9; 8,8; 9,3; 9,7; 10,4; 10,8; 10,10; 11,4; 11,7; 11,8; 12,4; 13,2; 13,3; 13,4; 13,7; 13,9; 14,3; 14,7; 15,3; 15,10
30	0,4; 0,11; 2,3; 2,4; 2,6; 2,7; 2,11; 3,10; 4,3; 4,7; 4,8; 5,10; 6,4; 6,6; 6,8; 6,9; 8,4; 8,9; 8,11; 9,8; 10,3; 10,7; 11,4; 11,8; 11,10; 12,4; 12,7; 12,8; 13,11; 14,2; 14,3; 14,4; 14,7; 14,9; 15,3; 15,7
31	0,4; 0,8; 1,4; 1,11; 2,4; 2,8; 3,3; 3,4; 3,6; 3,7; 3,11; 4,10; 5,3; 5,7; 5,8; 6,10; 7,4; 7,6; 7,8; 7,9; 8,4; 8,8; 9,4; 9,9; 9,11; 10,8; 11,3; 11,7; 12,4; 12,8; 12,10; 13,7; 14,11; 15,2; 15,3; 15,4; 15,7; 15,9

Index	Tuples
32	0,3; 0,4; 0,5; 0,8; 0,10; 1,4; 1,8; 2,3; 2,5; 2,8; 2,10; 2,11; 3,4; 3,8; 4,3; 4,4; 4,6; 4,7; 4,11; 5,10; 6,3; 6,7; 6,8; 7,10; 8,3; 8,5; 8,6; 8,9; 8,10; 9,4; 9,8; 10,4; 10,9; 10,11; 11,8; 12,3; 12,7; 13,3; 13,5; 14,7; 15,11
33	0,12; 1,3; 1,4; 1,5; 1,8; 1,10; 2,4; 2,8; 2,12; 3,3; 3,5; 3,8; 3,10; 3,11; 4,4; 4,8; 5,3; 5,4; 5,6; 5,7; 5,11; 6,10; 7,3; 7,7; 7,8; 8,10; 8,12; 9,3; 9,5; 9,6; 9,9; 9,10; 10,4; 10,8; 11,4; 11,9; 11,11; 12,8; 13,3; 13,7; 13,12; 14,3; 14,5; 15,7
34	0,8; 1,12; 2,3; 2,4; 2,5; 2,10; 3,4; 3,8; 3,12; 4,3; 4,5; 4,8; 4,10; 4,11; 5,4; 5,8; 6,3; 6,4; 6,6; 6,7; 6,11; 7,10; 8,3; 8,7; 9,10; 9,12; 10,3; 10,5; 10,6; 10,9; 10,10; 11,4; 11,8; 12,4; 12,9; 12,11; 14,3; 14,7; 14,12; 15,3; 15,5
35	0,4; 0,6; 1,8; 2,4; 2,6; 2,12; 3,3; 3,4; 3,5; 3,10; 4,4; 4,8; 4,12; 5,3; 5,5; 5,8; 5,10; 5,11; 6,4; 6,8; 7,3; 7,4; 7,6; 7,7; 7,11; 8,4; 8,6; 8,10; 9,3; 9,7; 10,10; 10,12; 11,3; 11,5; 11,6; 11,9; 11,10; 12,4; 12,8; 13,6; 13,9; 13,11; 15,3; 15,7; 15,12
36	0,4; 0,8; 0,13; 1,4; 1,6; 2,4; 2,13; 3,4; 3,6; 3,12; 4,3; 4,4; 4,5; 4,10; 5,4; 5,8; 5,12; 6,3; 6,5; 6,8; 6,10; 6,11; 7,4; 7,8; 8,3; 8,6; 8,7; 8,8; 8,11; 8,13; 9,4; 9,6; 9,10; 10,3; 10,7; 11,10; 11,12; 12,3; 12,5; 12,6; 12,9; 12,10; 13,13; 14,6; 14,9; 14,11
37	1,4; 1,8; 1,13; 2,4; 2,6; 3,4; 3,13; 4,4; 4,6; 4,12; 5,3; 5,4; 5,5; 5,10; 6,4; 6,8; 6,12; 7,3; 7,5; 7,8; 7,10; 7,11; 8,4; 8,8; 9,3; 9,6; 9,7; 9,8; 9,11; 9,13; 10,4; 10,6; 10,10; 11,3; 11,7; 12,10; 12,12; 13,3; 13,5; 13,6; 13,9; 13,10; 14,13; 15,6; 15,9; 15,11
38	0,7; 0,10; 0,12; 2,4; 2,7; 2,8; 2,10; 2,12; 2,13; 3,4; 3,6; 4,4; 4,13; 5,4; 5,6; 5,12; 6,3; 6,4; 6,5; 6,10; 7,4; 7,8; 7,12; 8,3; 8,5; 8,7; 8,8; 8,11; 8,12; 9,4; 9,8; 10,3; 10,6; 10,7; 10,8; 10,11; 10,13; 11,4; 11,6; 11,10; 12,3; 12,7; 13,7; 14,3; 14,5; 14,6; 14,9; 14,10; 15,13
39	0,14; 1,7; 1,10; 1,12; 2,14; 3,4; 3,7; 3,8; 3,10; 3,12; 3,13; 4,4; 4,6; 5,4; 5,13; 6,4; 6,6; 6,12; 7,3; 7,4; 7,5; 7,10; 8,4; 8,8; 8,12; 8,14; 9,3; 9,5; 9,7; 9,8; 9,11; 9,12; 10,4; 10,8; 11,3; 11,6; 11,7; 11,8; 11,11; 11,13; 12,4; 12,6; 12,10; 13,3; 13,7; 13,14; 14,7; 15,3; 15,5; 15,6; 15,9; 15,10
40	0,4; 0,6; 0,7; 0,10; 0,11; 1,14; 2,4; 2,6; 2,11; 2,12; 3,14; 4,4; 4,7; 4,8; 4,10; 4,12; 4,13; 5,4; 5,6; 6,4; 6,13; 7,4; 7,6; 7,12; 8,3; 8,5; 8,6; 8,7; 8,11; 9,4; 9,8; 9,12; 9,14; 10,3; 10,5; 10,7; 10,8; 10,11; 10,12; 11,4; 11,8; 12,3; 12,6; 12,7; 12,8; 12,11; 12,13; 13,7; 13,11; 14,3; 14,7; 14,14; 15,7
41	0,8; 1,4; 1,6; 1,7; 1,10; 1,11; 2,8; 2,14; 3,4; 3,6; 3,11; 3,12; 4,14; 5,4; 5,7; 5,8; 5,10; 5,12; 5,13; 6,4; 6,6; 7,4; 7,13; 8,4; 8,6; 8,8; 8,12; 9,3; 9,5; 9,6; 9,7; 9,11; 10,4; 10,8; 10,12; 10,14; 11,3; 11,5; 11,7; 11,8; 11,11; 11,12; 12,4; 12,8; 13,3; 13,6; 13,7; 13,11; 13,13; 14,7; 14,11; 15,3; 15,7; 15,14

Index	Tuples
42	0,4; 0,8; 0,15; 1,8; 2,6; 2,7; 2,8; 2,10; 2,11; 2,15; 3,8; 3,14; 4,4; 4,6; 4,11; 4,12; 5,14; 6,4; 6,7; 6,8; 6,10; 6,12; 6,13; 7,4; 7,6; 8,8; 8,13; 8,15; 9,4; 9,6; 9,8; 9,12; 10,3; 10,5; 10,6; 10,7; 10,11; 11,4; 11,8; 11,12; 11,14; 12,3; 12,5; 12,7; 12,8; 12,11; 12,12; 13,15; 14,3; 14,6; 14,7; 14,11; 14,13; 15,7; 15,11
43	0,8; 0,12; 1,4; 1,8; 1,15; 2,12; 3,6; 3,7; 3,8; 3,10; 3,11; 3,15; 4,8; 4,14; 5,4; 5,6; 5,11; 5,12; 6,14; 7,4; 7,7; 7,8; 7,10; 7,12; 7,13; 8,4; 8,6; 8,8; 8,12; 9,8; 9,13; 9,15; 10,4; 10,6; 10,8; 10,12; 11,3; 11,5; 11,6; 11,7; 11,11; 12,4; 12,8; 12,12; 12,14; 13,3; 13,5; 13,7; 13,11; 14,15; 15,3; 15,6; 15,7; 15,11; 15,13
44	0,4; 0,7; 0,8; 0,12; 0,14; 1,8; 1,12; 2,7; 2,12; 2,14; 2,15; 3,12; 4,6; 4,7; 4,8; 4,10; 4,11; 4,15; 5,8; 5,14; 6,4; 6,6; 6,11; 6,12; 7,14; 8,10; 8,13; 8,14; 9,4; 9,6; 9,8; 9,12; 10,8; 10,13; 10,15; 11,4; 11,6; 11,8; 11,12; 12,3; 12,5; 12,6; 12,7; 12,11; 13,7; 14,3; 14,5; 14,7; 14,11; 15,15
45	0,16; 1,4; 1,7; 1,8; 1,12; 1,14; 2,8; 2,12; 2,16; 3,7; 3,12; 3,14; 3,15; 4,12; 5,6; 5,7; 5,8; 5,10; 5,11; 5,15; 6,8; 6,14; 7,4; 7,6; 7,11; 7,12; 8,14; 8,16; 9,10; 9,13; 9,14; 10,4; 10,6; 10,8; 10,12; 11,8; 11,13; 11,15; 12,4; 12,6; 12,8; 12,12; 13,3; 13,5; 13,6; 13,7; 13,11; 13,16; 14,7; 15,3; 15,5; 15,7; 15,11
46	0,4; 0,6; 0,8; 0,12; 1,16; 2,6; 2,7; 2,14; 3,8; 3,12; 3,16; 4,7; 4,12; 4,14; 4,15; 5,12; 6,6; 6,7; 6,8; 6,10; 6,11; 6,15; 7,8; 7,14; 8,8; 8,11; 9,14; 9,16; 10,10; 10,13; 10,14; 11,4; 11,6; 11,8; 11,12; 12,8; 12,13; 12,15; 14,3; 14,5; 14,6; 14,7; 14,11; 14,16; 15,7
47	0,8; 1,4; 1,6; 1,8; 1,12; 2,8; 2,16; 3,6; 3,7; 3,14; 4,8; 4,12; 4,16; 5,7; 5,12; 5,14; 5,15; 6,12; 7,6; 7,7; 7,8; 7,10; 7,11; 7,15; 8,14; 9,8; 9,11; 10,14; 10,16; 11,10; 11,13; 11,14; 12,4; 12,6; 12,8; 12,12; 13,13; 13,15; 15,3; 15,5; 15,6; 15,7; 15,11; 15,16
48	0,4; 0,6; 0,7; 0,8; 0,12; 0,17; 1,8; 2,7; 2,17; 3,8; 3,16; 4,6; 4,7; 4,14; 5,8; 5,12; 5,16; 6,7; 6,12; 6,14; 6,15; 7,12; 8,4; 8,10; 8,11; 8,12; 8,15; 8,17; 9,14; 10,8; 10,11; 11,14; 11,16; 12,10; 12,13; 12,14; 13,7; 13,17; 14,13; 14,15
49	1,4; 1,6; 1,7; 1,8; 1,12; 1,17; 2,8; 3,7; 3,17; 4,8; 4,16; 5,6; 5,7; 5,14; 6,8; 6,12; 6,16; 7,7; 7,12; 7,14; 7,15; 8,12; 9,4; 9,10; 9,11; 9,12; 9,15; 9,17; 10,14; 11,8; 11,11; 12,14; 12,16; 13,10; 13,13; 13,14; 14,7; 14,17; 15,13; 15,15
50	0,14; 0,16; 2,4; 2,6; 2,7; 2,8; 2,12; 2,14; 2,16; 2,17; 3,8; 4,7; 4,17; 5,8; 5,16; 6,6; 6,7; 6,14; 7,8; 7,12; 7,16; 8,7; 8,12; 8,15; 8,16; 9,12; 10,4; 10,10; 10,11; 10,12; 10,15; 10,17; 11,14; 12,8; 12,11; 14,10; 14,13; 14,14; 15,7; 15,17
51	0,8; 0,18; 1,14; 1,16; 2,8; 2,18; 3,4; 3,6; 3,7; 3,8; 3,12; 3,14; 3,16; 3,17; 4,8; 5,7; 5,17; 6,8; 6,16; 7,6; 7,7; 7,14; 8,12; 8,16; 8,18; 9,7; 9,12; 9,15; 9,16; 10,12; 11,4; 11,10; 11,11; 11,12; 11,15; 11,17; 12,14; 13,11; 13,18; 15,10; 15,13; 15,14

Index	Tuples
52	0,11; 0,14; 0,15; 1,8; 1,18; 2,11; 2,15; 2,16; 3,8; 3,18; 4,4; 4,6; 4,7; 4,8; 4,12; 4,14; 4,16; 4,17; 5,8; 6,7; 6,17; 7,8; 7,16; 8,6; 8,7; 8,11; 8,15; 9,12; 9,16; 9,18; 10,7; 10,12; 10,15; 10,16; 11,12; 12,4; 12,10; 12,11; 12,12; 12,15; 12,17; 13,11; 13,15; 14,11; 14,18
53	1,11; 1,14; 1,15; 2,8; 2,18; 3,11; 3,15; 3,16; 4,8; 4,18; 5,4; 5,6; 5,7; 5,8; 5,12; 5,14; 5,16; 5,17; 6,8; 7,7; 7,17; 8,8; 8,16; 9,6; 9,7; 9,11; 9,15; 10,12; 10,16; 10,18; 11,7; 11,12; 11,15; 11,16; 12,12; 13,4; 13,10; 13,11; 13,12; 13,15; 13,17; 14,11; 14,15; 15,11; 15,18
54	0,12; 0,19; 2,11; 2,12; 2,14; 2,15; 2,19; 3,8; 3,18; 4,11; 4,15; 4,16; 5,8; 5,18; 6,4; 6,6; 6,7; 6,8; 6,12; 6,14; 6,16; 6,17; 7,8; 8,7; 8,12; 8,17; 8,19; 9,8; 9,16; 10,6; 10,7; 10,11; 10,15; 11,12; 11,16; 11,18; 12,7; 12,12; 12,15; 12,16; 13,19; 14,4; 14,10; 14,11; 14,12; 14,15; 14,17; 15,11; 15,15
55	0,12; 0,16; 1,12; 1,19; 2,12; 2,16; 3,11; 3,12; 3,14; 3,15; 3,19; 4,8; 4,18; 5,11; 5,15; 5,16; 6,8; 6,18; 7,4; 7,6; 7,7; 7,8; 7,12; 7,14; 7,16; 7,17; 8,8; 8,12; 8,16; 9,7; 9,12; 9,17; 9,19; 10,8; 10,16; 11,6; 11,7; 11,11; 11,15; 12,12; 12,16; 12,18; 13,7; 13,15; 14,19; 15,4; 15,10; 15,11; 15,12; 15,15; 15,17
56	0,5; 0,11; 0,12; 0,13; 0,16; 0,18; 1,12; 1,16; 2,5; 2,11; 2,13; 2,16; 2,18; 2,19; 3,12; 3,16; 4,11; 4,12; 4,14; 4,15; 4,19; 5,8; 5,18; 6,11; 6,15; 6,16; 7,8; 7,18; 8,4; 8,5; 8,6; 8,7; 8,8; 8,11; 8,13; 8,14; 8,17; 8,18; 9,8; 9,12; 9,16; 10,7; 10,12; 10,17; 10,19; 11,8; 11,16; 12,6; 12,7; 12,11; 12,15; 13,5; 13,11; 13,13; 14,7; 14,15; 15,19
57	0,20; 1,5; 1,11; 1,12; 1,13; 1,16; 1,18; 2,12; 2,16; 2,20; 3,5; 3,11; 3,13; 3,16; 3,18; 3,19; 4,12; 4,16; 5,11; 5,12; 5,14; 5,15; 5,19; 6,8; 6,18; 7,11; 7,15; 7,16; 8,8; 8,18; 8,20; 9,4; 9,5; 9,6; 9,7; 9,8; 9,11; 9,13; 9,14; 9,17; 9,18; 10,8; 10,12; 10,16; 11,7; 11,12; 11,17; 11,19; 12,8; 12,16; 13,6; 13,7; 13,11; 13,15; 13,20; 14,5; 14,11; 14,13; 15,7; 15,15
58	0,8; 0,16; 1,20; 2,5; 2,8; 2,11; 2,12; 2,13; 2,18; 3,12; 3,16; 3,20; 4,5; 4,11; 4,13; 4,16; 4,18; 4,19; 5,12; 5,16; 6,11; 6,12; 6,14; 6,15; 6,19; 7,8; 7,18; 8,8; 8,11; 8,15; 9,8; 9,18; 9,20; 10,4; 10,5; 10,6; 10,7; 10,8; 10,11; 10,13; 10,14; 10,17; 10,18; 11,8; 11,12; 11,16; 12,7; 12,12; 12,17; 12,19; 14,6; 14,7; 14,11; 14,15; 14,20; 15,5; 15,11; 15,13
59	0,6; 0,12; 0,14; 1,8; 1,16; 2,6; 2,12; 2,14; 2,20; 3,5; 3,8; 3,11; 3,12; 3,13; 3,18; 4,12; 4,16; 4,20; 5,5; 5,11; 5,13; 5,16; 5,18; 5,19; 6,12; 6,16; 7,11; 7,12; 7,14; 7,15; 7,19; 8,6; 8,8; 8,12; 8,14; 8,18; 9,8; 9,11; 9,15; 10,8; 10,18; 10,20; 11,4; 11,5; 11,6; 11,7; 11,8; 11,11; 11,13; 11,14; 11,17; 11,18; 12,8; 12,12; 12,16; 13,6; 13,7; 13,14; 13,17; 13,19; 15,6; 15,7; 15,11; 15,15; 15,20

Index	Tuples
60	0,7; 0,8; 0,12; 0,16; 0,21; 1,6; 1,12; 1,14; 2,7; 2,12; 2,21; 3,6; 3,12; 3,14; 3,20; 4,5; 4,8; 4,11; 4,12; 4,13; 4,18; 5,12; 5,16; 5,20; 6,5; 6,11; 6,13; 6,16; 6,18; 6,19; 7,12; 7,16; 8,7; 8,8; 8,11; 8,14; 8,15; 8,16; 8,19; 8,21; 9,6; 9,8; 9,12; 9,14; 9,18; 10,8; 10,11; 10,15; 11,8; 11,18; 11,20; 12,4; 12,5; 12,6; 12,7; 12,8; 12,11; 12,13; 12,14; 12,17; 12,18; 13,7; 13,21; 14,6; 14,7; 14,14; 14,17; 14,19
61	1,7; 1,8; 1,12; 1,16; 1,21; 2,6; 2,12; 2,14; 3,7; 3,12; 3,21; 4,6; 4,12; 4,14; 4,20; 5,5; 5,8; 5,11; 5,12; 5,13; 5,18; 6,12; 6,16; 6,20; 7,5; 7,11; 7,13; 7,16; 7,18; 7,19; 8,12; 8,16; 9,7; 9,8; 9,11; 9,14; 9,15; 9,16; 9,19; 9,21; 10,6; 10,8; 10,12; 10,14; 10,18; 11,8; 11,11; 11,15; 12,8; 12,18; 12,20; 13,4; 13,5; 13,6; 13,7; 13,8; 13,11; 13,13; 13,14; 13,17; 13,18; 14,7; 14,21; 15,6; 15,7; 15,14; 15,17; 15,19
62	0,7; 0,8; 0,15; 0,18; 0,20; 2,12; 2,15; 2,16; 2,18; 2,20; 2,21; 3,6; 3,12; 3,14; 4,7; 4,12; 4,21; 5,6; 5,12; 5,14; 5,20; 6,5; 6,8; 6,11; 6,12; 6,13; 6,18; 7,12; 7,16; 7,20; 8,5; 8,7; 8,8; 8,11; 8,13; 8,15; 8,16; 8,19; 8,20; 9,12; 9,16; 10,7; 10,8; 10,11; 10,14; 10,15; 10,16; 10,19; 10,21; 11,6; 11,8; 11,12; 11,14; 11,18; 12,8; 12,11; 12,15; 13,7; 13,15; 14,4; 14,5; 14,6; 14,7; 14,8; 14,11; 14,13; 14,14; 14,17; 14,18; 15,7; 15,21
63	0,8; 0,22; 1,7; 1,8; 1,15; 1,18; 1,20; 2,8; 2,22; 3,12; 3,15; 3,16; 3,18; 3,20; 3,21; 4,6; 4,12; 4,14; 5,7; 5,12; 5,21; 6,6; 6,12; 6,14; 6,20; 7,5; 7,8; 7,11; 7,12; 7,13; 7,18; 8,8; 8,12; 8,16; 8,20; 8,22; 9,5; 9,7; 9,8; 9,11; 9,13; 9,15; 9,16; 9,19; 9,20; 10,12; 10,16; 11,7; 11,8; 11,11; 11,14; 11,15; 11,16; 11,19; 11,21; 12,6; 12,8; 12,12; 12,14; 12,18; 13,11; 13,15; 13,22; 14,7; 14,15; 15,4; 15,5; 15,6; 15,7; 15,8; 15,11; 15,13; 15,14; 15,17; 15,18