CUBE TESTERS ON HASH FUNCTIONS

By

Muhammad Owais ul Haq

A thesis submitted to the Faculty of Information Security Department, National University of Sciences and Technology, Pakistan in partial fulfillment for the requirements of MS in Information Security

JULY 2014

ABSTRACT

Hashing Algorithms are of prime importance these days due to their implementation in applications requiring data authentication and data forgery detection and prevention. Without safeguards such as those offered by hash functions, data would be extremely vulnerable to attacks which can alter or make changes in it. 'Cube Testers' Attack is a latest technique which is being utilized to test different cryptosystems, both ciphers and hashes for non-randomness behavior. The attack distinguishes a family of functions from random functions using some testable properties like Balanced-ness, Constant-ness, presence of Low Degree, Linear Variables and presence of Neutral Variables. As pseudorandom object has low correlation with a structured object so it gives distinguishers that help finding nonrandomness in the hashes. Implementation of Cube testers is possible with black box access that is it can be independent of internal structure.

SHA-256 is the standard hash function recommended by NIST in 2002. It has been tested against different cryptanalytic techniques including collision, pseudo collision, preimage and second preimage attacks. It has been tested against Cube Testers for the first time. SHA-256 is tested against three properties using Cube Testers, Balance-ness, Impedance and Off By One. Reduced version of SHA-256 with 25 steps out of 64 steps has shown non-random behavior against Balance Test. Complete 64 step SHA-256 is found vulnerable against Impedance Test. Off By One test could not find any weakness against SHA-256.

DEDICATION

All praise and thanks to almighty Allah, the most Gracious and the most Compassionate, Master of the Day of Judgment.

I dedicate my work to my family and my teachers who have been a constant source of encouragement and guidance for me throughout my work.

ACKNOWLEDGEMENT

First of all, I would like to thank to Almighty Allah Who has blessed me with intellect and strength, Who increased my knowledge and made me able to successfully go through this phase and complete my work.

The first and the foremost person who I would like to acknowledge is my supervisor Dr. Mehreen Afzal. Despite of her busy calendar, she was always available for discussions and queries. She continuously kept me in the right direction by giving valuable suggestions. She used to ask for my progress time by time and after understanding my advancements, gave me appreciated ideas and recommendations. Without her constant support, help and encouragement, timely completion of my research work would have been impossible.

I am also immensely thankful to my guidance committee members Dr. Adil Masood Siddiqui, Lecturer Sharjeel Riaz and Lecturer Mian Waseem Iqbal for their persistent support and inspiration.

I would also like to thank J. P. Aumasson, the author of Cube Attack and Cube Testers Attack who clarified my queries through e-mails. Thanks to Alan Kaminsky who helped me in the understanding of the cube attack through emails.

I am also thankful to my Head of Department Dr. Babar and his team for administrative help and support.

I am immeasurably thankful to my family members who took care of me throughout my MS studies.

TABLE OF CONTENTS

LIST OF FIGURES

Figure No. **Page No.**

LIST OF TABLES

Table No. **Page No.**

KEY TO ABBREVIATIONS

SHA	Secure Hash Algorithm
PC	Personal Computer
MAC	Message Authentication Code
XOR	Exclusive OR Operation
ANF	Algebraic Normal Form
FPGA	Field-Programmable Gate Array
CV	Cube Vector
SV	Superpoly Vector
MD	Message Digest
C	Size of Cube Input
S	Size of Superpoly Input
Q	Superpoly Input
E	Estimated number of samples in a catagory
m	Total number of samples
$M^{(i)}$	I^{th} block of Input message
ROTR	Rotate Right
SHR	Shift Right

Chapter 1

Introduction

1.1 Overview

Communication systems in the contemporary world continue to grow and evolve. Integrity, confidentiality and authenticity have been big concerns in communications systems. Most unkeyed hash functions commonly found in practice were originally designed for the purpose of providing data integrity.

A hash function is any function that can be used to map digital data of arbitrary size to digital data of fixed size, with slight differences in input data producing very big differences in output data. A hash function is considered practically impossible to invert, that is, to recreate the input data from its hash value alone. Hash function should also satisfy the simple uniform hashing assumption -- that the hash function should look random. If it is to look random, this means that any change to a key, even a small one, should change the bucket index in an apparently random way. If we imagine writing the bucket index as a binary number, a small change to the key should randomly flip the bits in the bucket index. This is called information diffusion.

Cryptanalysis may be considered an integral part of design of a cryptographic algorithm. The process continues even after propositions and acceptance of the algorithms. A number of cryptanalytic techniques have emerged so far like differential, linear, impossible differential, integral attack and the related key attack. A new type of cryptanalytic technique named as cube attack has been proposed by Itai Dinur and Adi Shamir in 2009. One variant of cube attack is cube testers attack which combines the cube attack with efficient algebraic property testers, and can be used to mount distinguishers or to detect non-randomness in cryptographic primitives.

Both cube attacks and cube testers require black-box access and target primitives with secret and public variables. Meanwhile, they can be built on low-degree components. Nevertheless, cube testers don't require a low degree function and pre-computation, it is just to satisfy some testable property with significantly higher (or lower) probability than a random function.

Rather than recovering a secret key or otherwise attacking the primitive, cube tester attack probes the primitive's internal polynomial structure and can be used to analyze the primitive's statistical behavior.

1.2 Need for Research

Every cryptographic algorithm needs to be tested for the existing cryptanalytic techniques and the upcoming ones. Without testing, one is not sure whether the cryptosystem is secure or vulnerable to a particular attack. Cube testers attack is a new type of cryptanalytic technique and hash functions should be explored and tested against the cube testers attack. Cube testers attack was applied on MD6 by its authors in the paper in which Cube testers attack was proposed. SHA-256 is 256 bit output version of standard hash selected by NIST.

1.3 Problem Statement

There is a need to test the commonly used hash functions against the cube testers attack as it is a new emerging threat. SHA-256 has not yet been tested against Cube testers attack yet, despite being the standard Hash algorithm nominated by NIST. Some of the hash functions have been tested against property testing using cube testers attack. Cryptanalysis and attacks like differential cryptanalysis, linear cryptansalysis, algebraic cryptanalysis, integral attack and related key attack are in

common practice. The cube testers attack should also be included in the analysis/evaluation of the hash functions claiming to have “random” output.

1.4 Objectives

The objective of this research is to first, carry out a thorough study of the literature on cube testers attack and general property testing to understand the mechanics, and apply cube testers attack to find non random behavior in SHA-256. Then customized code of SHA-256 has been developed in C++ excluding the preprocessing phases in controlling input of hash function at bit level. Then code has been developed for Cube Testers Attack and Property testing which includes Balance test, Impedance test and Off By One test in C++ which was then applied on SHA-256 hash function.

1.5 Research Methodology and Achieved Goals

The research work has been divided into two main phases. In the first phase, detailed study and literature review has been carried out related to the cube testers attack. A strong theoretical concept has been built regarding the working of the cube testers attack and property testing. A detailed study of SHA-256 has also been done to understand its structure and then core function of SHA-256 was implemented in C++ in a customized fashion, excluding the pre-processing steps. In the second phase, the implementation of the cube testers attack on Core function of SHA-256 has been carried out. C++ has been used for the testing. Code for Balance-ness , Impedance and Off By One tests was developed and implemented on SHA-256 core function to check the hash algrorithm against these properties.

The three tests (balance test, impedance test and Off By One test) applied on SHA-256. Balance Test succeeded in finding non-randomness over 25 steps of main

function of SHA-256 out of 64 steps. Impedance test found randomness with complete 64 steps of SHA-256. While Off By One test failed to find any non-randomness as per the criteria defined in Chi Square test.

1.6 Thesis Organization

The thesis report has been divided into six chapters. Chapter 2 contains the literature view of this research in which all the related work found has been briefly discussed. The description of the cube testers attack and property testing is explained in Chapter 3 for the understanding of the concept of cube attack. In Chapter 4 structure of SHA-256 is discussed in detail. Chapter 5 contains the implementation details of Cube Testers attack on SHA-256 and Property Testing results. The properties tested on SHA-256 are Balance test, Off By One test and Impedance test. Chapter 6 concludes the report and suggests the future work.

Chapter 2

Literature Review

2.1 Introduction

In this chapter those hash functions and ciphers are discussed which have already been attacked and tested by the Cube Testers attack. The implementations of cube testers attack developed so far have also been reviewed. Maximum work has been carried out on hash functions MD6, CubeHash and Skein, finalists of the SHA-3 competition[1][4], in literature.

The chapter has been divided into four sections. Section 2.2 contains review of the hash functions that have been attacked by the cube testers attack. Section 2.3 presents the details of attack against stream ciphers. Section 2.4 describes some of the testable statistic properties which have been tested using cube testers attack.

2.2 Cube Testers Attack on Hash Functions

Initial targets of the cube testers attack were to find an efficient way of finding statistical weaknesses in hash functions, which could lead to more serious attacks. In earlier papers of cube testers, MD6, Skein and CubeHash became the most popular target of the cube attack. Trivium a stream cipher has also been tested against the cube attack by Jean-Phillipe Aumasson [4].

2.2.1 MD6

Jean-Phillippe Aumasson, Itai Dinur and Adi Shamir introduced the Cube Testers Attack in 2009 [1] [2]. They showed attack implementation on MD6, which appeared as a Round 1 contestant in SHA-3 competition. It could not proceed to

Round 2 because of its slow algorithm. It was asked to be reduced to 40 rounds from 80 but 40 round MD6 was susceptible to differential cryptanalysis.

The reduced versions of MD6 having 18 and 66 rounds have been attacked by the authors themselves. Cube testers detect imbalance over 18 rounds of MD6 in 2^{17} complexity. Cube testers when applied to a slightly modified version of the MD6 compression function, they can distinguish 66 rounds from random in 2^{24} complexity. Different testable properties were tested against MD6, these results were found while testing the balance of superpolys of MD6.

2.2.2 CubeHash

Alan Kaminsky has applied the cube testers on hash function CubeHash, a candidate of SHA-3 competition [1]. The cube test program results were subjected to the balance test, independence test and off-by-one test. Randomness was found for Off-by-One test only.

This test was applied using parallel computing. 40 dual core CPU's, each with 2.6 GHz clock and 8 GB main memory were used. The test took a total of 3,606,910,695,720 (approx. 2^{42}) CubeHash evaluations. The evaluations took 1.25 * 10^6 seconds.

2.2.3 Skein

Alan Kaminsky has attacked Skein, a SHA-3 candidate in 2010 [1]. The output of Cube Test Program was tested for balanceness, impedance and off-by-one test. Author found randomness for Off-by-One test.

Parallel computing setup was used for this test. 40 dual core CPU's were used, each having 2.6 GHz clock speed and 8 GB main memory. A total of 3,603,992,046,760 evaluations were performed.

2.3 Cube Testers Attack on Ciphers

Stream ciphers as well as Block ciphers have been tested for randomness tests via cube testers. Generally ciphers can be represented with low degrees ANF as compared to hash functions. So it is possible to extract the ANF of ciphers with reduced rounds. Once ANF is found, more properties can be tested against them.

2.3.1 Trivium

Trivium was tested by Jean-Philippe Aumasson in the paper where cube testers were proposed. Trivium takes as input a 80-bit key and a 80-bit Initial Vector and produces a key stream after 1152 rounds of initialization. Each round corresponds to clocking three feedback shift registers, each one having a quadratic feedback polynomial.

Using Cube Testers non-randomness was detected on 885 rounds of Trivium and verified attacks were experimented on reduced variants with up to 790 rounds. The best result on Trivium is a cube attack on a reduced version of 767 initialization rounds instead of 1152.

2.3.2 Bivium

Another stream cipher Bivium has been attacked by Shunbo Li, Yan Wang and Jialong Peng using cube testers attacks in 2010 [5]. Bivium has a key length of 80 bits and Non Linear Feedback Shift Register operates on a 177 bits internal state. 56 linearly independent equations have been found which include 40 single key bits and 16 equations requiring a total of 19 additions mod 2. Proposed attack reduces time complexity from $2^{39.12}$ for Eibach and $2^{27.5}$ for Vielhaber to 2^{26}.

2.3.3 Rabbit Stream Cipher

Rabbit Stream Cipher is one of the finalists of eSTREAM project which uses 128-bit secret key[8]. Analysis is based on chosen Initial Vector analysis on reduced N-S round of Rabbit though using multi cube tester.With 2^{25} complexity, using one iteration of next state function the keystream is completely distinguished from random.

2.3.4 Grain

Grain stream cipher was attacked using an efficient FPGA implementation [3]. The best result (a distinguisher on Grain-128 reduced to 237 rounds, out of 256) was achieved after a computation involving 2^{54} clockings of Grain-128, with a 256 * 32 parallelization.

For instance, running a 30-dimensional cube tester on Grain-128 takes 10 seconds with FPGA machine, against about 45 minutes of bitsliced C implementation.

2.4 Testable Properties

In this section, efficiently testable properties will be discussed, which can be used to build cube testers. Let C be the size of CV, and S be the size of SV, the complexity is given as number of evaluations of tested function f. Each query of the tester to the superpoly requires 2^C queries to the target cryptographic function.

2.4.1 Balance Test

A random function is expected to contain as many zeroes as ones in its truth table. Superpolys that have a strongly unbalanced truth table can thus be distinguished from random polynomials, by testing whether it evaluates as often to one as to zero, either deterministically (by evaluating the superpoly of each possible input), or probabilistically (over some random subset of the SV).

2.4.2 Independence Test

Under the null hypothesis, each pair of superpolys should behave like two independent fair coins. Therefore, over all the input samples, one-fourth the time the pair of outputs should be (0,0), and likewise for (0,1), (1,0) and (1,1) [1].

2.4.3 Off-By-One Test

Under the null hypothesis, over all the input samples, when one of the superpoly input bits is flipped from zero to one or one to zero, half the time the output bit should also also flip and half the time the output bit should not flip.

2.4.4 Low Degree Test

A random superpoly has degree atleast (S - 1) with high probability. Cryptographic functions that rely on a low-degree function, however, are likely to have superpolys of low degree. It closely relates to probabilistically checkable proofs and to error-correcting codes.

2.4.5 Presence of Linear Variable Test

This is a particular case of the low degree test, for degree, d=1 and a single variable. Indeed, the ANF of a random function contains a given variable in at least one monomial of degree at least two with probability close to 1.

2.4.6 Presence of Neutral Variable Test

Dually to the linearity test, one can test whether a SV is neutral in the superpoly, that is, whether it appears in at least one monomial.

Linearity Test and Neutrality Test does not require the superpoly to have a low degree to be tested.

2.5 Attacks on SHA-256

SHA-256 is the 256 bit output version of SHA-2, which is widely deployed in practical systems. The SHA-256 hash function has started getting attention recently by the crypanalysis community due to the various weaknesses found in its predecessors such as MD4, MD5, SHA-0 and SHA-1.

Reduced Round SHA-256 was attacked by Somitra Kumar Sanadhty and Palash Sarkar. Collisions were found after 18 steps[25]. Differential paths for 19, 20, 21, 22 and 23 rounds of steps of SHA-256 were also found.

SHA-256 was attacked by Jian Guo and Krystian Matusiewicz in 2009 and results were represented in paper, titled "Preimages for Step-Reduced SHA-2" [24]. A preimage attack for 42 step-reduced SHA-256 with time complexity $2^{251.7}$ and memory requirements of order 2^{12}. Attack applied was a meet-in-the-middle preimage attack.

A new cryptanalysis technique Biclique is tested against SHA-256 and SHA-512 hash functions for finding preimages by D. Khovratovich, C. Rechberger and A. Savelieva called Biclique Cryptanalysis. Results and observations are discussed in paper titled, "Bicliques for preimages: Attacks on skein-512 and SHA-3 family". Preimages were found against reduced steps of SHA-256. A preimage is found against 43 step SHA-256 with complexity $2^{254.9}$ and against 45 step SHA-256 preimage is found with complexity $2^{255.5}$.

A second order differential attack is carried out on SHA-256 by Mario Lamberger and Florian Mendel in 2011. Second-order differential attack is shown on the SHA-256 compression function reduced to 46 out of 64 steps. It is the best attack applied so far with a practical complexity.

A summary of attacks performed of SHA-256 is given in table 2.1

Table 2-1Cryptanalytic Attacks on SHA-256

Attack Method	**Attack**	**Authors**	**Year**	**Rounds**	**Complexity**
Deterministic	Collision[25]	S. Sanadhya and P. Sarkar	2008	24/64	$2^{28.5}$
Meet-in-the-middle	Preimage[24]	K. Aoki, J. Gao and Y. Sasaki	2009	42/64	$2^{251.7}$
Meet-in-the-middle	Preimage[26]	Jian Guo, San Ling and Christian Rechberger	2010	42/64	$2^{248.4}$
Differential	Pseudo Collision[18]	M. Lamberger and F. Mendel	2011	46/64	2^{178}
Biclique	Preimage[22]	D. Khovratovich, C. Rechberger, and A. Savelieva	2011	45/64	$2^{255.5}$
	Pseudo Preimage[22]		2011	52/64	2^{255}

2.6 Summary

Cube testers attack doesn't require the attack cryptosystem to have a low degree. A detailed review of the stream ciphers, block ciphers and hash functions in which non-randomness has been found using the cube testers attack is provided in the chapter. Testable properties are also discussed which can be tested with the help of Cube testers. The chapter also includes attacks launched against SHA-256 which are found in literature.

Chapter 3

Cube Testers Attack and Property Testing

3.1 Introduction

Generally a cryptosystem can be represented in the form of polynomials in GF(2) containing both the secret and the public variables. For hash functions and stream ciphers, the public variables refer to IV variables and for block ciphers it refers to the plaintext variables. The cube testers attack detects nonrandom behavior without recovering a secret key. Cube testers attack and its predecessors have been introduced in 2008 [4].

The Chapter 3 is divided into 6 sections. Section 3.2 contains the Cube Attack explanation. Section 3.3 discusses the theory of cube testers attack. Section 3.4 describes the teminology of the cube testers attack. Section 3.5 explains the attack methodology of the cube attack. Section 3.6 intoduces the concept of Property Testers and Section 3.7 discusses the Testable Properties one by one.

3.2 Cube Attack

Cube attacks are a powerful and generalize AIDA as a key-recovery attack. This exploits implicit low-degree equations in cryptographic algorithms. So it is tried to obtain linear equations in unknown key bits by combining outputs of the cipher for certain chosen Initial Vectors.

Let $f(k_1, \ldots, k_n, v_1, \ldots, v_m)$ be a multivariate polynomial of degree d with n secret variables $k_1, \ldots, k_n$ and m public variables $v_1, \ldots, v_m$ over GF(2). Given an index set $I \subset \{I, \ldots, m\}$, , the function can be represented algebraically under the form

$$f(k_1, \ldots\ldots, k_n, v_1, \ldots\ldots, v_m) = t_{(I)}p_{s(I)} + q(k_1, \ldots\ldots, k_n, v_1, \ldots\ldots, v_m) \quad (3.1)$$

Where $t_{(I)}$ is called the cube that the monomial contains all the $v_i's$ with $i \epsilon I$, $p_{s(I)}$ is a polynomial that has no variable in common with $t_{(I)}$, and no monomial in the polynomial q contains $t_{(I)}$. Summing f over the cube $t_{(I)}$ for other variables fixed, we get the following theorem.

Theorem 1.

$$\sum_I f(k_1, \ldots\ldots, k_n, v_1, \ldots\ldots, v_m) = \sum_I t_{(I)}p_{s(I)} + q(k_1, \ldots\ldots, k_n, v_1, \ldots\ldots, v_m) \quad (3.2)$$

To demonstrate these notions, let

$$f(k_1, k_2, v_1, v_2, v_3) = k_1k_2 + k_2v_1 + k_1v_1v_2 + k_2v_2v_3 + k_2v_1v_2 + k_1 + v_1v_2 + 1) \quad (3.3)$$

be a polynomial of degree 3 in 5 variables, and let be an index subset of size 2. We can represent f as

$$f(k_1, k_2, v_1, v_2, v_3) = v_1v_2(k_1 + k_2 + 1) + k_2v_1 + k_1v_1v_2 + (k_2v_2v_3 + k_1k_2 + k_1 + 1) \quad (3.4)$$

where

$$t_{1=}v_1v_2 \quad (3.5)$$

$$p_{s(I)} = k_1 + k_2 + 1 \quad (3.6)$$

$$q(k_1, k_2, v_1, v_2, v_3) = k_2v_2v_3 + k_1k_2 + k_1 + 1 \quad (3.7)$$

3.2.1 Definition 3.1

$p_{s(I)}$ is called the superpoly of I in f. A cube is called a maxterm if and only if its superpoly. $p_{s(I)}$ had degree 1 (i.e. is linear but not a constant). The polynomial f is called the master polynomial.

To recover the secret variables, cube attacks have two phases: the preprocessing phase and online phase.

3.2.2 Preprocessing Phase

In the preprocessing phase, the attacker finds sufficiently many maxterms of the master polynomial. For each maxterm, the coefficients of the secret variables are found in the symbolic representation of the linear superpoly. The main challenge of the attacker in the preprocessing phase is to find enough maxterms with linearly independent superpolys. The attacker randomly chooses a subset I of public variables and uses efficient linearity tests to check whether its superpoly is linear.

3.2.3 Online Phase

During the online phase, the secret variables are fixed. The attacker evaluates each linear superpoly by summing over the values of the cryptosystem for every possible assignment to its maxterm. Once enough linear superpolys are found, the key can be recovered by simple linear algebra techniques.

Cube attacks are provably successful when applied to random polynomials of degree d over n secret variables whenever the number m of public variables exceed $d + log_d n$. Their complexity is $n2^{d-1} + n^2$ bit operations, which is polynomial in n and amazingly low when d is small.

3.3 Cube Testers Attack Theory

Let Fn denotes the set of all function mapping $\{0,1\}^n$ to $\{0,1\}$, $n > 0$. For a given n, a random function is a random element of Fn, we have $|Fn| = 2\hat{}(2^n)$. In the ANF of a random function, each monomial (and in particular, the highest degree monomial $(x_1, \dots, x_n)$) appears with probability ½, hence a random function has

maximal degree of with probability ½. Similarly, it has degree $(n-2)$or less with probability $\frac{1}{2^{(n+1)}}$. Note that the explicit description of a random function can be directly expressed as a circuit with, in average, $2^{(n-1)}$ gates (AND and XOR), or as a string of 2^n bits where each bit is the coefficient of a monomial (encoding the truth table also requires 2^n bits, but hides the algebraic structure).

Informally, the distinguisher for a family $F \subsetneq Fn$ is a procedure that, given a function f randomly sampled from $F^* \in \{F, Fn\}$ efficiently determines which one of these two families was chosen as F^*. A family F is a pseudorandom if and only if there exists no efficient distinguisher for it. In practice, for example for hash functions, a family of function is defined by a k-bit parameter of the function. Randomly chosen and unknown to the adversary, and the function is considered broken (or, at least "nonrandom") if there exists a distinguisher making significantly less than 2^k queries to the function. Note that the distinguisher that runs in exponential time in the key may be considered as "efficient" in practice.

The terminology difference between a distinguisher and the more general detection of pseudorandomness, is that the former denotes a distinguisher where the parameter of the family of functions is the cipher's key, and thus cannot be modified by adversary through its queries; the latter considers part of the key as public input and assumes as secret an arbitrary subset of the input (including the input bits that are normally public, like IV bits). The detection of non randomness thus does not necessarily correspond to a realistic scenario.

3.3.1 Cube Variable (CV) and Superpoly Variable (SV) Inputs

Assume some polynomial $F(x, y)$ which can be shown in algebraic numeric form. Let $x_1x_2 \dots x_c$ be a subterm of $F(x, y)$ which is the product of the variables known as cube inputs. Then factorizing p by t_I yields

$$F(x, y) = (x_1x_2 \dots x_c)Q(y) + R(x, y) \qquad (3.8)$$

Where $x_1x_2 \dots x_c$ is the cube input, $Q(y)$ is the superpoly input of $F(x, y)$ and is the linear combination of all terms which do not contain cube inputs $x_1x_2 \dots x_c$.

3.4 Terminology

To distinguish $F \subsetneq Fn$ from Fn, cube testers partition the set of public variables into two complimentary subsets, called cube variables, CV and superpoly variables, SV.

These notions are illustrated with the example where four variables x_1, x_2, y_1 and y_2 are used. x_1 and x_2 are considered as cube inputs and y_1 and y_2 are considered as superpoly inputs.

$$f(x_1, x_2, y_1, y_2) = x_1 + x_1x_2y_1 + x_1x_2y_2 + y_1 \qquad (3.9)$$

x_1x_2 is considered as cube and $(y_1 + y_2)$ is considered as a superpoly, because

$$f(x_1, x_2, y_1, y_2) = x_1 + x_1x_2(y_1 + y_2) + y_1 \qquad (3.10)$$

Superpoly Inputs	Cube Inputs
$y_1 \ and \ y_2$	$x_1 \ and \ x_2$

Figure 3-1 Cube and Superpoly Examples

Here the cube variables(CV) are x_1 and x_2 and the superpoly variables (SV) are y_1 and y_2. Therefore by setting a value to y_1 and y_2, for example $y_1 = 0$ and $y_2 = 1$, one can compute $y_1 + y_2 = 1$ by summing function for all possibilities choices of x_1x_2. Note that it is not required for all superpoly variables to actually appear in the

superpoly of the maxterm. For example if function then the superpoly of x_1x_2 is y_1 but the superpoly variables are both y_1 and y_2.

When the given function is a hash function, not all inputs should be considered as variables and not all Boolean components should be considered as outputs, for the sake of efficiency and keeping the consequent equations simple as possible. For example if f maps 1024 bits to 256 bits, one may choose 8 cube variable input and 8 superpoly variable input and set a fixed value to the other outputs. These fixed inputs determine the coefficient of each monomial in the Algebraic Normal Form with cube variable and superpoly variable as variables. This is similar to the preprocessing phase of key recovery cube attacks where the attacker has access to all input variables. Finally for the sake of efficiency, attacker may only evaluate the superpolys for a subset of 256 Boolean components of the output.

Cube Testers distinguish a family of functions from random functions by testing a property of the superpoly for a specific choice of cube variable and superpoly variable. This idea will be explained with the help of simple examples.

Consider

$$f(x_1, x_2, y_1, y_2) = y_1 + x_1y_1y_2 + x_2y_1y_2 + y_1 \qquad (3.11)$$

And suppose that we choose cube variables x_1 and x_2 and superpoly variables y_1 and y_2 and evaluate the super poly of x_1x_2.

$$f(x_1, x_2, 0,0) + f(x_1, x_2, 0,1) + f(x_1, x_2, 1,0) + f(x_1, x_2, 1,1) = 0 \qquad (3.12)$$

This yields zero for any $(x_1, x_2) \in \{0,1\}^2$, that is the superpoly of x_1x_2 is zero, i.e none of the monomials x_1x_2, $x_1x_2y_1$, $x_1x_2y_2$ or $x_1x_2y_1y_2$ appears in f. In comparison, in a random function the superpoly of x_1x_2 is null with probability only $1/16$, which suggests that f was not chosen at random (indeed, it was chosen particularly sparse, for clarity). Generalizing the idea, one can deterministically test

whether the superpoly of a given maxterm is constant, and return "random function" if and only if the superpoly is not constant.

Let $f \in F_n, n > 4$. A probabilistic test is presented that detects the presence of monomials of the form $x_1x_2x_3x_4 \ldots x_j$ (e.g. $x_1x_2x_3$, $x_1x_2x_3x_n$, etc).

A random value of $(x_4 \ldots . x_n) \in \{0,1\}^{n-4}$ is chosen. Then $f(x_4 \ldots . x_n)$ is summed over all values of (x_1, x_2, x_3), to get

$$\sum_{(x_1,x_2,x_3)\in\{0,1\}^3} f(x_1, \ldots, x_n) = p(x_4, \ldots, x_n) \tag{3.13}$$

Where p is a monomial such that

$$f(x_1x_2 \ldots . x_n) = (x_1x_2x_3).p(x_4 \ldots . x_n) + q(x_1x_2 \ldots . x_n) \tag{3.14}$$

where the polynomial q contains no monomial with $x_1x_2x_3$ as a factor in its algebraic normal form.

The process is repeated N times, recording all the values of $p(x_4 \ldots . x_n)$. If f were a random function, it would contain at least one monomial of the form $x_1x_2x_3 \ldots . x_j$ with high probability, hence for a large enough number of repetitions N, one would record at least one nonzero $p(x_4 \ldots . x_n)$ with high probability. However if no monomial of the form $x_1x_2x_3 \ldots . x_j$ appears in the algebraic normal form, $p(x_4 \ldots . x_n)$ always evaluates to zero.

3.5 Cube Testers Attack Methodology

Consider a cryptographic primitive, such as a hash function in this particular case, to be a Boolean function with multiple inputs bits and output bits.

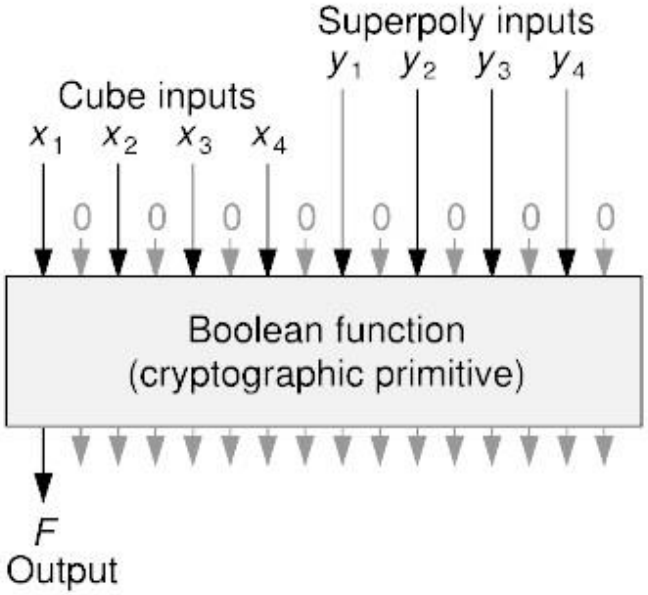

Figure 3-2 Cube Test of one Output Bit of a Cryptographic Primitive

Following the terminology shown in figure 3.1, some number c of the input bits are designated as a vector of cube inputs x = $(x_1, x_2, \ldots, x_c)$, and some number s of the input bits are designated as a vector of superpoly inputs y =$(y_1, y_2, \ldots, y_c)$. All input bits other than the cube inputs and superpoly inputs are set to 0 as they are unused bits so they are zeroed to keep following equations to a low complexity. Then a particular output bit can be treated as a polynomial function of the cube inputs and superpolys inputs: F(x, y).

Function F can be expressed as

$$F(x, y) = (x_1 x_2 \ldots x_c)Q(y) + R(x, y) \qquad (3.15)$$

In GF(2), multiplication is same as Boolean "and", and addition is the same as Boolean "exclusive-or". The first part of the right hand side of equation (3.15) consists of the terms in the polynomial that includes all the cube inputs plus one or more superpoly inputs. The cube inputs are factored out, leaving a polynomial in just the superpoly inputs. The second part of the right hand side of equation (3.15) consists

of the remaining terms in the polynomial F, which is another polynomial R in the cube and superpoly inputs. The polynomial Q is called the superpoly of F with respect to the cube inputs x.

The superpoly $Q(y)$ can be calculated by the summation procedure, without even knowing the polynomial formula for Q, as long as the overall Boolean function F can be evaluated.

For summation procedure, first all unused inputs of F are set to be zero. This includes all public and secret inputs which are not part of superpoly input or cube input. This is required to reduce the overall degree and density of the underlying equations. Then all superpoly inputs are (which are included in F) set to y. Then a loop is run for all possible cube inputs (i.e from all zeros to all ones). And each iteration's resulting equation is adding to Q, which is then shown as

$$Q = Q + F(x, y) \tag{3.16}$$

The value of Q is returned at completion of iterations.

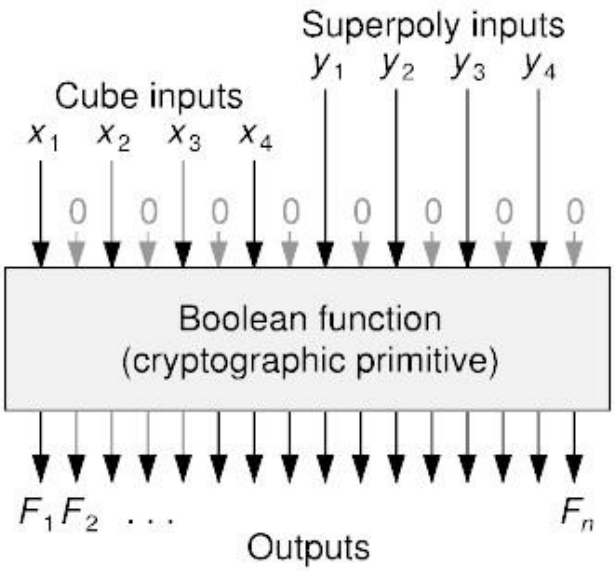

Figure 3-3 Cube Test of all Output Bits of a Cryptographic Primitive

In the summation of F over the 2^c values of x, each term in R (as shown in equation 3.1) is added in an even number of times, since no term in R contains all of x_1 through x_c. Therefore, in GF(2) arithmetic, the terms in R sum up to 0. The terms in Q, however are added in only once, when $x = 11 \ldots .11$. Therefore, the summation yields just Q.

The cube test is based on summation procedure. The null hypothesis is that the cryptographic primitive is a random polynomial. Therefore, for any particular choice of cube and superpoly inputs, the superpoly Q is also a random polynomial. Evaluate Q for some number of randomly chosen values for the superpoly inputs and apply a statistical test to the resulting series of superpoly output values. If the statistical test fails at a designated significance level, then the null hypothesis is disproved, Q is not a random polynomial, and the cryptographic primitive exhibits nonrandom behavior. Testing one or more superpolys might reveal nonrandom behavior where testing the cryptographic primitive as a whole might not reveal nonrandom behavior.

Calculating the superpoly Q requires calculating the whole cryptographic primitive, which yields n output bits, not just one. Each output bit is a different polynomial function of the cube and superpoly inputs, as shown in figure 3.2. Thus, the cube test actually tests multiple superpolys for non randomness.

3.6 Property Testers

Cube testers combine an efficient property tester on the superpoly, which is viewed either as a polynomial or as a mapping with a statistical decision rule. This section gives a general definition of cube testers, starting with basic definitions.

3.6.1 Definition 3.2

A family tester for a family of functions F takes as input a function f of same domain D and tests if f is close to F, with respect to a bound $\in$ on the distance

The tester accepts if $\delta(f, F) = 0$, rejects with high probability if f and F are not $\in$-close and behaves arbitrarily otherwise. Such a test captures the notion of property testing, when a property is defined by belonging to a family of functions P, a property tester is thus a family tester of a property P.

3.6.2 Concept of Property Testers

Cube testers detect non randomness by applying property testers to superpolys, informally, as soon as a superpoly has some "unexpected" property (that is anormally structured) it is identified as nonrandom. Given a testable property $P \subsetneq Fn$, cube testers run a tester for P on the superpoly function f, and use a statistical decision rule to return either random or nonrandom. The decision rule depends on the probabilities $\frac{|P|}{|Fn|}$ and $\frac{|P \cap F|}{|F|}$ and on a margin of error chosen by the attacker. A family F will be differentiated from using the property if

$$\frac{|P|}{|Fn|} - \frac{|P \cap F|}{|F|} |$$

$$\left| \frac{|P|}{|Fn|} - \frac{|P \cap F|}{|F|} \right|$$

is non-negligible. That is, the tester will determine whether f is significantly closer to P than a random function.

3.7 Testable Properties

In this section some examples of testable properties will be discussed, which can be applied to superpoly, that can be used to build cube testers. Let C be the size of

Cube Variables and S be the size of Superpoly Variables, the complexity is given as the number of evaluations of the tested function f. Note that each query of the tester to the superpoly requires 2^c queries to the target cryptographic function. The complexity of any property tester is thus, even in best case, exponential in the number of CV.

3.7.1 Balance Test

A cryptographic polynomial should be a random polynomial, it is expected to contain as many zeros as ones in its truth table. Superpolys that have a strongly unbalanced truth table can thus be distinguished from random polynomials, by testing whether it evaluates as often to one as to zero, either deterministically (by evaluating the superpoly for each possible input), or probabilistically (over some random subset of SV).

A probabilistic version of the test makes $N < 2^S$ iterations, for random distinct values of (x_{C+}). Complexity is respectively 2^n and $N.2^c$.

3.7.2 Constantness

A particular case of balance test considers the "constantness" property, i.e whether the superpoly defines a constant function, that is, it detects either that f has maximal degree strictly less than C (null superpoly), or that f has degree strictly greater than C (non-constant superpoly). This is equivalent to the maximal degree monomial test, used to detect non-randomness of a cryptographic primitive.

3.7.3 Low Degree

A random superpoly has degree atleast $(S-1)$ with high probability. Cryptographic functions that rely on a low-degree function are likely to have

superpolys of low degree. Because it closely relates to probabilistically checkable proofs and to error-correcting codes, extensive research has been done on low-degree testing. The test by Aon et al. [28], for a given degree d queries the function at about $d.4^d$ points and always accepts if the algebraic normal form of the function has degree at most k, otherwise it rejects with some bounded error probability.

3.7.4 Presence of Linear Variables

This is a particular case of low-degree test discussed in section 3.7.3, for degree $d = 1$ and a single variable. Indeed, the algebraic normal form of a random function contains a given variable in at least one monomial of degree atleast two with probability close to 1. One can thus test whether a given superpoly variable appears only linearly in the superpoly. One can test whether a given superpoly variable appears only linearly in the superpoly or not.

Randomly $(x_2, \ldots., x_S)$ is picked. Then both possible values of x_1 are put in the function. If

$$\mathrm{P}(0, x_2, \ldots., x_S) = \mathrm{P}(1, x_2, \ldots., x_S) \qquad (3.17)$$

Nonlinear is returned. And if

$$\mathrm{P}(0, x_2, \ldots., x_S) \neq \mathrm{P}(1, x_2, \ldots., x_S) \qquad (3.18)$$

Linear is returned. These conditions are repeated N times. This test answers correctly with probability about $1 - 2^{-N}$ and computes $N.2^{C+1}$ time the function f.

3.7.5 Independence Test

Under the null hypothesis, each pair of superpoly should behave like two independent fair coins. Therefore, over all the input samples, one-fourth the time the pair of outputs should be (0,0), one-fourth the time the pair of outputs should be (0,1), one-fourth the time the pair of outputs should be (1,0) and one-fourth the time the pair

of outputs should be (1,1). The counts for the chi-square test of superpoly pair (Q_i, Q_j) are $N1$ = observed number of (0,0) pairs in the series of m values for superpolys $Q_i \; and \; Q_j$, N_2 = observed number of (0,1) pairs, N_3 = observed number of (1,0) pairs and N_4 = observed number of (1,1) pairs and $E_1 = E_2 = E_3 = E_4 = m/4$.

3.7.6 Off-By-One Test

Under the null hypothesis, over all the input samples, when one of the superpoly input bits is flipped from 0 to 1 or 1 to 0, half of the time the output bit should be flipped and half of the time the output bit should not be flipped. The counts for the chi-square test of output Q_i and input s_j are N_1 = observed number of times Q_i did not flip when s_j flipped, N_2 = observed number of times Q_i flipped when s_j flipped, and $E_1 = E_2 = m/2$.

One subtlety in the off-by-one test is that the same occurrence must not be counted twice. For example, suppose two of the m superpoly input samples happen to be 101110 and 001110 (s = 6). Flipping the first bit in the first sample will cause the output bits to flip or not flip in the same way as flipping the first bit in the second sample. Thus, the outcomes from flipping the first bit for these two samples are not independent. In each such case, the number of samples will be reduced by 1.

3.8 Summary

In this chapter the description and methodology of the cube testers attack has been discussed. The complete process of the cube testers attack has been followed by taking an example of polynomial for better understanding. After the conceptual view, the actual procedure has been carried out by going through algorithms and theorems. At the end, concepts of property testers and examples of testable properties have been discussed.

Chapter 4

Structure of SHA-256

4.1 Introduction

Hash functions are a building block for numerous cryptographic applications. In this chapter, a description of hash function with particular focus on Secure Hash Algorithm (256 bit version) has been given. Properties of good hash functions have also been discussed. Focus has been laid on structure of SHA-256 throughout this chapter.

Section 4.2 gives a general introduction of hash functions. Section 4.3 contains the general classes of hash function. Section 4.4 discusses the general properties of good hash functions. Section 4.5 explains the structure of SHA-256 in detail.

4.2 Hash Function

A hash function is a cryptographic primitive that compresses an arbitrary length input into a fixed length output called message digest. It does this in such a way that output is effectively unique with regard to the input, and the process cannot be reversed to yield the input from the output. In strict mathematical terms, hash function can be defined as

$$h : \{0 , 1\}^* \rightarrow \{0 , 1\}^m \qquad (4.1)$$

where m is the fixed length of the hash function h in bits.

The output of h must be effectively unique. This means that a computation that produces and x and y such that $x \neq y$ and $H(x) = H(y)$ must take at least $2^{\frac{m}{2}}$ hash operation, which is approximately the number of has operations in which an x and y could be found using random search only.

Hash functions are also called one-way functions because it is easy to determine the hash from the message but mathematically infeasible to determine the message from the hash. This means that given a message digest z such that $z = h(x)$, computing x from z should require work atleast equivalent to hash operations, which is the number of hash operations necessary to find x by exhaustive search.

4.3 Classes of Hash Functions

At the highest level, hash functions can be split into two classes. Keyed and Unkeyed hash functions. Keyed hash functions are also called Message Authentication Codes (MACs), allows message authentication by symmetric techniques. MAC algorithms take two functionally distinct inputs, a message and a secret key, and produce a fixed-size say (n-bit) output, with the design intent that it is infeasible in practice to produce the same output without knowledge of key. MACs can be used to provide data integrity and symmetric data origin authentication, as well as identification in symmetric-key schemes.

An unkeyed hash function's hash value corresponding to a particular message x is computed in time T_1. The integrity of this hash value is protected in some manner. At a subsequent time T_2, the following test is carried out to determine whether the message has been altered, i.e whether a message x' is the same as the original message. The hash value of x' is computed and compared to the protected hash-value, if they are equal, one accepts that the inputs are also equal, and thus that the message has not been altered. The problem of preserving the integrity of a potentially large message is thus reduced to that of a fixed-size hash-value.

4.4 Hash Function Properties

Three properties are discussed in this section, for an unkeyed function h with inputs x and x' and outputs y and y'.

4.4.1 Preimage Resistance

For any pre-specified output, it is computationally infeasible to fine any input which hashes to that output, i.e, to fine any preimage x' such that $h(x') = y$, when given any y for which a corresponding input is not known.

4.4.2 Second Preimage Resistance

For any pre-specified output, it is computationally infeasible to find any second input which has the same output as any specified input, i.e, given x to find a second preimage $x' \neq x$ such that $h(x) = h(x')$.

4.4.3 Collision Resistance

It is computationally infeasible to find any two distinct inputs x , x' which hash to the same output, i.e, such that $h(x) = h(x')$. For collision resistance, there is free choice of both inputs.

4.5 Secure Hash Algorithm – 256 (SHA-256)

SHA-256 is a 256-bit has and is supposed to provide 128 bits of security against collision attacks. In SHA-256, the message to be hashed is first padded with its length in such a way that the result is a multiple of 512 bits long and then parsed into 512-bit message blocks $M^{(1)}, M^{(2)}, M^{(3)}, \ldots \ldots, M^{(N)}$.

The message blocks are then processed one at a time. Beginning with a fixed initial value $H^{(0)}$, sequentially compute

$$H^{(i)} = H^{(i-1)} + C_{M(i)}(H^{(i-1)}) \quad (4.2)$$

where $\boldsymbol{C}$ is the SHA-256 compression function and + means word-wise mod addition. $\boldsymbol{H^{(N)}}$ is the hash of $\boldsymbol{M}$.

4.5.1 Padding and Parsing

Input message is to be padded to make its length a multiple of 512 bits. Suppose the length of message M, in bits is l. Append the bit "1" to the end of the message, and then k zero bits, where k is the smallest non-negative solution to the equation $l + 1 + k \equiv 448 \, mod \, 512$. To this, append the 64-bit block which is equal to the number , length of message written in binary. So that length of actual given input could be determined from the input. For example, the (8-bit ASCII) message "abc" has length of 24 bits (as there are three characters, each of eight bits), so it is padded with a one, then 448 – (24 + 1) = 423 zero bits, and then its length to become the 512-bit padded message

01100001	01100010	01100011	1	00…0 0	0…011000
a	b	c		423 bits	64 bits

Parse the message into N 512-bit blocks $M^{(1)}M^{(2)} \ldots M^{(N)}$. The first 32 bits of message block i are denoted $M_0^{(i)}$, the next 32 bits are $M_1^{(i)}$, and so on up to $M_{15}^{(i)}$. Big-endian convention is used throughout, so within each 32-bit word, the left most bit is stored in the most significant bit position.

4.5.2 SHA-256 Hash Computation

SHA-256 can be used to hash a message, M, having a length of l bits, where $0 \leq l \leq 2^{64}$. The algorithm uses a message schedule of sixty four 32-bit words,

eight working variablesof 32 bits each and a hash value of eight 32-bit words. The final result of SHA-256 is a 256 –bit message digest.

The words of message schedule are labeled $W_0, W_1, \ldots, W_{63}$. The eight working variablesare labeled a, b, c, d, e, f, g and h. The words of the hash value are labeled $H_0^{(i)}, H_1^{(i)}, \ldots, H_7^{(i)}$, which will hold the initial value, $H^{(0)}$, replaced by each successive intermediate hash value (after each message block is processed), $H^{(i)}$, and ending with the final hash value, $H^{(N)}$. SHA-256 also uses two temporary words, T_1 and T_2.

4.5.3 SHA-256 Constants

SHA-256 uses a sequence of sixty-four constant 32-bit words, $K_0^{\{256\}}, K_1^{\{256\}}, \ldots, K_{63}^{\{256\}}$. These words represent the first thirty-two bits of the fractional parts of the cube roots of the first sixty-four prime numbers. In hex, these constant words are as follows.

Table 4-1Constants used in SHA-256 calculation

428a2f98	71374491	b5c0fbcf	e9b5dba5	3956c25b	59f111f1
923f82a4	ab1c5ed5	d807aa98	12835b01	243185be	550c7dc3
72be5d74	80deb1fe	9bdc06a7	c19bf174	e49b69c1	efbe4786
0fc19dc6	240ca1cc	2de92c6f	4a7484aa	5cb0a9dc	76f988da
983e5152	a831c66d	b00327c8	bf597fc7	c6e00bf3	c5a79147
06ca6351	14292967	27b70a85	2e1b2138	4d2c6dfc	53380d13
650a7354	766a0abb	81c2c92e	92722c85	a2bfe8a1	a81a664b
c24b8b70	c76c51a3	b192e819	d6990624	f40e3585	106aa070
19a4c116	1e376c08	2748774c	34b0bcb5	391c0cb3	4ed8aa4a
5b9cca4f	682e6ff3	748fa2ee	78a5636f	84c87814	8cc70208

90befffa	a4506ceb	bef9a3f7	6c7178f2		

4.5.4 SHA-256 Functions

SHA-256 uses six logical functions, where each function operates on 32-bit words, which are represented as x, y and z. The result of each function is a new 32-bit word.

$$Ch(x,y,z) = (x \cap y)(x' \cap z) \tag{4.3}$$

$$Maj(x,y,z) = (x \cap y) \oplus (y \cap z) \oplus (z \cap x) \tag{4.4}$$

$$\sum_0^{\{256\}}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x) \tag{4.5}$$

$$\sum_1^{\{256\}}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x) \tag{4.6}$$

$$\sigma_0^{\{256\}}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x) \tag{4.7}$$

$$\sigma_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x) \tag{4.8}$$

4.5.5 SHA-256 Computation Main Loop Steps

The SHA-256 hash computation uses functions and constants discussed in section 4.5.3 and 4.5.4. Addition is performed modulo 2^{32}. After preprocessing is completed, each message block, $M^{(1)}, M^{(2)}, \dots, M^{(N)}$ is processed in order using the following steps.

First of all message schedule W_t is calculated using the following equations.

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases} \tag{4.9}$$

$\sigma_1^{\{256\}}$ and $\sigma_0^{\{256\}}$ are defined above in equations 4.7 and 4.8.

After preparation of message schedule, eight working variables a, b, c, d, e, f, g and h are initialized with the $(i-1)^{st}$ hash value:

$$a = H_0^{(i-1)}$$

$$b = H_1^{(i-1)}$$

$$c = H_2^{(i-1)}$$

$$d = H_3^{(i-1)}$$

$$e = H_4^{(i-1)}$$

$$f = H_5^{(i-1)}$$

$$g = H_6^{(i-1)}$$

$$h = H_7^{(i-1)}$$

Then a loop is run from 0 to 63 showing 64 steps of main loop used to calculate the hash value. Following functions are calculated in the loop.

$$T_1 = h + \sum_1^{\{256\}}(e) + Ch(e, f, g) + K_t^{\{256\}} + W_t$$

$$T_2 = \sum_0^{\{256\}}(a) + Maj(a, b, c)$$

$$h = g$$

$$g = f$$

$$f = e$$

$$e = d + T_1$$

$$d = c$$

$$c = b$$

$$b = a$$

$$a = T_1 + T_2$$

Then the i^{th} intermediate hash value $H^{(i)}$ is found using the following equations.

$$H_0^{(i)} = a + H_0^{(i-1)}$$

$$H_1^{(i)} = b + H_1^{(i-1)}$$

$$H_2^{(i)} = c + H_2^{(i-1)}$$

$$H_3^{(i)} = d + H_3^{(i-1)}$$

$$H_4^{(i)} = e + H_4^{(i-1)}$$

$$H_5^{(i)} = f + H_5^{(i-1)}$$

$$H_6^{(i)} = g + H_6^{(i-1)}$$

$$H_7^{(i)} = h + H_7^{(i-1)}$$

All these values are concatenated to get a result of 256 bits.

$$H_0^{(N)} \parallel H_1^{(N)} \parallel H_2^{(N)} \parallel H_3^{(N)} \parallel H_4^{(N)} \parallel H_5^{(N)} \parallel H_6^{(N)} \parallel H_7^{(N)}$$

After these all steps hash for one input of 512 bits is found. Then these steps are repeated for total number of blocks of input. And output for each block is XORed with the last one. At the XOR calculation after the last block, last output is the final hash value found for that particular input.

4.6 Summary

This chapter covers the structural details of the hash function SHA-256 on which tests are to be carried out. Before testing the statistical properties, it was important to analyze the structure of SHA-256 in detail. The hashing algorithm of SHA-256 contains preprocessing steps, which includes padding and parsing of input message into 512 bit length blocks. Then main loop containing 64 iterations is run on

each block. And output hash of each block is XORed in last output. Thus in the end, a hash of 256 bits is generated.

Chapter 5

Cube Testers Attack on SHA-256

5.1 Introduction

In this chapter, details of cube testers attack and Property Testing on SHA-256 are included. SHA-256 is the target hash function. In this research, the results obtained on SHA-256 for Balanceness test, Off By One test and Impedance test have been discussed.

In Section 5.2, Cube Testers Attack on SHA-256 has been specified. Section 5.3 contains the reason for selecting particular superpoly inputs and cube inputs, Section 5.4 includes the Setting up of cube input for Statistical analysis and description of Chi Square test, Section 5.5 includes details of Cube Testers Attack Implementation as implemented in C++, Section 5.6 contains Balance Test methodology and results, Section 5.7 contains Impedance test methodology and results and in Section 5.8 Off by One test methodology and results are discussed.

5.2 SHA-256 Customizations for Cryptographic Tests

Some customizations are done in SHA-256 for the sake of simplicity so it can be tested for statistical properties easily. First of all, length of input taken for hash functions was limited 512 bits (i.e one block of input). So that XORing of output of more than one block doesn't complicate the results.

Secondly pre-processing phases are removed, which include padding and parsing. As padding amends non-zero bits (length of message in bits) at the end of the block, and for all the statistical tests, input bits which are not part of cube or superpoly input are supposed to be kept zero[1]. Parsing is used to break up a large input

message into 512 bit blocks. It is not required as input taken is of length of 512 bits already (one block input).

5.3 Selection of Cube and Superpoly Input Bit

Cube testers are kind of black box tests. As per the authors of Cube Testers the selection of "good" bits depends on the structure of the cipher, and significantly affects the results. Finding the optimal bits is a difficult problem. The choice of superpoly variables does depend often on an insight in the structure of the first few rounds of an iterative construction. Otherwise this is still a lot a matter of trial and error.

The structure of Hash functions in our case is often complex and selection of weak input bits is not easy. So for the sake of completion, whole input of SHA-256, i.e 512 bits were divided into 64 blocks, with each block of 8 bits. And all blocks were taken as Superpoly input one by one. Random blocks of Cube inputs were chosen while testing non-random behavior of Superpoly inputs. Different superpoly inputs gave nonrandom behavior while testing non random properties with different superpoly inputs.

In this way weak inputs of SHA-256 are also found out for different tests.

5.4 Input for Statistical Attack

Each run of Cube Tester on SHA-256 algorithm's core functions samples a group of superpoly and cube input of the target hash function, SHA-256. The input is defined by choosing c cube inputs and s superpoly inputs at random, m samples of the superpoly input values are chosen at random and the superpolys are calculated, yielding a series of M samples for each superpoly bit range, where $M = product\ (2^s, 2^c)$. These outputs are subjected to three statistical tests – balance test,

independence test and off by one test to attempt to disprove the null hypothesis that the target hash function is a random polynomial.

5.4.1 Chi-Square Test

Each of the three statistical tests is a chi-square test. The chi-square test categorizes the series of random values being tested into discrete bins and counts the occurrences in each bin. The $\mathcal{X}^2$ statistic is

$$\mathcal{X}^2 = \sum_{i=1}^{b} \frac{(N_i - E_i)^2}{E_i} \qquad (5.1)$$

Where b is the number of bins, N_i is the observed count in the i-th bin, and E_i is the expected count in the i-th bin if the null hypothesis is true. Typically the E_i values are derived from the total of the counts in all the bins.

The $\mathcal{X}^2$ statistic obeys a chi-square distribution with df degrees of freedom. When the E_i values are determined as df is $b - 1$. The significance is the probability that a statistic greater than or equal to $\mathcal{X}^2$ would be observed by chance when the null hypothesis is true. The significance is 1 minus the cumulative distribution function of the chi-square distribution:

$$Significance\ (d, \mathcal{X}^2) = 1 - P(\frac{d}{2} - \frac{\mathcal{X}^2}{2}) \qquad (5.2)$$

As the observed counts N_i deviate farther from the expected counts E_i, $\mathcal{X}^2$ increases and the significance decreases. If the significance falls below a certain threshold p, the statistical test fails (the null hypothesis is disproved at a significance of p), otherwise statistical test passes. Statistical tests of cryptographic pseudorandom number generators typically use p in the range 0.01 to 0.001. For tests applied in our research work, chi square criterion of 0.01 is being used.

5.4.1.1 Why choose a significance level of 0.01?

When p is larger, say 0.02 or 0.05, the chi-square tests are more stringent, smaller differences between the observed and expected values causes the significance to fall below the threshold and the test to fail. But this means a "false failure", where the test fails even though the function really is random, is more likely. On the other hand, when p is smaller, say 0.01, the chi-square tests are more lenient, larger differences are required between the observed and expected counts to cause the significance to fall below the threshold and the test to fail. A level of significance less than 0.01 can cause "false pass" where the test passes even though the function really is non-random. So value of p is taken 0.01 for these tests which is recommended for cryptanalytic tests.

	P										
DF	0.995	0.975	0.20	0.10	0.05	0.025	0.02	0.01	0.005	0.002	0.001
1	0.0000393	0.000982	1.642	2.706	3.841	5.024	5.412	6.635	7.879	9.550	10.828
2	0.0100	0.0506	3.219	4.605	5.991	7.378	7.824	9.210	10.597	12.429	13.816
3	0.0717	0.216	4.642	6.251	7.815	9.348	9.837	11.345	12.838	14.796	16.266
4	0.207	0.484	5.989	7.779	9.488	11.143	11.668	13.277	14.860	16.924	18.467
5	0.412	0.831	7.289	9.236	11.070	12.833	13.388	15.086	16.750	18.907	20.515
6	0.676	1.237	8.558	10.645	12.592	14.449	15.033	16.812	18.548	20.791	22.458
7	0.989	1.690	9.803	12.017	14.067	16.013	16.622	18.475	20.278	22.601	24.322
8	1.344	2.180	11.030	13.362	15.507	17.535	18.168	20.090	21.955	24.352	26.124
9	1.735	2.700	12.242	14.684	16.919	19.023	19.679	21.666	23.589	26.056	27.877
10	2.156	3.247	13.442	15.987	18.307	20.483	21.161	23.209	25.188	27.722	29.588
11	2.603	3.816	14.631	17.275	19.675	21.920	22.618	24.725	26.757	29.354	31.264

Figure 5-1 Chi Square Distribution Table

5.5 Cube Testers Attack Implementation

This portion includes the implementation details of the cube testers attack by describing the functionalities of each function in the C++ code.

5.5.1 The Main Function

Two arrays of binary zeros named msg_arr and $temp\_msg\_arr$ are declared in this main function. Then initial values of SHA-256 are declared in a variable

named H_Prev. Then input is changed as per the value of constants $const1$ and $const2$ declared at the start of the program. These constants declared at the start, define the position of superpoly input and cube input that is to be used in the program. Then the input bits of msg_arr are changed which are specified by $const1$ and $const2$ in the main function and resulting output is written in the file $input.txt$. This is done so that input which is fed to SHA-256 function can be verified for correctness later on if required.

Two loops in nested form are run to increase values of superpoly and cube input by 1 each and forward the updated input to SHA-256 program for hash computation. Each of these loops is run 256 times as number of cube input bits and superpoly inputs bits are 8 each. Inside the inner loop SHA-256 function is called which calculates the hash value and returns it to the main function.

The output from SHA-256 is also received back into the main function which is then written in the output file using file writing feature of C++.

5.5.2 SHA-256 Function

This function is used to calculate the hash of 512 bits input. The difference between the function made for the tests and standard SHA-256 hash function is that the pre-processing part is omitted. This SHA-256 function contains the following functions.

5.5.2.1 Split Function

This function divides the 512 bits input to 16 blocks of 32 bits each. Each block is input to the function in one step of hash function main round. It takes first 16 steps out of total 64 rounds of hash function to take in the whole 512 bit input. Then the main round's parts are implemented on the input blocks.

5.5.3 Chi Square Function

Values from the main function are fed to Chi Square function and it returns the Chi Square Value as per the specifications set for the particular test, that is being carried out. Estimated values for the output are already calculated. Actual values which are returned after calculated after SHA-256 function are compared with estimated values and extend of difference between expected and actual values defines the Chi Square value.

5.5.4 Environment Used for Cube Testers Attack

The Cube Testers Attack is implemented using Microsoft Visual Studio 2010 Version 10.0.40219.1 SP1 Release on a HP Pavilion M6 Notebook PC with Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz processor. It has 8 GB RAM and Windows 7 as baseline operating system.

In MS Visual Studio, Release configuration instead of default Debug configuration, so that the execution speed of the code increases.

5.6 Balance Test

Under the null hypothesis the hash function is a random function, each superpoly should behave like a fair coin. Over all the input samples, half the time the output should be zero and half the time the output should be one. The counts for the chi-square test of superpoly S_i are N_i which is equal to observed number of zeros in the series of m values for superpoly Q_i, N_2 is equal to observed number of ones. E_1 is the estimated number of zeros and E_2 is the estimated number of ones.

$$E_1 = E_2 = {}^{m}/_{2}$$

5.6.1 Balance Test Methodology

Preprocessing rounds of SHA-256 have been removed as they add non-zero bits in the last 64 bits of the input block. In cube tester attack, all unused bits of input, which are not part of cube input or superpoly input should be zero. For balance test, size of cube input '*c*' and superpoly input '*s*' is set to 8 bits each. So out of 512 input bits, only 16 bits of input are manipulated for a single test run. Rest 496 bits are kept zeros.

SHA-256 is run 2^8 times for a single superpoly input. As with a single superpoly input, 2^8 iterations of cube input are used. Superpoly input size is also eight bits, and each possible iteration of eight bits of superpoly input is used. So that makes 2^8 iterations of superpoly input.

As superpoly is 8 bits long, so total 64 superpolys are made from 512 bits input, each superpoly is of eight bit size. And for each single superpoly, 7 cube inputs are tried from random locations. So the total iterations of SHA-256 for a particular round are $2^8 * 7 * 2^8 * 64$ which makes 29360128 ($\cong 2^{25}$) iterations for SHA-256 for balance ness test of a particular round of SHA-256.

The rounds tested for unbalanceness are rounds 17 and up because in it takes initial 16 rounds of SHA-256 to use whole input in the SHA-256 digest structure. Each round uses 32 bits of input and mixes it with output of last round to find output of next round. So any non-randomness found for less than 17 rounds is meaningless as whole input is not used up till round 17. So total complexity for all rounds tested, i.e round 17 to round 30 is $2^8 * 7 * 2^8 * 64 * 14$, which makes 411041792+ ($\cong 2^{28.6}$) iterations of reduced round SHA-256.

Results of cube testers attack are fed to chi square test to test the variation is just result of a chance or there is significant difference in actual and expected results.

Degree of freedom used for this test is 1 as there are only two possible results and degree of freedom is one less than the number of possible results. Value of p is kept to 0.01 as recommended for cryptographic algorithms. The resulting value is 6.635 as per the chi square table shown in Figure 5.1.

5.6.2 Balance Attack Results

Non-randomness is found from round 17 up till round 25 of SHA-256. Result are given for all reduced round SHA-256's. Balance test has been applied uptill 30 rounds.

All results are given in table 5.1 and table 5.2.Cube inputs from seven different locations are tried with each of single superpoly, the chi square value mentioned in observation table is the highest value among the chi square values with different cube inputs. All 512 bits of input are divided into 64 blocks of 8 bits each. So for one round of SHA-256, 64 chi square values are given in table 5.1 and table 5.2, each value is highest among 7 values found for that particular bit range with specific number of rounds of SHA-256. Values exceeding Chi Square threshold, i.e 6.635 are formatted bold and underlined.

Table 5-1 Chi SquareValues of Balance Test with 17 Round SHA-256 to 23 Round SHA-256 for all Input Bit Ranges

Rounds / **Bit Ranges**	17	18	19	20	21	22	23
1 – 8	4.585	2.345	4.346	5.422	3.656	2.818	0.641
9 – 16	5.432	1.023	3.628	5.815	5.82	2.51	0.962
17 – 24	6.471	0.925	0.345	3.726	0.315	1.49	1.535
25 – 32	1.489	5.032	1.249	5.15	6.492	0.676	3.304
33 – 40	5.173	3.141	0.387	0.886	5.361	1.592	5.497
41 – 48	3.742	**7.308**	4.133	3.551	0.639	0.598	3.792
49 – 56	2.905	0.256	5.029	0.984	1.139	3.533	6.436
57 – 64	4.845	3.044	0.357	3.496	1.924	1.977	5.225
65 – 72	6.363	0.59	4.838	2.969	1.598	5.832	0.505

73 – 80	5.144	1.758	4.724	3.949	5.589	2.159	1.023
81 – 88	4.002	4.969	1.152	**7.557**	6.029	2.388	3.266
89 – 96	2.971	1.026	3.125	**7.215**	3.867	5.177	5.591
97 – 104	1.25	6.042	3.627	0.709	4.445	4.846	1.656
105 – 112	0.653	3.422	0.035	4.977	3.484	1.167	3.849
113 – 120	5.582	1.292	1.303	1.157	6.15	3.664	4.830
121 - 128	1.067	4.524	5.084	3.117	2.24	1.58	4.822
129 – 136	5.984	2.927	4.547	3.238	1.147	3.5	0.944
137 - 144	5.804	3.799	6.3	3.620	2.323	4.28	3.82
145 – 152	6.327	0.568	3.688	5.744	0.046	4.839	4.089
153 – 160	0.475	0.458	4.125	2.297	2.046	1.673	5.991
161 – 168	1.622	3.379	0.634	1.424	4.036	3.914	4.224
169 – 176	1.916	5.545	3.026	2.756	3.36	5.377	3.747
177 – 184	6.053	4.420	2.189	0.384	2.481	**8.594**	1.139
185 – 192	4.671	3.946	3.647	1.298	4.173	2.619	3.938
193 – 200	4.722	2.84	3.795	2.015	1.675	6.086	0.296
201 – 208	4.627	0.704	1.928	5.522	5.458	0.599	5.182
209 – 216	5.674	0.854	4.609	5.780	2.956	5.247	0.323
217 – 224	0.685	5.804	4.141	3.174	0.631	1.700	1.514
225 – 232	3.628	5.116	2.624	6.221	3.223	1.017	1.616
233 – 240	2.338	0.357	5.286	6.424	3.745	**7.052**	5.478
241 – 248	0.928	2.88	5.681	2.439	4.967	1.587	1.357
249 – 256	4.728	0.495	0.465	5.82	0.744	0.543	6.177
257 – 264	1.164	1.081	0.625	2.754	0.503	4.296	5.545
265 – 272	4.457	1.82	2.589	4.387	2.410	3.233	2.107
273 - 280	4.208	1.474	1.171	3.922	4.929	1.419	1.210
281 – 288	4.034	4.554	5.346	4.545	5.540	0.380	5.411
289 – 296	4.673	1.987	1.785	3.165	0.933	6.135	**6.929**
297 – 304	0.588	0.039	1.636	1.583	0.129	2.109	0.587
305 – 312	2.05	0.470	2.661	0.511	1.490	5.285	2.632
313 – 320	0.676	4.733	5.855	2.377	2.070	4.785	0.717
321 – 328	2.612	4.070	2.198	1.986	4.571	2.976	2.777
329 – 336	5.556	3.561	2.464	0.298	4.411	2.252	3.483
337 – 344	6.138	5.194	2.824	4.93	3.91	2.852	2.961
345 – 352	**6.813**	5.902	3.382	3.122	0.468	4.09	**7.237**
353 – 360	2.662	4.928	4.208	1.632	0.542	6.052	2.042
361 – 368	4.398	5.731	1.245	0.550	5.594	0.529	0.179
369 – 376	1.377	2.434	0.484	5.092	2.314	6.089	6.013
377 – 384	3.663	0.953	4.876	4.424	0.680	3.438	4.851
385 – 392	1.542	0.352	1.255	4.772	3.287	3.54	2.287
393 – 400	4.724	1.511	4.055	4.293	5.241	5.324	6.382
401 – 408	0.879	5.067	1.382	1.007	3.581	2.822	1.896
409 - 416	3.571	0.246	5.36	4.859	2.610	3.67	4.337
417 – 424	0.493	4.073	1.888	3.6	0.219	5.722	2.742
425 – 432	4.084	0.543	1.144	2.902	5.035	2.655	4.058
433 – 440	4.932	3.912	3.925	0.921	3.724	0.693	4.996
441 – 448	6.208	1.334	4.991	3.329	5.304	3.504	0.858
449 – 456	1.63	6.13	2.148	5.238	3.705	0.246	0.016

457 – 464	**6.72**	3.36	5.101	5.788	1.066	3.404	6.243
465 – 472	**6.941**	4.16	3.752	4.524	2.573	3.296	3.384
473 – 480	**6.786**	4.007	3.180	4.755	2.695	3.703	3.974
481 – 488	**7.064**	**7.84**	0.773	5.704	2.919	5.44	1.979
489 – 496	**8.116**	**7.1079**	6.058	2.302	5.665	4.479	6.451
497 – 504	**7.193**	**6.919**	3.19	**9.162**	4.181	3.989	0.881
505 – 512	**7.377**	**7.458**	0.034	**7.196**	6.172	3.698	0.379

Table 5-2 Chi SquareValues of Balance Test with 24 Round SHA-256 to 30 Round SHA-256 for all Input Bit Ranges

Rounds / **Bit Ranges**	24	25	26	27	28	29	30
1 – 8	5.309	5.326	4.507	5.435	5.152	6.186	1.540
9 – 16	4.615	4.795	0.131	3.918	1.022	4.672	5.503
17 – 24	5.051	5.312	0.201	4.641	3.992	4.683	1.896
25 – 32	0.529	5.866	0.366	0.119	0.793	4.625	2.696
33 – 40	5.943	4.497	5.893	0.514	5.661	3.457	2.135
41 – 48	5.775	4.143	5.061	5.275	3.792	5.478	1.724
49 – 56	5.956	0.082	5.117	5.614	4.794	4.158	2.217
57 – 64	2.44	1.885	4.626	3.106	2.548	2.776	1.280
65 – 72	4.475	4.112	0.748	1.729	3.917	6.043	4.250
73 – 80	1.463	2.779	5.12	4.423	2.43	0.827	5.752
81 – 88	0.316	0.215	2.481	3.808	3.743	0.112	5.592
89 – 96	1.266	2.405	3.164	3.033	4.877	3.909	0.038
97 – 104	4.248	0.165	5.484	2.64	2.615	3.113	4.043
105 – 112	5.481	1.182	1.66	1.46	5.648	1.825	2.036
113 – 120	1.975	3.136	3.732	1.371	5.554	0.653	0.156
121 - 128	0.966	2.318	2.936	2.386	1.825	5.713	5.377
129 – 136	3.164	4.219	2.152	4.753	4.789	3.839	1.592
137 - 144	4.54	5.179	4.009	0.711	0.073	3.493	0.695
145 – 152	2.308	3.067	0.698	1.023	1.423	1.851	5.748
153 – 160	0.645	0.085	4.787	1.169	0.831	1.809	5.434
161 – 168	2.972	2.766	3.993	6.306	2.806	0.594	3.292
169 – 176	5.95	2.414	6.383	2.966	0.317	1.733	1.878
177 – 184	3.459	3.947	3.164	4.158	1.624	2.671	6.162
185 – 192	2.895	2.03	5.301	0.172	0.486	6.036	1.867
193 – 200	4.559	3.655	5.184	6.282	0.384	0.927	5.287
201 – 208	3.817	2.651	6.204	3.973	6.487	5.48	4.047
209 – 216	3.562	3.324	4.245	3.164	1.297	1.832	3.09
217 – 224	3.715	3.238	0.751	2.057	6.249	0.921	1.604
225 – 232	6.497	3.192	2.309	1.949	5.274	0.941	6.046
233 – 240	0.458	4.01	2.263	5.904	4.208	3.615	2.792

241 – 248	4.81	3.468	6.196	6.363	0.678	0.373	1.009
249 – 256	4.841	3.7698	4.722	5.615	6.305	5.558	1.227
257 – 264	1.376	3.34	0.801	0.422	4.986	3.477	2.008
265 – 272	1.975	3.943	5.445	6.22	2.502	5.477	1.476
273 - 280	1.543	5.242	2.062	0.704	2.802	4.425	2.042
281 – 288	1.556	5.116	3.234	4.744	2.027	6.342	6.475
289 – 296	0.171	3.633	0.887	3.018	6.217	5.741	1.406
297 – 304	1.201	1.552	0.458	3.168	4.071	4.051	6.317
305 – 312	6.228	1.976	0.297	1.099	5.931	6.476	2.04
313 – 320	6.322	0.134	1.943	0.503	2.25	4.242	2.537
321 – 328	1.179	6.211	6.296	2.424	0.852	1.265	3.681
329 – 336	3.465	6.054	1.776	1.842	0.932	0.15	1.208
337 – 344	5.471	0.376	0.749	0.547	4.735	4.623	1.922
345 – 352	2.791	5.384	4.75	5.302	4.293	0.107	2.115
353 – 360	5.274	4.1	4.743	3.441	4.198	5.335	5.617
361 – 368	4.384	4.485	5.746	1.519	6.382	1.68	4.877
369 – 376	1.457	1.225	2.648	2.438	4.481	5.595	6.016
377 – 384	1.052	6.093	0.397	2.057	0.9527	2.663	2.443
385 – 392	0.506	4.150	2.458	3.312	2.322	5.793	1.094
393 – 400	1.126	0.968	1.565	0.225	3.33	6.311	4.193
401 – 408	4.964	5.894	4.992	1.168	3.753	6.047	1.939
409 - 416	4.158	5.683	4.374	6.405	3.904	4.185	4.224
417 – 424	**7.526**	2.748	3.275	0.914	6.156	4.334	3.087
425 – 432	4.867	**7.834**	5.577	3.733	6.289	5.138	2.721
433 – 440	5.14	0.923	1.966	1.464	1.967	1.904	4.267
441 – 448	0.304	4.578	5.217	6.28	5.727	2.036	1.684
449 – 456	5.556	4.717	1.543	3.318	3.737	4.708	2.823
457 – 464	6.215	5.944	2.096	3.436	1.048	2.645	4.33
465 – 472	5.528	6.203	5.294	1.065	0.190	0.761	0.511
473 – 480	4.853	2.741	5.531	4.960	1.129	4.528	5.696
481 – 488	1.92	3.523	2.885	1.491	1.741	2.449	5.212
489 – 496	3.332	1.726	2.199	1.319	6.125	6.075	1.943
497 – 504	4.228	**6.914**	4.593	1.131	6.042	2.607	5.172
505 – 512	5.967	**7.170**	5.799	0.376	1.664	1.802	2.679

5.6.2.1 Results for SHA-256 with 17 Rounds

17 round SHA-256 showed non-randomness for a number of superpoly inputs. Output is non-random for superpoly input bits from 345 to 352 and 457 to 512. 17 round Hash function was mainly nonrandom when superpoly belongs to last part of input. Reason is that last part of input is just used in last 3 to 4 rounds. So it doesn't get enough iteration to remove its randomness.

5.6.2.2 Results for SHA-256 with 18 Rounds

In 18 round SHA-256, non-randomness is found when superpoly is set to last bit ranges, i.e 4 blocks of 8 bits each i.e bit number 481 to 512 bits. It is also nonrandom for bit range 41 to 48.

5.6.2.3 Results for SHA-256 with 20 Rounds

For 20 round SHA-256, nonrandomness is reduced to quiet an extent. And it is found non-random for superpoly input is from bit number 81 to bit number 88, bit number 89 to bit number 96 and last 2 pairs of bits, i.e bit number 497 to bit number 512.

5.6.2.4 Results for SHA-256 with 22 Rounds

For 22 round SHA-256, non-randomness is found only for super poly input bit ranges from 177 to 184 and from bit number 233 to bit number 240. Rest all Chi-Square values are under the threshold Chi-Square value given in Chi Square table.

5.6.2.5 Results for SHA-256 with 23 Rounds

For 23 round SHA-256, non-randomness is found for bit ranges 289 to 296 and from 345 to 352.

5.6.2.6 Results for SHA-256 with 24 Rounds

For 24 round SHA-256, non-randomness is found for bit range 417 to 424.

5.6.2.7 Results for SHA-256 with 25 Rounds

For 25 round SHA-256, non-randomness is found for bit ranges 425 to 432 and from 497 to 504.

Table 5-3 Balance Test Results on SHA-256

Number of Rounds	Weak Superpoly Input Bits
17	345-352, 457-512
18	41-48, 481-512
20	81-88, 89-96, 497-512
22	177-184, 233-240
23	289-296, 345-352
24	417-424
25	425-432, 497-504
26 and more rounds	No weak input found with 8-bit superpoly input

5.6.2.8 Results for SHA-256 with 26 and more Rounds

Chi Square Values for 26 and above steps are comfortably satisfy the null hypothesis of non-randomness.

5.7 Impedance Test

Under the null hypotheses, each pair of superpoly should behave like two independent fair coins. Therefore over all the input samples, one-fourth the time the pair of outputs should be (0,1) and similarly for (0,1), (1,0) and (1,1). The counts for the chi-square of super poly (S_i, S_j) are N_1 = ovserved number of (0,0) pairs in the series of m values for superpolys S_i and S_j, N_2 = observed number of (0,1) pairs, N_3 = observed number of (1,0) pairs and N_4 = observed number of (1,1) pairs and $E_1 = E_2 = E_3 = E_4 = {}^{m}/_{4}$.

5.7.1 Impedance Test Methodology

Preprocessing rounds of SHA-256 have been removed as they add non-zero bits in the last 64 bits of the input block. In cube tester attack, all unused bits of input, which are not part of cube input or superpoly input should be zero.

For impedance test, size of cube input 'c' and superpoly input 's' is set to 8 bits each. So out of 512 input bits, only 16 bits of input are manipulated for a single test run. Rest 496 bits are kept zeros.

SHA-256 is run $2^8 * 7$ times for a single superpoly input. As with a single superpoly input, 2^8 iterations of one cube input are used and 7 different cube inputs are used with single superpoly. In the table below only the highest value of chi square is shown which is obtained using any one of cube inputs.

As two superpolys are tested for a single impedance test, SHA-256 algorithm is run $2^8 * 7 * 2$ times for each test of a superpoly. And for one superpoly input pair, 65536 output bit pairs are checked. So for test total SHA-256 algorithm is run $229376 (= 2^8 * 7 * 2 * 2^6)$ times. And total of 4194304 bit pairs are tested using chi square test for non-randomness.

Degree of freedom used for this test is 3 as there are four possible output results and $df = N - 1$ and level of significance i.e value of p is kept to 0.01 which is strictest for cryptographic algorithms. The threshold resulting value is 11.345 as per the chi square table shown in Figure 5.1.

5.7.2 Impedance Test Results

Impedance test deducted non-randomness most of the inputs on which it is applied. Number of rounds is not reduced either. Superpoly input and Cube Inputs are of 8 bits each during this test similar to Balance Test.

Randomness is only found when superpoly input bit range is from bit number 225 to bit number 232, from bit number 241 to bit number 248 and from bit number 265 to bit number 272. Rest all bit showed non-randomness failing to pass impedance test.

Bit ranges giving random results are given below in table 5-5.

Table 5-4 Superpoly Bit Ranges used for Impedance Test and respective Chi Square Values

Bit Range of Superpolys	Maximum Chi Square Value
1 – 8	21.1048
9 – 16	14.103
17 – 24	16.0018
25 – 32	14.4567
33 – 40	21.1448
41 – 48	16.4007
49 – 56	15.2349
57 – 64	18.2416
65 – 72	15.7476
73 – 80	13.4911
81 – 88	14.9569
89 – 96	15.0206
97 – 104	20.7972
105 – 112	12.5139
113 – 120	14.3792
121 - 128	20.2707
129 – 136	15.5327
137 - 144	13.7425
145 – 152	13.7013
153 – 160	15.3298
161 – 168	15.1259
169 – 176	19.111
177 – 184	19.7831
185 – 192	16.6366
193 – 200	14.2089
201 – 208	17.8696
209 – 216	20.4261
217 – 224	19.1312
225 – 232	**10.9291**
233 – 240	19.6805
241 – 248	**9.3333**

249 – 256	16.6262
257 – 264	12.0787
265 – 272	**10.4713**
273 - 280	14.8073
281 – 288	16.965
289 – 296	18.8754
297 – 304	11.3493
305 – 312	14.2485
313 – 320	14.7768
321 – 328	19.9442
329 – 336	15.9002
337 – 344	13.5158
345 – 352	19.6415
353 – 360	13.0736
361 – 368	16.1736
369 – 376	15.2147
377 – 384	12.4775
385 – 392	14.3013
393 – 400	12.675
401 – 408	16.9726
409 - 416	17.9666
417 – 424	11.894
425 – 432	18.6752
433 – 440	19.766
441 – 448	15.8215
449 – 456	13.7868
457 – 464	14.1442
465 – 472	17.3581
473 – 480	18.7198
481 – 488	15.0384
489 – 496	19.6326
497 – 504	15.9176
505 – 512	19.7059

Table 5-5 Weak Bit Ranges found by Impedance Test

Number of Rounds	**Less Weak Superpoly Input Bits**
256	225-232
256	241-248
256	265-272

5.8 Off By One Test

Under the null hypothesis, over all the input samples, when one of the superpoly input bits is flipped from 0 to 1 or 1 to 0, half the time the output bit should also flip and half the time the output bit should not flip. The counts for the chi-square test of output Q_i and input s_j are N_1 = observed number of times Q_i did not flip when s_j flipped, N_2 = observed number of times Q_i flipped when s_j flipped and $E_1 = E_2 = {}^{m}/_{2}$. Applying the off by one test to one cube test program run yields 256.s pass fail results, one for each combination of a superpoly output and a superpoly input. This test is in concept similar to Avalanche Test.

One subtlety in the off by one test is that the same occurance must not be counted twice. For example, suppose two of the m superpoly input samples happen to be 1101110 and 1001110 (s=7). Flipping the second bit in the first sample will cause the output bits to flip or not flip in the same way as flipping the second bit in the second sample. Thus, the outcomes from flipping the first bit for these two samples are not independent. In each case, the number of samples will be reduced by 1.

5.8.1 Off By One Test Methodology

Preprocessing rounds of SHA-256 have been removed as they add non-zero bits in the last 64 bits of the input block. In cube tester attack, all unused bits of input, which are not part of cube input or superpoly input should be zero. For balance test, size of cube input '*c*' and superpoly input '*s*' is set to 8 bits each. So out of 512 input bits, only 16 bits of input are manipulated for a single test run. Rest 496 bits are kept zeros.

Results of cube testers attack are fed to chi square test to test the variation is just result of a chance or there is significant difference in actual and expected results.

Degree of freedom used for this test is 1 as there are only two possible results. And value of p is kept to 0.01 as recommended for cryptographic algorithms. The resulting value is 6.635 as per the chi square table shown in Figure 5-1.

In binary every odd number is one bit different than the immediate smaller even number. In Off By One test, for 2^8 (=256) repetitions of superpoly , 128 comparisons are to be made. Comparison will be made such that output of one even repetition is saved. Then output of immediate next odd input is taken and both outputs are XORed. It is an XORing of two pairs of 256 bits each. Output 256 bits should contain 128 zeros and 128 ones as per the null hypothesis.

Total number of computations of SHA-256 done are 2^{25} ($=2^8 * 7 * 2^8 * 2^6$). Total number of comparisons done are 2^{13} ($=2^7 * 2^6$). Each comparison includes comparing 256 bits.

5.8.2 Off By One Test Results

Off by one test is applied on SHA-256 with Superpoly input and Cube Inputs are of 8 bits each. This test failed to find any non randomness in the output using Chi Square's lenient most level of significance which is 0.01 for cryptographic algorithms.

Only a very few number of samples gave significance level of non-randomness under Chi Square limits. On basis of those results one can't say that Off By One test found any nonrandomness in SHA-256.

Table 5-6 Chi SquareValues of Off By One Test with 17 Round SHA-256 to 23 Round SHA-256 for all Input Bit Ranges

Rounds / Bit Ranges	17	18	19	20	21	22	23
1 – 8	8.048	8.597	8.280	11.297	6.924	9.740	11.102
9 – 16	12.975	7.044	9.659	11.523	10.312	8.734	12.814
17 – 24	11.336	13.153	13.132	7.348	6.904	10.311	12.759

25 – 32	9.090	11.962	9.846	9.725	11.883	7.5200	11.059
33 – 40	10.187	9.158	11.275	12.006	10.723	7.2835	11.497
41 – 48	8.990	11.091	9.954	10.178	8.616	12.332	11.799
49 – 56	7.660	11.990	8.937	10.347	7.421	12.145	6.649
57 – 64	12.392	10.835	11.836	13.182	10.169	6.650	9.79
65 – 72	8.192	7.791	12.357	9.230	11.551	7.774	9.575
73 – 80	10.545	12.242	13.224	6.829	6.950	8.415	9.372
81 – 88	11.473	9.255	9.299	8.836	13.138	11.176	8.511
89 – 96	9.844	8.082	10.525	13.008	7.882	8.328	10.938
97 – 104	12.235	8.968	12.375	9.321	9.030	11.774	7.873
105 – 112	12.957	10.478	12.192	11.700	10.020	12.974	9.782
113 – 120	9.097	7.967	7.096	7.059	9.186	9.291	13.14
121 - 128	9.562	8.996	11.272	7.840	8.749	6.652	10.787
129 – 136	7.183	10.165	6.745	12.1205	11.780	7.864	12.463
137 - 144	10.787	9.805	9.303	8.149	8.847	11.617	6.727
145 – 152	6.792	10.105	12.340	10.078	8.321	13.058	7.025
153 – 160	12.967	8.771	8.316	7.594	7.054	8.68	11.135
161 – 168	11.082	12.454	9.596	8.774	8.909	10.688	11.48
169 – 176	12.327	10.554	8.139	9.820	10.756	8.885	12.101
177 – 184	8.046	7.371	12.391	12.397	8.263	12.808	8.989
185 – 192	7.464	11.090	8.748	12.037	7.060	8.447	7.606
193 – 200	11.731	12.018	9.194	7.034	12.101	7.043	11.000
201 – 208	8.420	12.419	11.961	8.794	10.125	8.294	12.229
209 – 216	8.394	12.709	10.354	8.358	7.229	10.04	9.907
217 – 224	12.658	10.696	7.731	10.264	6.929	8.417	8.831
225 – 232	7.982	6.829	9.947	7.1173	11.784	11.779	11.52
233 – 240	11.964	11.862	8.000	12.341	11.247	8.048	9.393
241 – 248	10.466	9.218	6.672	8.787	9.220	11.25	8.204
249 – 256	12.818	11.481	8.048	7.920	7.081	8.968	7.06
257 – 264	7.507	9.551	7.761	7.127	12.613	7.281	10.688
265 – 272	9.028	7.351	12.363	10.900	12.277	7.59	11.161
273 - 280	12.204	8.231	11.211	11.813	6.638	9.13	9.503
281 – 288	9.658	8.981	7.888	12.702	6.686	9.24	9.906
289 – 296	8.617	7.995	7.607	7.556	11.640	8.105	8.421
297 – 304	10.920	9.337	8.217	8.531	8.518	10.263	12.536
305 – 312	10.732	6.959	10.596	11.861	11.058	8.552	8.04
313 – 320	8.212	8.806	10.159	11.539	8.414	12.927	7.361
321 – 328	10.284	10.195	9.676	7.606	11.432	11.265	7.17
329 – 336	8.346	8.719	11.351	11.556	10.300	10.706	12.58
337 – 344	8.019	10.941	11.178	8.048	7.072	12.15	10.595
345 – 352	10.654	9.880	10.405	9.302	8.425	9.342	10.304
353 – 360	11.614	9.243	6.658	9.305	11.120	7.397	10.021
361 – 368	10.333	6.848	9.529	7.094	7.623	11.961	10.996
369 – 376	7.728	11.199	9.526	9.642	12.734	9.636	7.777
377 – 384	9.975	10.771	9.979	9.844	11.620	9.618	8.102
385 – 392	11.73	11.593	11.395	8.876	7.932	12.361	9.37
393 – 400	8.883	7.7632	9.726	12.996	9.518	7.809	13.06
401 – 408	9.287	8.3745	10.767	13.197	8.757	11.714	13.113

409 - 416	11.924	7.784	11.555	9.225	9.313	8.824	8.084
417 – 424	11.161	10.421	11.467	11.245	11.909	9.168	8.688
425 – 432	12.219	9.343	12.651	11.936	10.689	8.509	8.311
433 – 440	8.503	6.911	12.961	8.673	12.875	6.866	8.441
441 – 448	10.774	9.917	10.448	7.983	12.44	6.950	11.104
449 – 456	12.328	7.946	9.314	7.338	6.816	11.960	10.392
457 – 464	7.498	6.870	9.311	10.387	6.740	12.274	11.038
465 – 472	10.731	11.849	11.591	9.023	8.443	7.946	8.946
473 – 480	7.037	12.829	11.825	12.783	11.145	12.626	12.988
481 – 488	7.091	8.100	11.264	9.766	12.582	9.202	12.341
489 – 496	9.119	12.509	11.447	8.715	10.092	7.297	8.637
497 – 504	12.857	11.003	8.637	8.050	10.323	7.125	6.709
505 – 512	12.600	9.982	7.639	7.763	6.821	9.906	12.158

Table 5-7 Chi SquareValues of Off By One Test with 24 Round SHA-256 to 30 Round SHA-256 for all Input Bit Ranges

Rounds / Bit Ranges	24	25	26	27	28	29	30
1 – 8	7.066	9.96597	8.0791	9.993	11.271	8.438	12.498
9 – 16	9.952	12.4588	9.281	10.818	6.645	6.823	9.166
17 – 24	7.446	9.65393	7.249	7.547	8.022	10.396	9.780
25 – 32	9.556	9.63064	8.759	12.428	6.697	7.225	12.416
33 – 40	10.162	8.31425	7.615	11.335	10.129	12.766	12.315
41 – 48	12.042	6.94623	8.784	7.564	7.695	8.446	8.424
49 – 56	11.286	12.3019	11.622	8.892	9.211	13.193	11.490
57 – 64	7.384	6.71944	8.625	12.139	11.740	7.153	8.743
65 – 72	8.984	12.3306	10.338	12.374	10.444	6.849	12.598
73 – 80	10.869	11.0827	9.155	9.479	9.103	8.005	12.427
81 – 88	12.909	11.5835	11.331	11.179	11.102	12.965	7.275
89 – 96	9.917	12.019	7.301	11.315	7.894	8.360	10.566
97 – 104	7.173	11.6853	6.816	12.514	9.050	7.430	8.559
105 – 112	7.809	12.6293	11.485	8.563	8.088	9.538	7.207
113 – 120	7.667	11.3403	8.434	11.388	9.125	10.636	7.939
121 - 128	9.472	9.78413	12.484	10.486	11.127	12.081	9.099
129 – 136	9.768	9.879	8.644	12.501	7.664	8.730	12.069
137 - 144	11.972	9.533	12.917	12.114	13.184	6.730	10.552
145 – 152	11.706	8.959	11.908	8.769	12.914	13.153	8.899
153 – 160	9.623	10.939	12.616	11.521	9.571	10.238	7.766
161 – 168	12.534	8.003	7.704	11.212	7.387	13.262	7.344
169 – 176	13.255	13.043	10.290	6.765	8.494	7.431	10.069
177 – 184	7.638	10.084	10.957	10.699	11.731	7.205	10.951
185 – 192	9.904	9.783	11.344	7.508	8.353	8.728	10.496

193 – 200	12.229	12.712	9.597	6.950	11.045	9.254	6.776
201 – 208	12.111	11.333	6.806	8.491	7.212	12.628	12.137
209 – 216	11.976	9.175	12.242	12.892	10.459	9.876	6.934
217 – 224	9.995	9.949	9.553	9.774	8.720	12.299	12.788
225 – 232	11.428	8.917	12.109	6.803	10.353	10.199	12.240
233 – 240	12.531	12.459	11.138	8.841	11.340	8.118	9.541
241 – 248	7.450	10.779	6.890	12.233	8.629	8.175	6.674
249 – 256	7.927	7.888	12.275	10.227	12.053	11.056	10.660
257 – 264	8.178	6.869	8.501	11.067	11.189	9.908	8.959
265 – 272	7.468	10.080	8.460	9.208	9.669	10.589	10.656
273 - 280	10.788	10.422	8.478	7.645	10.625	7.437	9.989
281 – 288	9.898	11.184	8.461	12.744	10.001	11.924	12.785
289 – 296	10.497	11.253	12.547	9.461	7.625	8.401	9.563
297 – 304	9.165	10.679	12.247	13.120	10.088	7.376	10.936
305 – 312	12.410	10.002	7.932	7.451	12.514	7.164	6.775
313 – 320	12.944	13.144	7.578	11.939	11.538	9.103	11.214
321 – 328	12.878	8.679	12.611	9.228	11.061	8.459	10.039
329 – 336	7.652	12.174	10.445	9.115	12.717	10.7586	9.788
337 – 344	12.206	13.116	13.099	11.921	7.196	11.507	6.849
345 – 352	11.263	7.616	7.992	13.141	8.450	9.856	8.774
353 – 360	8.017	6.993	10.908	12.399	11.941	12.093	7.898
361 – 368	9.393	12.910	7.766	13.238	12.793	12.981	6.662
369 – 376	7.256	8.853	12.275	9.353	13.040	12.390	12.695
377 – 384	7.318	11.913	7.680	11.827	9.678	9.657	12.109
385 – 392	10.733	10.461	8.816	8.777	6.945	9.661	10.608
393 – 400	12.812	7.299	10.86	11.296	12.810	9.786	13.073
401 – 408	9.693	8.936	9.546	7.481	8.455	12.666	10.766
409 - 416	12.168	9.301	11.823	13.027	12.83	13.129	8.820
417 – 424	8.022	10.677	8.114	8.070	8.943	10.779	12.108
425 – 432	12.961	7.289	12.164	9.262	6.774	11.347	7.767
433 – 440	10.300	7.380	11.056	12.478	9.806	7.401	7.283
441 – 448	9.536	11.130	8.501	8.876	10.18	8.079	10.194
449 – 456	11.311	9.993	8.449	11.854	6.716	10.245	11.432
457 – 464	11.487	12.720	11.600	12.054	8.954	9.313	11.081
465 – 472	12.466	7.186	10.504	10.517	9.217	10.584	11.691
473 – 480	7.379	10.754	13.022	8.230	10.126	8.962	12.093
481 – 488	12.036	11.313	11.194	8.513	10.702	11.096	11.433
489 – 496	11.383	9.600	12.465	12.999	10.378	11.308	6.644
497 – 504	12.290	9.786	12.236	10.148	9.697	12.172	6.962
505 – 512	12.137	9.404	12.734	9.881	12.333	10.723	8.576

5.9 Conclusion

The three tests (balance test, impedance test and Off By One test) applied on SHA-256 are discussed in this chapter. The theory, attack methodology and results are

included in this chapter. Statistical standard, Chi Square Test, which is used for deducting conclusion on the attacks is also discussed in this chapter.

Balance Test succeeded in finding non-randomness over 25 rounds of main function of SHA-256 out of 64 rounds. Impedance test found randomness with complete 64 rounds of SHA-256. While Off By One test failed to find any non-randomness as per the criteria defined in Chi Square test.

Chapter 6

Conclusion and Future Work

6.1 Introduction

In this chapter, the thesis has been concluded. Some possible enhancements of the work have been given in Section 6.3.

6.2 Conclusion

Cube Testers Attack is a relatively newer technique of cryptanalysis and its application on different new ciphers is important. A platform has been developed for the application of cube attack to any hash function.

The three tests (balance test, impedance test and Off By One test) applied on SHA-256 are discussed in chapter 6. Balance Test succeeded in finding non-randomness over 25 steps of main function of SHA-256 out of 64 steps. Impedance found randomness with complete 64 steps of SHA-256. While Off By One test failed to find any non-randomness as per the criteria defined in Chi Square test.

6.3 Future Work

Cube Testers can be used to check non random behavior in other hash functions especially light weight hash functions. This test is yet to be tested against Keccak, the SHA-3 standard hash function. Using same framework, properties other than those tested in this thesis can also be tested against SHA-256 and other advanced hashes.

Better results can be found against balanceness and Off by one test by increasing the size of cube input and superpoly input. For this better hardware resources are required. Parallel Computing can also help in this regard.

This platform can be used to help testing ciphers, block and especially stream ciphers for non-randomness. And weaknesses found by Cube Testers can be used to predict result of advanced attacks like Dynamic Cube Attacks on ciphers.

6.4 Summary

In this chapter, the thesis has been concluded and future work is proposed which mainly focuses on increasing the sizes of superpoly and cube samples, testing other properties. Testing light weight ciphers and hashes against these properties.

APPENDICES

Appendix-A

C++ Code for Balance Test on SHA-256

```
//Implementation of SHA 256 for BALANCE TEST

#include <conio.h>
#include<iostream>
#include<vector>
#include<fstream>
#include<exception>
#include<string>
#include <sstream>
#include <atlbase.h>
#include <atlstr.h>
#include <winbase.h>
#include <math.h>

//#define const1 50
//#define const2 20
#define const1 45
#define const2 33
using namespace std;

typedef unsigned int uint;


void BalanceTestWithChiSqr(char* Filename)
{
	ifstream infile;
	ofstream outfile;
	char temp;
	u_int counter1 = 0, counter0 = 0;
	u_int expected_freq = 0;
	double X0 = 0.00, X1 = 0.00;
	double temp3 = 0.0, temp4 = 0.0;

	expected_freq = 65536 / 2;
	//open and read file

	infile.open(Filename, ios::binary | ios::app);

	//if file exists then proceed
	if (!infile)
	{
		//error
		cout << "\nError opening file for balance test\n";
```

```
        }
        else
        {
            while (infile.read(&temp, 1))
            {
                if (temp == 0x31)
                {
                    counter1++;
                }
                else if (temp == 0x30)
                {
                    counter0++;
                }
            }
        }
        infile.close();
            if (counter0 < expected_freq)
            temp3 = (counter0 - expected_freq)*(-1);
        else
            temp3 = (counter0 - expected_freq);


        if (counter1 < expected_freq)
            temp4 = (counter1 - expected_freq)*(-1);
        else
            temp4 = (counter1 - expected_freq);


        X0 = 2 * (pow(temp3, 2) / expected_freq);
        X1 = pow(temp4, 2) / expected_freq;
        outfile.open("Balancetest.txt", ios::binary | ios::app);
        if (!outfile)
        {
            cout << "\nError opening balancetest.txt file\n";
        }
        else
        {
            outfile << X0 << endl;
            outfile << X1 << endl;
            outfile.close();
        }
}
string fromDecimal(uint n, int b)
{
        string chars="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        string result="";
        while(n>0)
        {
            result=chars.at(n%b)+result;
            n/=b;
```

```
        }

        return result;
}

        uint K[]=
        {       0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b,
        0x59f111f1, 0x923f82a4, 0xab1c5ed5,
  0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe,
        0x9bdc06a7, 0xc19bf174,
  0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa,
        0x5cb0a9dc, 0x76f988da,
  0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147,
        0x06ca6351, 0x14292967,
  0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb,
        0x81c2c92e, 0x92722c85,
  0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624,
        0xf40e3585, 0x106aa070,
  0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a,
        0x5b9cca4f, 0x682e6ff3,
  0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb,
        0xbef9a3f7, 0xc67178f2};

   void makeblock(vector<uint>& ret, string p_msg)
        {
             uint cur=0;
             int ind=0;
             for(uint i=0; i<p_msg.size(); i++)
             {
                  cur = (cur<<8) | (unsigned char)p_msg[i];

                  if(i%4==3)
                  {
                       ret.at(ind++)=cur;
                       cur=0;
                  }
             }
        }

class Block
{
        public:
        vector<uint> msg;

        Block():msg(16, 0) { }


        Block(string p_msg):msg(16, 0)
        {
             makeblock(msg, p_msg);
```

```
        }

};


void split(vector<Block>& blks, string& msg)
{
        for(uint i=0; i<msg.size(); i+=64)
        {

try
{
                makeblock(blks[i/64].msg, msg.substr(i, 64));
    }
    catch(...)
        }
}
string mynum(uint x)
{
 string ret;
  for(uint i=0; i<4; i++)
  ret+=char(0);

  for(uint i=4; i>=1; i--) //big endian machine used
  {
       ret += ((char*)(&x))[i-1];
  }
  return ret;
}
uint ch(uint x, uint y, uint z)
{
 return (x&y) ^ (~x&z);
}
uint maj(uint x, uint y, uint z)
{
 return (x&y) ^ (y&z) ^ (z&x);
}
uint fn0(uint x)
{
 return rotr(x, 2) ^ rotr(x, 13) ^ rotr(x, 22);
}

uint fn1(uint x)
{
 return rotr(x, 6) ^ rotr(x, 11) ^ rotr(x, 25);
}

uint sigma0(uint x)
{
        return rotr(x, 7) ^ rotr(x, 18) ^ shr(x, 3);
```

```
}

uint sigma1(uint x)
{
	return rotr(x, 17) ^ rotr(x, 19) ^ shr(x, 10);
}


void sha256(string msg_arr,uint *H)
{
	string msg;

	msg=msg_arr;

	uint num_blk = msg_arr.size()*8/512;
	vector<Block> M(num_blk, Block());
	split(M, msg_arr);


	    for(uint i=0; i<1; i++)
	    {
		vector<uint> W(64, 0);
		for(uint t=0; t<16; t++)
		{
			W[t] = M[i].msg[t];
		}

		for(uint t=16; t<64; t++)
		{
		W[t] = sigma1(W[t-2]) + W[t-7] + sigma0(W[t-15]) + W[t-16];
		}

		uint work[8];
		for(uint i=0; i<8; i++)
		work[i] = H[i];
		for(uint t=0; t<17; t++)
		{
			uint t1, t2;
	t1 = work[7] + fn1(work[4]) + ch(work[4], work[5], work[6]) + K[t] + W[t];
			t2 = fn0(work[0]) + maj(work[0], work[1], work[2]);
			work[7] = work[6];
			work[6] = work[5];
			work[5] = work[4];
			work[4] = work[3] + t1;
			work[3] = work[2];
			work[2] = work[1];
			work[1] = work[0];
			work[0] = t1 + t2;
		}
		for(uint i=0; i<8; i++)
```

```
			{
			H[i] = work[i] + H[i];
		}
			}
}
int main()
{	string
	msg_arr="0000000000000000000000000000000000000000000000000
	00000000000000000000000000000000000000000000000000000000000
	00000000000";
	// 128 zeros
	string msg;
	uint H_Prev[8]={0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
	0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19};
	string
	temp_msg_arr="0000000000000000000000000000000000000000000
	00000000000000000000000000000000000000000000000000000000000
	000000000000000";
	//128 zeros
	char temp;
	ofstream outfile;
	ofstream outfile2;
	ofstream outfile3;  //addition
	ifstream infile;  //addition
	char* FileName="input.txt";
	char* OutFileName="Output.txt";
	char* OutFileSHA2Name="SHA2Out.txt";
	outfile.open(OutFileName, ios::binary);
	outfile.close();
	outfile3.open("Balancetest.txt", ios::binary);
	outfile3.close();
	outfile.open(FileName, ios::binary);
		{
				char temparr[128];
				for(int i=0;i<128;i++)
					temparr[i]=0x00;
				for(int k=0;k<128;k++)
				{
					outfile.write(&temparr[k],1);
				}
	}
		outfile.close();
		infile.open(FileName, ios::binary);
				{
			int cnt=0,cnt_arr=0;
			msg_arr[0];
			for(int i=0;cnt_arr<128;i=i+2)
			{
				if(cnt==0)
				{
```

```
                infile.read(&temp,1);
                msg_arr[cnt_arr]=temp<<4;
                cnt=1;
            }
            if(cnt==1)
            {
                infile.read(&temp,1);
                msg_arr[cnt_arr]=msg_arr[cnt_arr]|temp;
                cnt_arr++;
                cnt=0;
            }

        }

    }
    infile.close();
        msg=msg_arr;

        for(int k=0;k<256;k++)
        {
            for(int l=0;l<256;l++)
            {

                uint H[]={0x6a09e667, 0xbb67ae85, 0x3c6ef372,
0xa54ff53a, 0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19};
                sha256(msg,H);
                                        for(int m=0;m<8;m++)
                {
                    H_Prev[m]=H[m]^H_Prev[m];
                }
                msg[const1]++;

            }

        outfile.open(OutFileName, ios::binary|ios::app);
                if (!outfile)
            {
                        //error
            }
            else
            {
                for(int k=0;k<8;k++)
                {
                    uint tmp=0x00000000;
                    tmp=H_Prev[k];
                    for(int cntr=0;cntr<32;cntr++)
                    {
                        char ch=0x00;

                        ch= (tmp & 0x80000000)>>31;
```

```
                                        ch=ch+'0x30';

                                        tmp=tmp<<1;
                                        outfile<<ch;
                                }
                        }
                }
                outfile.close();
                                        for(int m=0;m<8;m++)
                {
                        H_Prev[m]=0x00;
                }

                msg[const2]++;
                                }

        BalanceTestWithChiSqr(OutFileName);
        cout<<"Test Done";
        getch();
        return 0;
}
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] A. Kaminsky, "Cube test analysis of the statistical behavior of cubehash and skein," 2010, ark@cs.rit.edu 14736 received 7 May 2010. [Online]. Available: http://eprint.iacr.org/2010/262

[2] B. Zhu, G. Gong, X. Lai, and K. Chen, "Another view on cube attack, cube tester, aida and higher order differential cryptanalysis."

[3] J.-P. Aumasson, I. Dinur, L. Henzen, W. Meier, and A. Shamir, "Efficient fpga implementations of high-dimensional cube testers on the stream cipher grain-128," Cryptology ePrint Archive, Report 2009/218, 2009, http://eprint.iacr.org/

[4] J.-P. Aumasson, I. Dinur, W. Meier, and A. Shamir, "Cube testers and key recovery attacks on reduced-round md6 and trivium," in Fast Software Encryption, ser. Lecture Notes in Computer Science, O. Dunkelman, Ed. Springer Berlin Heidelberg, 2009, vol. 5665, pp. 1–22.

[5] S. Li, Y. Wang, and J. Peng, "Cube testers on bivium," in Communications and Intelligence Information Security (ICCIIS), 2010 International Conference on, oct. 2010, pp. 121 –124.

[6] I. Dinur, T. Gneysu, C. Paar, A. Shamir, and R. Zimmermann, "An experimentally verified attack on full grain-128 using dedicated reconfigurable hardware," in Advances in Cryptology ASIACRYPT 2011, ser. Lecture Notes in Computer Science, D. Lee and X. Wang, Eds. Springer Berlin Heidelberg, 2011, vol. 7073, pp. 327–343.

[7] J.-P. Aumasson, "Practical distinguisher for the compression function of blue midnight wish," 2010. [Online]. Available: http://131002.net/data /papers/Aum10.pdf

[8] N. R. Darmian, "A distinguish attack on rabbit stream cipher based on multiple cube tester." IACR Cryptology ePrint Archive, vol. 2013, p. 780, 2013.

[9] F.-M. Quedenfeld and C. Wolf, "Algebraic properties of the cube attack," Cryptology ePrint Archive, Report 2013/800, 2013, http://eprint.iacr.org/.

[10] L. Joel, "Cube attacks on cryptographic hash functions," 2009. [Online]. Available: http://hdl.handle.net/1850/10821

[11] Nikova, "New developments in symmetric key cryptanalysis," in D.SYM.6 ECRYPT II, 2010.

[12] S. Rao, “Parallel cube testing on gpus,” 2010, RIT Computer Science M.S. project, May 2010. [Online]. Available: http://www.cs.rit.edu/ ark/students/bwb1636/index.shtml

[13] J.-P. Aumasson, E. Ksper, L. Knudsen, K. Matusiewicz, R. degrd, T. Peyrin, and M. Schlffer, “Distinguishers for the compression function and output transformation of hamsi-256,” in Information Security and Privacy, ser. Lecture Notes in Computer Science, R. Steinfeld and P. Hawkes, Eds. Springer Berlin Heidelberg, 2010, vol. 6168, pp. 87–103.

[14] S. Nikova, “New developments in symmetric key cryptanalysis,” in D.SYM.9 ECRYPT II, 2011.

[15] I. Dinur and A. Shamir, “Breaking grain-128 with dynamic cube attacks,” in Fast Software Encryption, ser. Lecture Notes in Computer Science, A. Joux, Ed. Springer Berlin Heidelberg, 2011, vol. 6733, pp. 167–187.

[16] I. Dinur, P. Morawiecki, J. Pieprzyk, M. Srebrny, and M. Straus, “Practical Complexity cube attacks on round-reduced keccak sponge function,” Cryptology ePrint Archive, Report 2014/259, 2014, http://eprint.iacr.org/.

[17] J.-P. Aumasson, “On the pseudorandomness of shabal’s keyed permutation,” Available online, 2009. [Online]. Available: http://131002.net/data/papers/Aum09.pdf

[18] M. Lamberger and F. Mendel, “Higher-order differential attack on reduced sha-256,” Cryptology ePrint Archive, Report 2011/037, 2011, http://eprint.iacr.org/.

[19] M. Iwamoto, T. Peyrin, and Y. Sasaki, “Limited-birthday distinguishers for hash functions,” in Advances in Cryptology - ASIACRYPT 2013, ser. Lecture Notes in Computer Science, K. Sako and P. Sarkar, Eds. Springer Berlin Heidelberg, 2013, vol. 8270, pp. 504–523.

[20] J.-P. Aumasson and D. Khovratovich, “First analysis of keccak,” Available online, 2009. [Online]. Available: http://131002.net/data/papers/AK09.pdf

[21] E. Filiol, “A new statistical testing for symmetric ciphers and hash functions,” in Information and Communications Security, ser. Lecture Notes in Computer Science, R. Deng, F. Bao, J. Zhou, and S. Qing, Eds. Springer Berlin Heidelberg, 2002, vol. 2513, pp. 342–353.

[22] D. Khovratovich, C. Rechberger, and A. Savelieva, “Bicliques for preimages: Attacks on skein-512 and the sha-2 family,” in Fast Software Encryption, ser. Lecture Notes in Computer Science, A. Canteaut, Ed. Springer Berlin Heidelberg, 2012, vol. 7549, pp. 244–263.

[23] O. K. Ali Doanaksoy, Bar Ege and F. Sulak, “Cryptographic randomness testing of block ciphers and hash functions,” Cryptology ePrint Archive, Report 2010/564, 2010, http://eprint.iacr.org/.

[24] K. Aoki, J. Guo, K. Matusiewicz, Y. Sasaki, and L. Wang, “Preimages for step-reduced sha-2,” in Advances in Cryptology ASIACRYPT 2009, ser. Lecture Notes in Computer Science, M. Matsui, Ed. Springer Berlin Heidelberg, 2009, vol. 5912, pp. 578–597.

[25] S. Sanadhya and P. Sarkar, “Attacking reduced round sha-256,” in Applied Cryptography and Network Security, ser. Lecture Notes in Computer Science, S. Bellovin, R. Gennaro, A. Keromytis, and M. Yung, Eds. Springer Berlin Heidelberg, 2008, vol. 5037, pp. 130–143.

[26] Jian Guo, San Ling, Christian Rechberger, and Huaxiong Wang (2010). "Advanced meet-in-the-middle preimage attacks: First results on full Tiger, and improved results on MD4 and SHA-2". *Advances in Cryptology - ASIACRYPT 2010*. Lecture Notes in Computer Science (Springer Berlin Heidelberg) **6477**: pp. 56–75.

[27] L. Yang, M. Wang, and S. Qiao, “Side channel cube attack on present,” in Cryptology and Network Security, ser. Lecture Notes in Computer Science, J. Garay, A. Miyaji, and A. Otsuka, Eds. Springer Berlin Heidelberg, 2009, vol. 5888, pp. 379–391.