2009

Cube attacks on cryptographic hash functions

Joel Lathrop

Rochester Institute of Technology

Department of Computer Science

Master's Thesis in Computer Science

Cube Attacks on Cryptographic Hash Functions

---

by

Joel Lathrop

Chair:	Stanisław Radziszowski
Reader:	Christopher Homan
Observer:	Edith Hemaspaandra

May 21, 2009

Abstract

Cryptographic hash functions are a vital part of our current computer systems. They are a core component of digital signatures, message authentication codes, file checksums, and many other protocols and security schemes. Recent attacks against well-established hash functions have led NIST to start an international competition to develop a new hashing standard to be named SHA-3.

In this thesis, we provide cryptanalysis of some of the SHA-3 candidates. We do this using a new cryptanalytical technique introduced a few months ago called cube attacks. In addition to summarizing the technique, we build on it by providing a framework for estimating its potential effectiveness for cases too computationally expensive to test. We then show that cube attacks can not only be applied to keyed cryptosystems but also to hash functions by way of a partial preimage attack. We successfully apply this attack to reduced-round variants of the ESSENCE and Keccak SHA-3 candidates and provide a detailed analysis of how and why the cube attacks succeeded. We also discuss the limits of theoretically extending these attacks to higher rounds. Finally, we provide some preliminary results of applying cube attacks to other SHA-3 candidates.

Contents

Chapter 1

Introduction

1.1 The Problem

Cryptographic hash functions are a vital part of our current computer systems. They are a core component of digital signatures, message authentication codes, file checksums, and many other protocols and security schemes. Because of this, their cryptographic strength is paramount to the continuance of computer security as we know it.

Unfortunately, that strength has recently been weakened. In 2005 Wang et. al. released details on how to break the widely-used MD5 hash function by finding colliding messages [40]. This was followed by an attack against the SHA-1 hash function, which while not producing a practical break did significantly reduce the complexity of finding one [39]. With both MD5 and SHA-1—the two most widely-used hash functions—weakened by successful attacks, it became vital to provide more secure hash functions in order to protect the continuation of effective cryptography in computer security.

To this end, the National Institute of Standards and Technology announced a cryptographic hash algorithm competition to produce a new hash family which they would standardize as SHA-3 [26]. The purpose of this competition is to provide a hash function which has undergone extensive scientific and public review to verify its strength and effectiveness. The competition was announced on November 2, 2007 [27] and is scheduled to choose a final winner in 2012 [25].

The selection of a winner will involve extensive public review of the candidates. While NIST will be forming an internal committee to review the

hash functions, detailed cryptanalysis and comments from the public will be a vital part of their review.

1.2 Our Contributions

We first provide a detailed history of the events that led up to the SHA-3 hash function competition. We then describe a new type of cryptanalytical attack devised by Itai Dinur and Adi Shamir called a *cube attack*. We build on Dinur and Shamir's description of cube attacks by describing a way to analyze a cryptographic function to estimate the maximum difficulty of a cube attack even if that difficulty is so high that it would not be practical to actually try the attack.

Finally, we describe successful, practical cube attacks against reduced-round versions of the SHA-3 candidates ESSENCE and Keccak. We provide the cubes used to reproduce these attacks and an analysis of the ESSENCE and Keccak round functions with the goal of explaining why the attacks were successful and how far they could be theoretically extended. We also briefly describe our results from applying cube attacks to reduced-round variants of several other SHA-3 candidates which proved resistant.

1.3 Overview

In Chapter 2 we provide basic cryptographic background knowledge necessary this thesis depends on. In Chapter 3 we describe the history of events leading up to the NIST SHA-3 competition. Cube attacks are introduced in Chapter 4 and our method for estimating the complexity of a cube attack is described in Chapter 5. In Chapter 6 we describe various cube attacks against reduced variants of SHA-3 candidates, specifically successful attacks against reduced variants of ESSENCE and Keccak. SHA-3 candidates that proved resistant to attack are listed in Section 6.3. We describe work related to cube attacks and hash functions in Chapter 7. Finally, in Chapter 8 we give some concluding remarks.

Chapter 2

Preliminaries

2.1 Notation

Throughout this document we use the following notation:

- Numbers are presumed to be decimal unless stated otherwise. If a number is expressed in a different base, the number will be given with the base in a subscript and most significant digits on the left. For example, 10110101_2 is the binary form of the decimal number 181.
- $\neg X$ is bitwise negation of X.
- $X \wedge Y$ is the bitwise AND of X and Y.
- $X \vee Y$ is the bitwise OR of X and Y.
- $X \oplus Y$ is the bitwise exclusive OR of X and Y.
- $X \ll i$ is X shifted i bits to the left and padded with 0's on the right. If X is a fixed width integer, then as many as i of the most significant bits may be lost.
- $X \gg i$ is X shifted i bits to the right and padded with 0's on the left. If X is a fixed width integer, then the i least significant bits will be lost.
- $X \lll i$ is X rotated i bits to the left.
- $X \ggg i$ is X rotated i bits to the right.

- $X||Y$ is the concatenation of X and Y.

2.2 Hash Functions

A hash function[1] is a cryptographic primitive that compresses an arbitrary length input into a fixed length output called a message digest. It does this in such a way that the output is effectively unique[2] with regard to the input, and the process cannot be reversed to yield the input from the output.

In more specific mathematical terms, a hash function can be defined as:

$$h : \{0,1\}^* \rightarrow \{0,1\}^m$$

where m is the fixed length of the hash function h in bits [32].

The output of h must be effectively unique. This means that a computation that produces an x and x' such that $x \neq x'$ and $h(x) = h(x')$ must take at least $2^{m/2}$ hash operations, which is approximately the number of hash operations in which an x and x' could be found by random search alone[3] [32].

The hash function h must also be irreversible (also known as being "one-way"). This means that given a message digest y such that $y = h(x)$, computing x from y should require no less than the work equivalent to 2^m hash operations, which is the number of hash operations necessary to find x by exhaustive guessing.

2.2.1 Hash function security problems

In order to maintain these properties and be considered cryptographically secure, a hash function must not be susceptible to the following attacks [36]:

Preimage Given a hash function h and a hash value y, find message value x such that $h(x) = y$.

[1]For the purposes of this discussion, when we say "hash function" we will be referring to an unkeyed, one-way hash function.

[2]By "effectively unique" we mean that while other inputs which would produce the same output may exist, the likelihood of finding such an input is so negligible that it is not pragmatically worth considering.

[3]This is known as a "birthday attack." For more information on the math that results in these probabilities, see `http://en.wikipedia.org/wiki/Birthday_attack` or Section 7.4 of [32].

Second-Preimage Given a hash function h and a message value x, find another message value x' such that $x \neq x'$ and $h(x) = h(x')$.

Collision Given a hash function h, find two message values x and x' such that $x \neq x'$ but $h(x) = h(x')$.

Length Extension Given a hash function h, a hash value $h(x)$, and the message length $|x|$, find an x' such that $h(x||x')$ can be calculated.

In addition to these, an ideal hash function must also be partial preimage resistant [23, §9.2.6]:

Partial Preimage Resistance For some hash function h, given a hash value $y = h(x)$ it should be just as hard to recover a substring of x from y as to recover all of x. Furthermore, if all but t bits of x are known, it should still take on average 2^{t-1} hash operations to find these bits.

In this thesis, we will perform partial preimage attacks in the following manner: Given a hash function h and a function $g(x) = h(y||x)$ where y is secret but g is available to the attacker, find y.

2.2.2 Merkle-Damgård chaining scheme

Many hash functions—including MD5 and SHA-1—are constructed using the Merkle-Damgård chaining scheme. In this scheme, a hash function h is built up from a compression function f such that if f is collision resistant, then h is as well [36].

This is done as follows. Given a compression function $f : \{0,1\}^m \times \{0,1\}^t \to \{0,1\}^m$ for $m, t \geq 1$ and an input x of length n, split x into t size blocks such that $x = x_1||x_2||x_3||\cdots||x_k$. If the last block x_k is less than t bits in length, right pad it with 0's.[4] Then append a final block x_{k+1} which contains the right-justified binary representation of n. The compression function f is then used to chain the input blocks together by computing

[4]Some hashes which claim to be using a Merkle-Damgård chaining scheme will append a '1'-bit to the message before the 0-padding. Strictly speaking, this is not a pure Merkle-Damgård chaining scheme but instead is a different construction similar to that used in SHA-1 [28]. However, this variation is common enough that it deserves mention.

$H_i = f(H_{i-1}, x_i), 1 \leq i \leq k+1$. The initial value H_0 is generally a special constant which is part of the hash function's definition. The hash function result for input x can now be iteratively defined as $h(x) = H_{k+1} = f(H_k, x_{k+1})$. In this way, the hash function h has been built from the compression function f via the Merkle-Damgård chaining scheme [23]. A visual representation of the Merkle-Damgård chaining scheme is given in Figure 2.1.

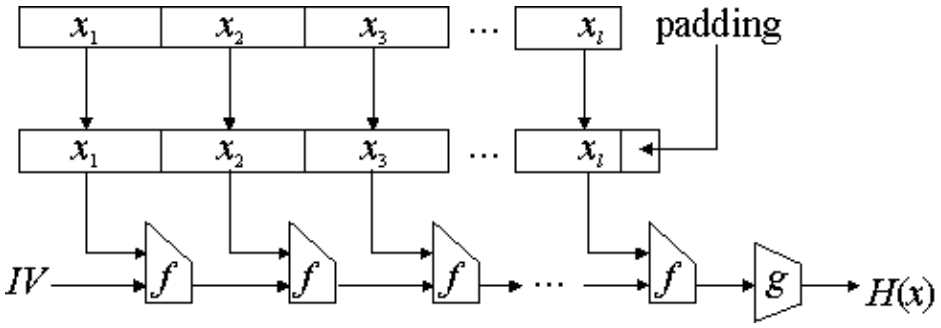

Figure 2.1: Merkle-Damgård construction [20].

Chapter 3

History of the NIST SHA-3 Competition

Cryptographic hash functions are an integral component of the grand scheme of cryptography. In the past few decades, much focus has been given to block cipher design, with hash functions somewhat relegated to the sidelines. However, events in the past few years have thrown hash functions into the focus of the international cryptographic community. To understand the work presented in this thesis in context, it is necessary to review these events.

3.1 The Breaking of MD5 and Weakening of SHA-1

MD5 is a hash function devised by Ron Rivest and published in 1992 [30], eventually becoming one of the most widely used hash functions in existence. While weaknesses were found in its compression function in 1996 [15], the full MD5 hash function remained secure. The discovery of these weaknesses did however provoke the creation of a more secure, yet very similar, hash called SHA-1.

3.1.1 Wang et. al.'s attack and improvements

In 2004, a group of Chinese researchers led by Xiaoyun Wang broke MD5 [40]. Through a slight variation on classical differential cryptanalysis they were able to construct colliding two-block message inputs. This was accomplished

by constructing two connected differential paths of all 64 rounds of MD5 for the two message block input. Using this technique, Wang's team was able to generate collisions for MD5 in about one hour on an IBM P690 supercomputer. This attack was also able to find collisions in HAVAL-128, MD4, RIPEMD, and SHA-0 [37, 38, 41].

The announcement of this attack generated quite a stir in the cryptographic community. About one year after Wang's attack was published in 2005, numerous improvements on the attack had been made by the cryptographic community. In March 2006, Black et. al. published a paper which combined the existing improvements on the attack such that colliding two-block message pairs could be generated in an average of 11 minutes on commodity PC hardware [10]. Around the same time, Vastimil Klima released a paper in which he introduced a new technique he called "tunneling" which reduced the search time necessary to find an MD5 collision to about 31 seconds on commodity hardware [17]. In June 2007, M.M.J. Stevens published his Master's thesis in which he detailed how to generate collisions in about 6 seconds on commodity hardware [34]. In his thesis, he also detailed an algorithm for automatically generating new differential paths for MD5 and an algorithm for generating chosen-prefix collisions for MD5.

While one segment of the cryptographic community was improving the speed of the MD5 attack, others were devising ways of applying it to demonstrate new weaknesses in computer security infrastructure. In a paper entitled "MD5 To Be Considered Harmful Someday" [16], Dan Kaminsky outlined methods to attack system auditing software (such as Tripwire), intrusion detection systems, Digital Rights Management systems, and peer-to-peer file sharing networks using only the original example collisions that Wang et. al. first published. These attacks utilized the properties of MD5's Merkle-Damgård chaining structure which allows an arbitrary set of blocks to be appended to each input in the collision pair while still maintaining the collision.

Another important set of applied attacks was created by Lenstra, Wang, and de Weger who devised a method to generate X.509 certificates with colliding RSA signatures by constructing the certificates' public keys using MD5 collisions [19, 18]. Stevens extended this attack using his chosen-prefix collisions to generate colliding X.509 certificates with different identities [34]. In December 2008, a team including Stevens and Lenstra used these X.509 attacks to create forged SSL certificates that were authentically signed by a trusted Certificate Authority [33, 35]. Their work involved improvements

on the existing cryptographic attack, employed some very clever engineering, and targeted design flaws in automatic Certificate Authorities.

By the end of 2008, MD5 had been so severely broken, both theoretically and practically, that it was unquestionably no longer fit for use. Unfortunately, there are still a wide variety of applications and protocols which use MD5, and this can lead to dangerous, real-world attacks like the X.509 certificate attack just mentioned.

3.1.2 Extension to SHA-1

With MD5 broken, SHA-1 [28] was the next hash function to turn to. SHA-1 was similar to MD5 but had some features to its design which made it appear more secure. In 1996, Hans Dobbertin found a collision for the MD5 compression function [15]. While this did not compromise the security of the full MD5 hash function, it was enough for some people to begin using SHA-1 instead of MD5. As a result, when MD5 was broken, the infrastructure to switch to SHA-1 was already in place, and in fact a number of systems that were already using SHA-1 were not adversely affected.

It turned out though that SHA-1 was susceptible to the same attack used against MD5, albeit to a lesser degree. Wang, Yin, and Yu detailed an attack that could find collisions in full 80-round SHA-1 using less than 2^{69} hash compressions, which is less than the 2^{80} of a brute force attack [39]. This constituted a theoretical break of SHA-1 but was still a ways from a practical break that could be executed with modern hardware. De Cannière and Rechberger advanced this by devising an algorithm to find collisions for 64-round SHA-1 in 2^{35} hash compressions on average and generated an example collision [13]. They then extended this to 70-round SHA-1 in 2^{44} hash compressions [12]. These developments appeared to bring an attack against the full 80-round SHA-1 within reach. At the time of this writing, a massive joint computing effort is taking place to find the first SHA-1 collision [1]. Even more promising though is a recent result presented at the rump session of EuroCrypt 2009 [11], in which the presenters gave a complete differential path which yielded an attack of complexity 2^{52} on the full SHA-1, a dramatic improvement from the previous best result.

3.2 NIST's Call for Hash Function Designs

Since their creation, MD5 and SHA-1 had become incredibly widely used hash functions, arguably the most widely used within the public domain. With the break of MD5 and a theoretical break of SHA-1 that was teetering on the edge of producing a full collision, it quickly became clear that replacements must be found. The SHA-2 family of hashes had been released in 2001 by the National Institute of Standards and Technology (NIST) [29], but their design was based on principles similar to those used in MD5 and SHA-1. This raised concerns that they may eventually be found vulnerable to the same class of attacks that had been used against MD5 and SHA-1. If this happens, it would have catastrophic effects for digital signatures, which in turn would compromise a large number of critically important security protocols relied on by the public.

3.2.1 SHA-3 competition announcement

In response to the attack against SHA-1, the National Institute of Standards and Technology held a conference to assess the remaining strength of its approved hash functions [25]. At this conference, it was concluded that while the approved hash functions were still secure for the moment, it was necessary to choose a new hash function which would provide a long-term replacement for SHA-1. While there are a number of more recently developed hash functions which could be used to replace MD5 and SHA-1, they each have their strengths and weaknesses. Since it is valuable for the sake of standardization to have a single hash function which is known to be secure and widely accepted, and to determine which of these hash functions is superior to the others, NIST called for a competition to determine a new, standardized hash function.

In an announcement in the Federal Register, NIST wrote [27]:

> NIST has decided that it is prudent to develop a new hash algorithm to augment and revise FIPS 180-2 [the SHA-2 family of hashes]. The new hash algorithm will be referred to as "SHA-3", and will be developed through a public competition, much like the development of the Advanced Encryption Standard (AES).

The formal announcement listed a set of submission criteria for candidate hash functions which included the following [27]:

1. The hash function should be publicly disclosed and free of royalties and intellectual property encumbrances.

2. The hash function should be implementable on a wide range of hardware and software platforms.

3. The hash function should be able to produce digest sizes of 224, 256, 384, and 512 bits with a maximum message length of $2^{64} - 1$ bits.

Each accepted hash function was then to be evaluated on the following criteria [27]:

Security It should be resistant to specific known attacks (e.g., differential cryptanalysis) and resistant to solutions for the security problems mentioned in section 2.2.1. Additionally, it should have special security properties specific to certain applications such as HMACs and PRNGs.

Cost It should be computationally efficient and should consume a minimal amount of memory.

Algorithm and Implementation Characteristics It should be flexible and simple.

The deadline for submitting hash functions was October 31, 2008. Following the submission deadline, NIST reviewed the candidates and filtered out those which did not meet the minimum submission requirements. The competition then proceeds in two rounds, in which the pool of candidate hash functions will be reduced to the final algorithm(s) which will be used in the SHA-3 standard.

3.2.2 Round One

On November 1, NIST began reviewing the submissions and selecting those that met the minimum requirements for inclusion in round one of the competition. Sixty-four hash functions were submitted to NIST before the deadline. After internal review, NIST found 51 of the 64 submissions to meet the minimum submission requirements, and these submissions became the first round candidates [3].

In February of 2009, NIST held the First SHA-3 Candidate Conference at Katholieke Universiteit Leuven in Belgium. The purpose of this conference was to formally announce the first round candidates and provide the public opportunities to question the hash function creators regarding their submissions [27]. This marked the beginning of the competition's first round, which will last for approximately twelve months after which there will be a second conference announcing those candidates which survived to the second round.

The candidates that made it into the first round are listed in Table 3.1 [2]. Those which at the time of this writing have been conceded broken and withdrawn are marked as such.

Table 3.1: SHA-3 Round One candidates

Hash Name	**Principal Submitter**
Abacus (conceded broken)	Neil Sholer
ARIRANG	Jongin Lim
AURORA	Masahiro Fujita (Sony)
BLAKE	Jean-Philippe Aumasson
Blender	Colin Bradbury
Blue Midnight Wish	Svein Johan Knapskog
BOOLE (conceded broken)	Greg Rose
Cheetah	Dmitry Khovratovich
CHI	Phillip Hawkes
CRUNCH	Jacques Patarin
CubeHash	D. J. Bernstein
DCH (conceded broken)	David A. Wilson
Dynamic SHA	Xu Zijie
Dynamic SHA2	Xu Zijie
ECHO	Henri Gilbert
ECOH	Daniel R. L. Brown
EDON-R	Danilo Gligoroski
EnRUPT	Sean O'Neil
ESSENCE	Jason Worth Martin
FSB	Matthieu Finiasz
Fugue	Charanjit S. Jutla
Grøstl	Lars Ramkilde Knudsen
Continued on the next page ...	

Table 3.1: (continued)

Hamsi	Özgül Küçük
JH	Hongjun Wu
Keccak	Joan Daemen
Khichidi-1 (conceded broken)	M. Vidyasagar
LANE	Sebastiann Indesteege
Lesamnta	Hirotaka Yoshida
Luffa	Dai Watanabe
LUX	Ivica Nikolić
MCSSHA-3	Mikhail Maslennikov
MD6	Ronald L. Rivest
MeshHash (conceded broken)	Björn Fay
NaSHA	Smile Markovski
SANDstorm	Rich Schroeppel
Sarmal	Kerem Varıcı
Sgàil	Peter Maxwell
Shabal	Jean-François Misarsky
SHAMATA (conceded broken)	Orhun Kara
SHAvite-3	Orr Dunkelman
SIMD	Gaëtan Leurent
Skein	Bruce Schneier
Spectral Hash	Çetin Kaya Koç
StreamHash (conceded broken)	Michal Trojnara
SWIFFTX	Daniele Micciancio
Tangle (conceded broken)	Rafael Alvarez
TIB3	Daniel Penazzi
Twister	Michael Gorski
Vortex	Michael Kounavis
WaMM (conceded broken)	John Washburn
Waterfall (conceded broken)	Bob Hattersley

Chapter 4

Cube Attacks

4.1 Overview

In parallel with these developments in the field of cryptographic hash functions, a new type of cryptanalytical attack, called a Cube Attack, was developed. In September 2008, Itai Dinur and Adi Shamir released a paper entitled "Cube Attacks on Tweakable Black Box Polynomials" [14] in which they described this new type of attack.

Cube attacks are particularly interesting in that they are a new form of generic attack; they can be applied to any cryptosystem that takes secret and public inputs to produce an attack that derives the secret input. Additionally, for the attack to be successful, no knowledge of the cryptosystem under attack is necessary; the cryptosystem can be treated like a "black box."

Finally, and perhaps most interestingly, cube attacks are not limited by the size of a cryptosystem's output bit polynomials, but instead are limited solely by the *degree* of those polynomials. Hence, a cryptosystem can have corresponding polynomials that contain arbitrarily many terms, but as long as the degree of those polynomials is sufficiently small, the cryptosystem is vulnerable to a cube attack.

In the following sections, we will provide a summary description of Dinur and Shamir's cube attacks. Please note that all credit for the ideas presented in this chapter goes to Dinur and Shamir. For a more detailed description, we refer the reader to [14].

4.2 Terminology

A cryptosystem to which a cube attack is to be applied is viewed as a series of binary polynomials over GF(2), one polynomial for each bit of output the cryptosystem produces. (From this point on, all polynomials referred to in this chapter are presumed to be binary polynomials over GF(2) unless otherwise specified.) Each polynomial takes the cryptosystem's secret and public inputs and produces a single bit output. The cryptosystem under attack must also take both a secret input and a public input, e.g., a cipher which takes a key and a plaintext.

Definition 4.1. Assume some polynomial $p(x_1, \ldots, x_n)$ and a set $I \subseteq \{1, \ldots, n\}$ of indices to the variables of p. Let t_I be a subterm of p which is the product of the variables indexed by I. Then factoring p by t_I, yields

$$p(x_1, \ldots, x_n) \equiv t_I \cdot p_{S(I)} + q(x_1, \ldots, x_n)$$

where we will call $p_{S(I)}$ the *superpoly* of I in p, and q is the linear combination of all terms which do not contain t_I.

Definition 4.2. A *maxterm* is a term t_I with a corresponding superpoly $p_{S(I)}$ such that $deg(p_{S(I)}) = 1$, i.e. the superpoly of I in p is a non-constant linear polynomial.

Definition 4.3. Any k-size subset I of indices to variables in a polynomial p defines a k-dimensional boolean *cube* with corners corresponding to all possible assignments of 0/1 to the variables in I. This cube can be represented by a set C_I of 2^k vectors corresponding to the cube's corners. For a vector $v \in C_I$, we define $p_{|v}$ to be a derivation of p in which the variables in I are set to the values in v.

We define p_I to be the sum of all such derived polynomials on the cube C_I, i.e. $p_I \triangleq \sum_{v \in C_I} p_{|v}$. We say that p_I is p summed over the cube C_I.

4.3 Cube Attack Theory

4.3.1 Structure

A cube attack presumes a cryptosystem which takes a secret and a public input. The attacker has no knowledge of the cryptosystem but is given access to two things:

1. A *simulator* which takes pairs of secret and public inputs and produces outputs.
2. An *oracle* which contains a hidden secret and when given a public input will produce an output.

What a cube attack allows the attacker to do is to derive the oracle's secret given these starting parameters.

4.3.2 The primary observation

The primary observation on which cube attacks are built is deceptively simple:

Theorem 4.1. *For any polynomial p and subset of variables I, $p_I \equiv p_{S(I)} \mod 2$.*

(See [14, §3] for a proof of this theorem.)

Stated another way, the value of any superpoly can be computed by computing the sum of the values of the entire polynomial for each vector in the superpoly's cube.

Let's consider a small example. Let p be a polynomial with public inputs v_1, v_2 and secret inputs x_1, x_2, x_3.

$$\begin{aligned} p(v_1, v_2, x_1, x_2, x_3) &= v_1v_2x_1 + v_1v_2x_2 + v_2x_2x_3 + v_1v_2 + v_2 + x_1x_3 + x_3 + 1 \\ &= v_1v_2(x_1 + x_2 + 1) + (v_2x_2x_3 + x_1x_3 + v_2 + x_3 + 1) \end{aligned}$$

Now, consider a set I which defines a cube C_I and its corresponding subterm and superpoly.

$$I = \{1, 2\}$$
$$t_I = v_1v_2$$
$$p_{S(I)} = x_1 + x_2 + 1$$

Summing p over the cube C_I produces the value of the superpoly $p_{S(I)}$.

$$\begin{aligned} p_I &= 0 \cdot 0(x_1 + x_2 + 1) + (0 \cdot x_2 \cdot x_3 + x_1 \cdot x_3 + 0 + x_3 + 1) \\ &\quad + 0 \cdot 1(x_1 + x_2 + 1) + (1 \cdot x_2 \cdot x_3 + x_1 \cdot x_3 + 1 + x_3 + 1) \\ &\quad + 1 \cdot 0(x_1 + x_2 + 1) + (0 \cdot x_2 \cdot x_3 + x_1 \cdot x_3 + 0 + x_3 + 1) \\ &\quad + 1 \cdot 1(x_1 + x_2 + 1) + (1 \cdot x_2 \cdot x_3 + x_1 \cdot x_3 + 1 + x_3 + 1) \\ &= x_1 + x_2 + 1 \end{aligned}$$

4.3.3 Secondary observations

From this we can derive several other observations:

- The value of a superpoly $p_{S(I)}$ on the secret input bits is the *value* of p_I. No knowledge of the internal algebraic structure of p is necessary.
- The superpoly $p_{S(I)}$ of any maxterm t_I is linear. (This follows from the definition of a maxterm.)
- If we can find maxterms on the public input bits, and set all other public input bits to zero, if their corresponding superpolys exist and are non-constant, then we have a set of linear polynomials on the secret input bits which can be used to form a system of linear equations.
- For a polynomial p of degree d, the degree of any maxterm t_I is at most $d - 1$.
- Computing p over some cube C_I requires $2^{|I|}$ evaluations of p. Since $|I| \leq d - 1$ for any maxterm t_I, then the complexity of computing a superpoly value is $\mathcal{O}(2^{d-1})$ evaluations of p.
- Presume p has n secret input bits and m public input bits. Since computing p over some cube is of complexity $\mathcal{O}(2^{d-1})$, we could brute force guess b of the n secret bits for some $b \leq d - 1$ without increasing this complexity. Doing so could make a cube attack easier since there would now be only $n - b$ secret bits it must recover.

4.3.4 Cube attacks in a nutshell

This finally leads us to a general description of a cube attack.

1. Find maxterms on the public input bits with linearly independent superpolys on the secret input bits by simulating the victim function with crafted public and secret inputs.

2. For each superpoly on the secret inputs, query the oracle for each vector in the maxterm's cube. Combine the superpolys and their values to form a system of linear equations.

3. Solve the system of linear equations for the secret. (This can be combined with brute forcing some of the secret bits if the superpolys don't cover all of them.)

4.4 Precomputation

Having now described the theory and general form of a cube attack, we turn our attention to specifics. A cube attack can be split into two phases: precomputation and an online attack. Precomputation is the part of the attack which can be done independent of the oracle. The online attack is the phase in which the results from precomputation are used to query the oracle and derive the secret.

In the precomputation phase, the objective is to find maxterms that correspond to superpolys on the secret bits and to derive the algebraic structure of those superpolys. Once enough linearly independent superpolys have been found, they can stored and used in all subsequent online attacks on this cryptosystem. (In other words, precomputation is performed once for a cryptosystem and then used to query arbitrarily many oracles.)

4.4.1 Deriving superpolys

We know from Definition 4.2 that there is a one-to-one correspondence between a maxterm and a superpoly. Since a cube attack presumes no starting knowledge of the cryptosystem's algebraic structure, we must derive the algebraic structure of a superpoly from its maxterm.

Given a superpoly $p_{S(I)}$, we know it is composed of single variable terms of the form x_j for some secret bit index j and a constant term (either 0 or 1 since we are operating in GF(2)). So, a superpoly's algebraic structure can be derived by simply finding the coefficients for all x_j and the value of its constant term. These values can be determined as follows:

Constant term: Compute the value of p_I with all inputs set to zero except for the variables in I.

Coefficient of x_j: Compute the value of p_I with all inputs set to zero except for the variables in I and x_j which is set to one. The superpoly contains x_j if and only if the result differs from the result for the constant term.

Since knowing a maxterm t_I means knowing its set of public input bit indices I, this method allows us to determine the algebraic structure of a superpoly $p_{S(I)}$ solely through knowledge of its maxterm t_I and complete evaluations of the polynomial p.

4.4.2 Finding maxterms

Since a maxterm's superpoly is linear, we know that for some maxterm t in polynomial p of degree d there is no term q in p such that $t|q$ and $deg(q) > deg(t) + 1$. Therefore, for dense polynomials (such as those found in most cryptosystems), it is highly likely that $deg(t)$ will be very close to $d - 1$.

To find maxterms then, our best bet is to search for them near the degree d of the polynomial. While there are no doubt many ways to do this, one way is a random walk such as the one described below.

1. Presume a polynomial p with n secret input bits, m public input bits, and an unknown degree of d.
2. Choose some $1 \leq k \leq m$ and a subset I of k public variables.
3. Compute the value of p_I, holding all public variables not in I constant (e.g., setting them to zero).
    - If I is too big, then $p_{S(I)}$ will be constant.
        - Test by computing p_I for several different secret inputs and checking that the result is always the same.
        - If it is, then drop one variable from I.
    - If I is too small, then $p_{S(I)}$ will be nonlinear.

- Test using a nonlinearity test, e.g., by choosing secret inputs x and y and verifying that $p_I(0) + p_I(x) + p_I(y) = p_I(x + y)$ holds a sufficient number of times, as in [14].
- If it's nonlinear, then add a new public variable to I.

4. If $p_{S(I)}$ is linear at the borderline between a constant and nonlinear result, then we've found a maxterm and $d \geq k + 1$. Record I.

5. Otherwise, go back to Step 2 and try with a new I.

6. Repeat from Step 2 above (optionally starting with $k = d - 1$ for our current best guess of d) until we have a set of maxterms which have $n - b$ corresponding superpolys that are linearly independent, where b is the number of secret bits we plan to obtain by brute force guessing.

4.5 Online Attack

Once enough linearly independent superpolys have been found, an online attack can be initiated. The structure of the online attack is comparatively quite simple:

1. For each linear polynomial $p_{S(I)}$, find its value using Theorem 4.1, i.e. querying the oracle for every assignment of public input bits in its cube (leaving all other public input bits held constant) and summing the results.

2. Combine the values with the superpolys to form a system of linear equations on the secret input bits hidden in the oracle.

3. If we have n superpoly equations, then solve the system to reveal the values of the secret input bits.

   If we chose to brute force guess b of the n secret bits, then we only have $n - b$ superpoly equations. Fix those $n - b$ superpoly equations and for each $v \in \{0, 1\}^b$ add b equations setting the remaining b secret bits to v and solve the system. Solve all 2^b such systems of linear equations and test their resulting secrets in order to find the values of the secret input bits.

4.5.1 Attacking multiple output bits

In describing this attack, we have not made much mention of the cryptosystem's output, especially the number of bits in that output. We have noted that each output bit of a cryptosystem can be defined as a polynomial on all the cryptosystem's inputs (both secret and public). Most cryptosystems have multiple output bits, and this means there are multiple output bit polynomials to attack.

A cube attack can work successfully against a single output bit, but it is frequently more efficient to target multiple output bits. When targeting multiple output bits, one simply tests a potential maxterm on all targeted output bits in parallel and associates any superpolys found with the output bit they were found on. Therefore, a single maxterm can be associated with multiple superpolys from different output bits, though there is still a one-to-one mapping between a maxterm and superpoly for a given output bit. When a superpoly's value is computed, only the output bit that superpoly is associated with is considered. In this way, a cube attack can target multiple output bits of a cryptosystem in parallel.

4.6 Complexity

As previously noted the complexity of a cube attack is limited by the degree of the output bit polynomials of the cryptosystem under attack. Since complexity of an attack generally refers to the online phase of the attack, we will consider the online phase first.

Consider a polynomial p of degree d with n secret input bits and m public input bits. The complexity of computing the values of n superpolys is at most $2^{d-1}n$ queries to the oracle. The complexity of solving the system of linear equations is $\mathcal{O}(n^2)$ [1].

So the total complexity of the online attack is $\mathcal{O}(2^{d-1}n) + \mathcal{O}(n^2)$.

As for the preprocessing phase, it is composed of a number of cube evaluations, each of which take $\mathcal{O}(2^{d-1})$ simulations of the cryptosystem. The exact number of cube evaluations will depend on the method used to find maxterms and how many maxterm guesses will fail, which in turn depends on the structure of the cryptosystem.

[1] Provided that the matrix of superpoly linear equations is nonsingular, which can be enforced by simply finding a few more superpolys if necessary.

It should be noted that the value of d is generally not known prior to the attack, so while the upper bound exists, the attacker does not necessarily know what it is. Methods for estimating d are covered in Chapter 5.

The memory complexity of both the preprocessing and online attacks is negligible.

4.7 A Short Example

Let us now consider a very small example of a complete cube attack. Consider a function $p(v_0, v_1, v_2, x_0, x_1, x_2)$ where v_0, v_1, v_2 are the public inputs and x_0, x_1, x_2 are the secret inputs. In order to aid understanding, we provide the definition of p:

$$\begin{aligned} p(v_0, v_1, v_2, x_0, x_1, x_2) = v_0v_1x_0 + v_0v_1x_1 + v_2x_0x_2 + v_1x_2 + v_0x_0 + v_0v_1 \\ + x_0x_2 + v_1 + x_2 + 1 \end{aligned}$$

However, it should be noted that this definition is considered unknown within the attack. It is provided solely for the reader's enlightenment.

4.7.1 Example precomputation

We will begin our search for maxterms with the subterm $t_{I_0} = v_2$ defined by the cube indices $I_0 = \{2\}$. We then have

$$p_{I_0}(v_0, v_1, x_0, x_1, x_2) = p(v_0, v_1, 0, x_0, x_1, x_2) + p(v_1, v_2, 1, x_0, x_1, x_2)$$

However, when using some p_I we will always fix the public inputs not in the set of cube indices I to some constant (e.g. zero). We mention this as a technicality, and for the rest of this example will simply presume that for some p_I the public inputs not in the set of cube indices I are set to zero, i.e.

$$p_{I_0}(x_0, x_1, x_2) = p(0, 0, 0, x_0, x_1, x_2) + p(0, 0, 1, x_0, x_1, x_2)$$

Next, we test to see if p_{I_0} is constant.

$$\begin{aligned} p_{I_0}(0, 0, 0) &= 0 \\ p_{I_0}(1, 0, 1) &= 1 \end{aligned}$$

Since different inputs produce different results p_{I_0} cannot be constant. So, we run a linearity test, choosing some $x = (1, 0, 1)$ and $y = (0, 1, 1)$, with $x \oplus y = (1, 1, 0)$.

$$\begin{aligned} p_{I_0}(0) + p_{I_0}(x) + p_{I_0}(y) &= p_{I_0}(x \oplus y) \\ p_{I_0}(0,0,0) + p_{I_0}(1,0,1) + p_{I_0}(0,1,1) &= p_{I_0}(1,1,0) \\ 0 + 1 + 0 &= 0 \end{aligned}$$

So the linearity test fails, too. p_{I_0} must be nonlinear. If we cheat for a moment and peek at the definition of p again, we see this is true:

$$\begin{aligned} p(v_0, v_1, v_2, x_0, x_1, x_2) &= v_0v_1x_0 + v_0v_1x_1 + v_2x_0x_2 + v_1x_2 + v_0x_0 + v_0v_1 \\ &\quad + x_0x_2 + v_1 + x_2 + 1 \\ &= t_{I_0} \cdot p_{S(I_0)} + q_0(v_0, \ldots, x_2) \\ &= v_2(x_0x_2) + (v_0v_1x_0 + v_0v_1x_1 + v_1x_2 + v_0x_0 + v_0v_1 \\ &\quad + x_0x_2 + v_1 + x_2 + 1) \end{aligned}$$

Next, we try the subterm $t_{I_1} = v_0v_2$ defined by cube indices $I_1 = \{0, 2\}$. We then have

$$\begin{aligned} p_{I_1}(x_0, x_1, x_2) &= p(0,0,0,x_0,x_1,x_2) + p(0,0,1,x_0,x_1,x_2) \\ &\quad + p(1,0,0,x_0,x_1,x_2) + p(1,0,1,x_0,x_1,x_2) \end{aligned}$$

Testing p_{I_1} shows it to be constant:

$$\begin{aligned} p_{I_1}(0,0,0) &= 0 \\ p_{I_1}(0,0,1) &= 0 \\ p_{I_1}(0,1,0) &= 0 \\ p_{I_1}(0,1,1) &= 0 \\ p_{I_1}(1,0,0) &= 0 \\ p_{I_1}(1,0,1) &= 0 \\ p_{I_1}(1,1,0) &= 0 \\ p_{I_1}(1,1,1) &= 0 \end{aligned}$$

This indicates that the subterm t_{I_1} does not exist in p. If we cheat again for a moment and peek at the definition of p, we see this is true; no term in p contains v_0v_2.

Next we try subterm $t_{I_2} = v_0 v_1$ with $I_2 = \{0, 1\}$. It proves to be non-constant

$$p_{I_2}(0, 0, 0) = 1$$
$$p_{I_2}(1, 0, 1) = 0$$

... and linear ...

$$p_{I_2}(0) + p_{I_2}(x) + p_{I_2}(y) = p_{I_2}(x \oplus y)$$
$$p_{I_2}(0, 0, 0) + p_{I_2}(1, 0, 1) + p_{I_2}(0, 1, 1) = p_{I_2}(1, 1, 0)$$
$$1 + 0 + 0 = 1$$

Normally, one would perform several linearity tests to make sure that t_{I_2} really was a maxterm, but for the sake of brevity, we will just show this one.

Now, we must deduce the superpoly $p_{S(I_2)}$ that corresponds to t_{I_2}. First we compute the free term:

$$p_{I_2}(0, 0, 0) = 1$$

Next we test for the presence of each secret variable:

$$p_{I_2}(1, 0, 0) = 0$$
$$p_{I_2}(0, 1, 0) = 0$$
$$p_{I_2}(0, 0, 1) = 1$$

Since the value differs from the free term for variables x_0 and x_1, we now know $p_{S(I_2)}$ to be:

$$p_{S(I_2)} = 1 + x_0 + x_1$$

Hence, we now have the maxterm $v_0 v_1$ defined by cube indices $\{0, 1\}$ with corresponding superpoly $1 + x_0 + x_1$. If we cheat again for a moment and peek at the definition of p, we can see this is true:

$$\begin{aligned} p(v_0, v_1, v_2, x_0, x_1, x_2) &= v_0 v_1 x_0 + v_0 v_1 x_1 + v_2 x_0 x_2 + v_1 x_2 + v_0 x_0 + v_0 v_1 \\ &\quad + x_0 x_2 + v_1 + x_2 + 1 \\ &= t_{I_2} \cdot p_{S(I_2)} + q_2(v_0, \ldots, x_2) \\ &= v_0 v_1 (1 + x_0 + x_1) \\ &\quad + (v_2 x_0 x_2 + v_1 x_2 + v_0 x_0 + x_0 x_2 + v_1 + x_2 + 1) \end{aligned}$$

Next we try subterm $t_{I_3} = v_0$ with $I_3 = \{0\}$. It proves to be non-constant

$$\begin{aligned} p_{I_3}(0,0,0) &= 0 \\ p_{I_3}(0,0,1) &= 1 \end{aligned}$$

... and linear ...

$$\begin{aligned} p_{I_3}(0) + p_{I_3}(x) + p_{I_3}(y) &= p_{I_3}(x \oplus y) \\ p_{I_3}(0,0,0) + p_{I_3}(1,0,1) + p_{I_3}(0,1,1) &= p_{I_3}(1,1,0) \\ 0 + 1 + 0 &= 1 \end{aligned}$$

We find the superpoly free term $p_{I_3}(0,0,0) = 0$ and coefficients:

$$\begin{aligned} p_{I_3}(1,0,0) &= 1 \\ p_{I_3}(0,1,0) &= 0 \\ p_{I_3}(0,0,1) &= 0 \end{aligned}$$

Which gives us $p_{S(I_3)} = x_0$. If we cheat again for a moment and peek at the definition of p, we can see this is true:

$$\begin{aligned} p(v_0, v_1, v_2, x_0, x_1, x_2) &= v_0v_1x_0 + v_0v_1x_1 + v_2x_0x_2 + v_1x_2 + v_0x_0 + v_0v_1 \\ &\quad + x_0x_2 + v_1 + x_2 + 1 \\ &= t_{I_3} \cdot p_{S(I_3)} + q_3(v_0, \ldots, x_3) \\ &= v_0(v_1x_0 + v_1x_1 + x_0 + v_1) \\ &\quad + (v_2x_0x_2 + v_1x_2 + x_0x_2 + v_1 + x_2 + 1) \end{aligned}$$

At first this might seem confusing: Shouldn't $p_{S(I_3)} = v_1x_0 + v_1x_1 + x_0 + v_1$? Well, technically it is. However, when we sum over the cube defined by I_3, we set all the public variables not in I_3 (i.e. v_1 and v_2) to zero. Doing so results in

$$\begin{aligned} p_{S(I_3)} &= v_1x_0 + v_1x_1 + x_0 + v_1 \\ &\Rightarrow 0 \cdot x_0 + 0 \cdot x_1 + x_0 + 0 \\ &= x_0 \end{aligned}$$

Finally we try subterm $t_{I_4} = v_1$ with $I_4 = \{1\}$. It proves to be non-constant

$$\begin{aligned} p_{I_4}(0,0,0) &= 1 \\ p_{I_4}(0,0,1) &= 0 \end{aligned}$$

... and linear ...

$$\begin{aligned} p_{I_4}(0) + p_{I_4}(x) + p_{I_4}(y) &= p_{I_4}(x \oplus y) \\ p_{I_4}(0,0,0) + p_{I_4}(1,0,1) + p_{I_4}(0,1,1) &= p_{I_4}(1,1,0) \\ 0 + 0 + 0 &= 0 \end{aligned}$$

We find the superpoly free term $p_{I_4}(0,0,0) = 1$ and coefficients:

$$\begin{aligned} p_{I_4}(1,0,0) &= 1 \\ p_{I_4}(0,1,0) &= 1 \\ p_{I_4}(0,0,1) &= 0 \end{aligned}$$

Which gives us $p_{S(I_4)} = 1 + x_2$. Peeking at the definition of p one last time shows this is correct:

$$\begin{aligned} p(v_0, v_1, v_2, x_0, x_1, x_2) &= v_0v_1x_0 + v_0v_1x_1 + v_2x_0x_2 + v_1x_2 + v_0x_0 + v_0v_1 \\ &\quad + x_0x_2 + v_1 + x_2 + 1 \\ &= t_{I_4} \cdot p_{S(I_4)} + q_4(v_0, \ldots, x_4) \\ &= v_1(v_0x_0 + v_0x_1 + x_2 + v_0 + 1) \\ &\quad + (v_2x_0x_2 + v_0x_0 + x_0x_2 + x_2 + 1) \end{aligned}$$

Once again, we must remember that when summing over the cube defined by I_4 all public inputs not in I_4 (i.e. v_0 and v_2) are set to zero. This results in

$$\begin{aligned} p_{S(I_4)} &= v_0x_0 + v_0x_1 + x_2 + v_0 + 1 \\ &\Rightarrow 0 \cdot x_0 + 0 \cdot x_1 + x_2 + 0 + 1 \\ &= 1 + x_2 \end{aligned}$$

At this point, we have found three cubes with linearly independent superpolys, as shown in Table 4.1. We may now proceed with the online attack.

Table 4.1: Maxterms (given as cube indices) and superpolys.

Superpoly polynomial $p_{S(I)}$	Cube indices I
x_0	$\{0\}$
$1 + x_0 + x_1$	$\{0, 1\}$
$1 + x_2$	$\{1\}$

4.7.2 Example online attack

Let there be some oracle $g(v_0, v_1, v_2)$ implemented as $p(v_0, v_1, v_2, x_0, x_1, x_2)$ with a hidden secret defined by $x_0 = 1, x_1 = 0, x_2 = 1$. We will use our precomputed cube attack on p to discover this secret by querying the oracle g.

To begin our attack, we must find the values of each superpoly. Recall from Theorem 4.1 that $p_{S(I)} = p_I \mod 2$. This applies to the oracle g, so for one of our cube index sets I we can find the value of its superpoly $g_{S(I)}$ in the oracle by computing g_I.

$$\begin{aligned} x_0 = q_{\{0\}} = g(0,0,0) + g(1,0,0) &= 1 \\ 1 + x_0 + x_1 = q_{\{0,1\}} = g(0,0,0) + g(1,0,0) + g(0,1,0) + g(1,1,0) &= 0 \\ 1 + x_2 = q_{\{1\}} = g(0,0,0) + g(0,1,0) &= 0 \end{aligned}$$

This gives us a system of linear equations which we solve to find

$$\begin{aligned} x_0 &= 1 \\ x_1 &= 0 \\ x_2 &= 1 \end{aligned}$$

This matches the values being hidden by the oracle g. Therefore, the cube attack has completed successfully.

Chapter 5

Estimating the Effectiveness of a Cube Attack

In [14] cube attacks were presented as a generic attack against a black box cryptosystem. There is incredible power and utility in a generic attack that can be used for black box cryptanalysis. However, there is also benefit in being able to analyze a cryptosystem's design in order to determine the effectiveness of an attack. What we will attempt to do in this chapter is establish a framework for analyzing a cryptosystem to determine its potential weakness to a cube attack. We will also look into some tactics that can aide in applying a cube attack.

5.1 Degree Analysis

At the heart of cube attacks is the concept of polynomial degree. Therefore, the best place to begin is by looking at what kinds of operations frequently used in cryptosystems affect degree and how they do so.

5.1.1 Basic primitives

The most basic operations used in cryptosystems are bitwise and mathematical operations. So, we will begin by analyzing how these operations translate into arithmetic in GF(2) on their bits.

One note on notation: In this chapter, we will refer to multi-bit integers by capital letters such as X and refer to the i^{th} bit of X as x_i using the

corresponding lower case letter. The degree of X, $\mathrm{d}(X)$, is considered to be the degree of the highest degree polynomial amongst its bits, i.e.

$$\mathrm{d}(X) = \max_{x \in X}(\mathrm{d}(x))$$

Bitwise operations

The GF(2) polynomial form and degree change of bitwise operations are as follows:

Bitwise operation	GF(2) polynomial	Degree change
$X \wedge Y$	xy	$\mathrm{d}(X \wedge Y) = \mathrm{d}(X) + \mathrm{d}(Y)$
$X \vee Y$	$xy + x + y$	$\mathrm{d}(X \vee Y) = \mathrm{d}(X) + \mathrm{d}(Y)$
$X \oplus Y$	$x + y$	$\mathrm{d}(X \oplus Y) = \max(\mathrm{d}(X), \mathrm{d}(Y))$
$\neg X$	$1 + x$	$\mathrm{d}(\neg X) = \mathrm{d}(X)$
$X \ll n$	x [a]	$\mathrm{d}(X \ll n) \leq \mathrm{d}(X)$
$X \gg n$	x	$\mathrm{d}(X \gg n) \leq \mathrm{d}(X)$
$X \lll n$	x	$\mathrm{d}(X \lll n) = \mathrm{d}(X)$
$X \ggg n$	x	$\mathrm{d}(X \ggg n) = \mathrm{d}(X)$

[a] By this we mean that the underlying polynomials do not change. Shifts and rotates move the polynomials to new bits, but they do not change the underlying algebraic structure of the polynomials as they move them. The exception is when bits are lost in a shift of a fixed width integer; in these cases if the remaining bits' polynomials are of lesser degree than a polynomial of a bit shifted out, the degree of X decreases.

Addition and subtraction modulo 2^n for some $n \in \mathbb{Z}^+$

Addition and subtraction modulo 2^n remain fairly simple because the modulus operation is just a bitmask which does not really affect the underlying polynomials other than possibly setting high bits to 0.

The GF(2) representation of $X+Y$ is x_0+y_0 for bit 0 and $x_i+y_i+c_{i-1}$ for all bits $0 < i < n$ where $c_i = x_i y_i + c_{i-1}x_i + c_{i-1}y_i$ and $c_0 = x_0 y_0$. From these formulas two important characteristics of addition become apparent. First, addition is a bit mixing operation: It introduces bit variables to a polynomial that are not from the bit to which that polynomial corresponds. Secondly, the maximum degree of the output bits from an addition increases from low bits to high bits. This is due to the carry bit which depends on all previous bits in both X and Y. Hence, the degree of some output bit $0 \leq j < n$ is

$$\mathrm{d}((X+Y)_j) = \max\left(\max_{0\le k<j}\left(\mathrm{d}(x_k)+\mathrm{d}(y_k)+\sum_{k<i<j}\max(\mathrm{d}(x_i),\mathrm{d}(y_i))\right),\mathrm{d}(x_j),\mathrm{d}(y_j)\right)$$

To simplify things, in most cases a pretty good estimate would be

$$\mathrm{d}((X+Y)_j) \ge \sum_{0<i<j}\max(\mathrm{d}(x_i),\mathrm{d}(y_i))$$

Multiplication, division, exponentiation, and addition/subtraction modulo m where $m \neq 2^n$ for some $n \in \mathbb{Z}^+$

These cases are more complex. In general it is probably easiest to treat them as black boxes and run a small cube test against them separately to get an empirical idea of their degree complexity.

5.1.2 Cryptographic design primitives

There are also a number of cryptographic primitives which appear frequently enough that they deserve special attention.

Bit shuffling permutations

Bit shuffling permutations come in a variety of shapes and sizes, but they all pretty much do the same thing: Shuffle the bit polynomials around to different bits. An example would be the permutation step of a substitution permutation network. Bit shuffling, just like shifts and rotates, does not change the algebraic structure of the bit polynomials; it just moves them to new bit positions. As such, it has no effect on the polynomial degree.

S-Boxes

S-boxes appear in almost all block cipher and some hash function designs. When analyzing a cipher that uses an S-box, there are one of three choices: treat it as a block box function and run a cube attack against it to see what degree maxterms it will yield, compute its output bit polynomials, or simply compute the upper bound of its degree based on the number of input bits it takes. Since some S-boxes have a small input size, the last option

can occasionally be useful. The degree of an S-box can be expressed by the inequality

$$\mathrm{d}(S(X)) \leq |X| \cdot \max_{x \in X} (\mathrm{d}(x))$$

So for an S-box with 8 inputs and a set of inputs whose highest degree is 4, the degree of the S-box output would be at most 32.

With an S-box that has a small number of inputs, it may frequently be more useful to just compute its output bit polynomials. This is useful not only to determine the degree of the S-box outputs but also to look for configurations where low degree maxterms might be created.

Finally if all else fails, just running a cube attack against the S-box and using the empirical results in the larger estimation can suffice. This may be the only option for S-boxes with a significant number of inputs.

Rounds

For the sake of this discussion, we will define a round function as a function whose output is fed back into itself as its input a fixed number of times. In most cases, the round function operates on all its input and tries to reflect that in its output. In these cases, given a round function R which is cycled through n rounds, the degree of n round applications is bounded by $(\mathrm{d}(R))^n$.

In some cases a "round" doesn't actually modify the degree of its entire output (e.g., a shift register of some type). In these cases, the output usually needs to be subdivided and analyzed separately. (An example of this will be seen later in our analysis of the ESSENCE compression function.)

5.1.3 Tying it all together

To estimate the bound on the degree of an entire cryptographic function, one simply follows its input starting with a degree of one for each output bit and changing the degree with each primitive encountered. If a construct is encountered that has an unknown, confusing, or excessively complex structure, one can run a cube attack against that construct in isolation, add one to the degree of the largest maxterm found in a working attack, and use that as the degree estimation for that construct. If one cannot even get a working cube attack against the construct, then it's unlikely that a cube attack against the cryptosystem will be successful.

5.2 Tactics and Gotchas

Having established a system of estimating the bound on the degree of a cryptosystem, one may become discouraged when discovering that the degree of a particular cryptosystem could be painfully high. This does not always mean a cube attack will fail (though it's frequently a good indicator). In any situation though, there are certain things to keep in mind.

5.2.1 Secret data flow

The purpose of a cube attack is to extract the secret, usually a key. As such, the secret is the only thing we care about. Examine and watch the data flow within a cryptosystem's design. There may be all kind of complicated things going on, but the only thing we care about is where the public input and secret input bits combine and are rearranged. As a result, one can often safely ignore various kinds of generation or expansion components and just look at how their results change the shape of the public and secret input bit interactions.

Furthermore, it is important to keep in mind what is really being attacked. If one is attacking a cryptsystem with a key, is it necessary to make the key bits the secret bits of the attack? Is the key expansion phase reversible? If so, then the secret bits could be the expanded key, and the the expansion could be reversed to recover the original key. This removes the degree of the key expansion from the degree of difficulty affecting the attack. Going a step further, is the key even necessary? If the application of the attack can work solely with the key expansion, then there is no need to increase the difficulty of the attack by recovering the key when the expansion is sufficient.

Through whatever ways available, one should cut away at the cryptosystem so as to run the cube attack as close as possible to the "heart" of the cryptosystem. The closer one gets to the center, the more likely the degree will be low and the easier the attack will be.

5.2.2 The elusive maxterm

Just because the degree of a cryptosystem is high, it does not mean that all the usable maxterms are of degree one less than that. Frequently, maxterms will be found of degree that are considerably lower than the degree of the cryptosystem. This is because in a cube attack all the public input bits that

are not in the maxterm are fixed, usually to zero. This tends to annihilate a lot of higher degree terms. The minimum degree of usable maxterms is usually less influenced by the degree of the cryptosystem than it is by the ratio of the number of secret bits to the number of public bits in the polynomial's terms and the extent to which the secret bits have diffused throughout the polynomial. The degree of the cryptosystem just establishes an upper bound near which we have a very high probability of finding maxterms.

It should also be noted though that if one notices the presence of a low degree term composed of a few public input bits and one secret bit, that does not indicate the existence of a maxterm. If there also exists another term in the same polynomial in which the same public and secret bits exist along with another secret bit, the corresponding superpoly for the subterm composed of those public bits would not be linear, and therefore the subterm would not be a maxterm.

5.2.3 Hybrid attacks

Finally, whenever possible it is best to combine cube attacks with other methods in order to cut high degree components out and bring the cube attack closer to the heart of the cryptosystem. If a component of the cryptosystem is reversible or easily attacked by some other method, have the cube attack begin or end just after or before that component, and then use some other method to pass through that component and connect with the cube attack.

Examples of components that can frequently be left out of a cube attack and be handled by some other means are key expansion, initialization phases, and finalization phases. By removing these components, the degree of the modified cryptosystem usually decreases, giving a cube attack a better chance of success.

These types of hybrid attacks could be particularly useful against cryptosystems that heavily rely on their initialization or finalization steps to raise their degree.

Chapter 6

Attacks on SHA-3 Candidates

Having laid the foundation of cube attacks, we now turn our attention to their application to candidates in the SHA-3 hash function competition. We found that very few hash functions were succeptible to a pure cube attack, but we did find successful reduced-round attacks against the ESSENCE and Keccak hash functions.

We will discuss these reduced-round attacks, as well as briefly mentioning several other hash functions we found resilient to pure cube attacks.

6.1 Attack on Reduced-Round ESSENCE

6.1.1 The ESSENCE hash function

ESSENCE is a candidate for the SHA-3 hash function competition run by NIST [21]. It is described by its authors as "a family of hash functions" because it has a number of configurable parameters. However, we will devote our attention only to the variant of ESSENCE defined by the configuration values set for the official submission of ESSENCE to NIST.

General Structure

We will not attempt to provide a complete specification of ESSENCE, but will instead focus on describing the primary characteristics of its structure as a hash function.

Two compression functions: ESSENCE employs two compression functions, one for processing 256 bit blocks and one for processing 512 bit blocks. Which function is used is determined by the size of the output hash requested.

Message partitioning: As a hash function, ESSENCE takes an arbitrarily long message. This message is internally split into "chunks" of 1048576 bytes (one megabyte) in size, except for the last chunk which may be smaller. Each of these chunks is further subdivided into 256 or 512 bit blocks depending on which compression function is being used.

Padding: If the message data for the last block of the last chunk is not a full block size in length, it is padded with zeros.

Chaining structure: The design of ESSENCE provides for parallelism through the use of Merkle hash trees. However, the tree depth is set to 0 in the configuration of the official variant of ESSENCE submitted to NIST. This produces a sequential design.

Each chunk is hashed separately using the compression function and a standard Merkle-Damgård chaining structure. The resulting hashes from each chunk are then taken to form the blocks of a new "message" and are also hashed together using a Merkle-Damgård chaining structure. The result is that the final ESSENCE hash is essentially a hash of hashes.

Finalization: When producing the final ESSENCE hash by hashing the chunk hashes, a final block is appended to the sequence of chunk hashes. This final block contains the various configuration values used by the variant of ESSENCE being employed as well as the number of chunks in the message in order to prevent length extension attacks. All this data is stored in the first 256 bits of the final block. If a 512 bit compression function is being used, then the last 256 bits of the final block are the hexadecimal expansion of the fractional part of π starting with the 64^{th} digit.

So the final ESSENCE hash is the hash of each of the chunk hashes and the final block.

Initialization vectors: The initialization vector used in the Merkle-Damgård chain of the final hash is set to the hexadecimal expansion of the fractional part of π. The initialization vectors used for individual blocks within a chunk

are composed of configuration information for the ESSENCE variant in use as well as the sequence number of the chunk being processed. This data is 256 bits long; if the compression function is 512 bits then the last 256 bits of the chunk IV are the same as the last 256 bits of the IV used for the final hash (i.e. values from the fractional part of π).

The Compression Function

As previously noted, ESSENCE has two different compression functions, one for 256 bit outputs and one for 512 bits outputs. Both compression functions are Davies-Meyer constructions based on keyed permutations, respectively called E_{256} and E_{512} permutations. The only difference between these permutations is in the size of the registers used (respectively 32 or 64 bits) and the L permutation (respectively called L_{32} and L_{64}). When the discussion applies to both permutations, we will simply refer to them as E.

The E permutation takes a key k and a sequence of eight registers $r0, \ldots, r7$ and produces a new sequence of eight registers. It is a feedback shift register which is defined as follows [22]:

$$\begin{aligned} &E(k, \{r0, r1, r2, r3, r4, r5, r6, r7\}) = \\ &\{F(r6, r5, r4, r3, r2, r1, r0) \oplus r7 \oplus L(r0) \oplus k, r0, r1, r2, r3, r4, r5, r6\} \end{aligned} \tag{6.1}$$

The size of the registers $r0, \ldots, r7$ and the key k are 32 bits for E_{256} and 64 bits for E_{512}.

The feedback function F is a bitwise function with each bit of its output independently depending on the same bit in each input variable (i.e. just as in other bitwise operations like bitwise AND). F can then be defined for single bit input variables as a function in GF(2) defined by the following polynomial [22]:

$$
\begin{aligned}
F(a,b,c,d,e,f,g) \;=\; & abcdefg + abcdef + abcefg + acdefg + \\
& abceg + abdef + abdeg + abefg + \\
& acdef + acdfg + acefg + adefg + \\
& bcdfg + bdefg + cdefg + \\
& abcf + abcg + abdg + acdf + adef + \\
& adeg + adfg + bcde + bceg + bdeg + cdef + \\
& abc + abe + abf + abg + acg + adf + \\
& adg + aef + aeg + bcf + bcg + bde + \\
& bdf + beg + bfg + cde + cdf + def + \\
& deg + dfg + \\
& ad + ae + bc + bd + cd + \\
& ce + df + dg + ef + fg + \\
& a + b + c + f + 1
\end{aligned}
\tag{6.2}
$$

Finally, the linear function L is a linear feedback shift register in the Galois configuration. For L_{64} the characteristic polynomial is

$$
\begin{aligned}
p_{64}(x) \;=\; & x^{64} + x^{63} + x^{61} + x^{60} + x^{55} + x^{53} + x^{50} + x^{49} + \\
& x^{46} + x^{44} + x^{41} + x^{40} + x^{36} + x^{33} + x^{32} + \\
& x^{31} + x^{30} + x^{29} + x^{26} + x^{25} + x^{23} + x^{20} + x^{18} + x^{17} + \\
& x^{14} + x^{13} + x^{11} + x^{8} + x^{7} + x^{4} + x^{2} + x + 1
\end{aligned}
\tag{6.3}
$$

For L_{32} the characteristic polynomial is

$$
\begin{aligned}
p_{32}(x) \;=\; & x^{32} + x^{31} + x^{24} + x^{22} + x^{19} + x^{17} + \\
& x^{13} + x^{12} + x^{11} + x^{9} + x^{8} + x^{5} + x^{4} + x^{2} + 1
\end{aligned}
\tag{6.4}
$$

In both cases, a call to $L_i(x)$ initializes the LFSR with x and shifts it i steps [22].

Having now defined the E permutation, we can define the ESSENCE compression function $G(R, K, n)$. As previously noted, there are two compression functions: They are G_{256} which uses E_{256} and G_{512} which uses E_{512}. Just as

with the E permutation, we will refer to both compression functions as G when the discussion applies to them both.

The compression function $G(R, K, n)$ is essentially two connected E permutations. It is parameterized by the sequence of integers R which initializes the registers in the primary permutation E_R, a sequence of integers K which initializes a sequence of key registers in the key scheduling permutation E_K, and the number of rounds n to step both permutations. The key scheduling permutation E_K varies slightly from an ordinary E permutation in that it does not have a key value being exclusive or'ed into it with each step, but instead its output provides the key value for each round of the E_R permutation.

The compression function $G(R, K, n)$ can be formally defined as:

Algorithm 1 Compression function $G(R, K, n)$

$K' \leftarrow K$
$R' \leftarrow R$
for $i = 1$ to n **do**
  $R' \leftarrow E(K'[7], R')$ {the E_R permutation}
  $K' \leftarrow E(0, K')$ {the E_K permutation}
end for
return $R \oplus R'$

ESSENCE employs the $G(R, K, n)$ compression function in Davies-Meyer mode with the number of rounds n fixed at 32, i.e.

$$H_0 = IV$$
$$H_i = G(H_{i-1}, M_{i-1}, 32)$$

where H_i is the new hash formed by the addition of a message block M_{i-1}.

6.1.2 Cube attack on a reduced-round ESSENCE compression function

We attacked reduced-round variants of both the 256-bit ESSENCE hash function and the ESSENCE $G_{256}(R, K, n)$ compression function. We will consider the compression function first.

Attack configuration

Since cube attacks are structured to recover some kind of secret, we used the compression function to make a very naive HMAC: The first 32 bits of the message block stored a secret value, and the remaining 224 bits stored a public value. We used only 32 bits for the secret so that it would fill exactly one register of the E permutation; this was done to ease analysis.[1] Just as in the ESSENCE hash function itself, we call the compression function with $R = IV$ and $K = M$, where IV is an initialization vector and M is the message block composed of the secret and public values. With this configuration we used cube attacks to attempt a partial pre-image attack against the ESSENCE compression function.

Attack results

As previously noted, the official ESSENCE configuration sets the number of rounds to 32, i.e. the compression function is $G(R, K, 32)$. We successfully found a complete cube attack that recovers the full 32-bit secret on up to 19 rounds, i.e. $G_{256}(IV, M, 19)$.

In Table 6.1 we give our precomputation of the attack. In each row we give a superpoly $p_{S(I)}$, the cube indices I that define the corresponding maxterm t_I, and the output bit the superpoly resides on. Note that because we attacked all output bits in parallel, a single maxterm may be associated with multiple superpolys on multiple output bits. In the superpoly polynomials, an x_j variable refers to bit j of the secret input. Likewise, a cube index i refers to bit i of the public input. Note that these bit indices are indices into the secret and public inputs, not indices into the hash input, so both secret and public bit indices start from 0. To convert this to hash input bit indices using our partial pre-image attack configuration, the secret bit indices would remain the same and the public bit indices would be increased by 32.

Finally, we should note that the maxterms given in this table (via their cube indices) are simply the first ones we found that had linearly independent superpolys; we did a random walk and every time we found a maxterm with a superpoly that was linearly independent to those we already had, we added it to our list. There are certainly more such maxterms, but only these were necessary to mount an attack.

[1] The attack can be extended to larger secret values, but it becomes marginally more difficult as the ratio of initial secret value registers to public value registers increases.

Table 6.1: Maxterms for 19 round ESSENCE compress: $G_{256}(IV, M, 19)$

Superpoly Polynomail	Cube Indices	Output Bit Index
$1 + x_1$	{4,6,21}	234
$1 + x_6$	{0,4,30}	225
x_{10}	{0,4,30}	234
$x_0 + x_6 + x_{10}$	{0,4,30}	253
$x_7 + x_{12}$	{5,7}	233
$x_6 + x_{14}$	{0,4,30}	238
x_{18}	{18,20,46}	253
$x_1 + x_2 + x_4 + x_6 + x_{14} + x_{16} + x_{18}$	{14,25}	242
$x_3 + x_{19}$	{11,13,19}	238
$1 + x_3 + x_{17} + x_{19}$	{11,13,19}	241
$x_{16} + x_{19} + x_{21}$	{19,20}	226
$1 + x_3 + x_{13} + x_{21}$	{11,13,19}	224
$1 + x_{19} + x_{21}$	{11,13,19}	228
$x_8 + x_{21}$	{19,20,59}	226
$1 + x_6 + x_{11} + x_{12} + x_{21}$	{0,11,26}	230
$1 + x_3 + x_5 + x_{15} + x_{17} + x_{23}$	{19,20,59}	235
$x_1 + x_4 + x_6 + x_{14} + x_{25}$	{14,25}	238
$1 + x_1 + x_5 + x_6 + x_{15} + x_{18} + x_{21} + x_{26}$	{19,20,59}	224
$1 + x_4 + x_9 + x_{10} + x_{12} + x_{13} + x_{16} + x_{19} + x_{26} + x_{27}$	{18,45}	233
$1 + x_{27}$	{4,6,21}	239
$1 + x_4 + x_{12} + x_{15} + x_{23} + x_{25} + x_{27}$	{18,20,46}	236
$1 + x_4 + x_8 + x_{17} + x_{19} + x_{23} + x_{24} + x_{27}$	{18,20,46}	240
$1 + x_{28}$	{11,13,19}	252
$1 + x_{16} + x_{19} + x_{21} + x_{24} + x_{29}$	{19,20}	250
$1 + x_3 + x_8 + x_{21} + x_{24} + x_{29}$	{19,20,59}	250
$x_2 + x_3 + x_5 + x_6 + x_7 + x_{10} + x_{11} + x_{14} + x_{16} + x_{18} + x_{26} + x_{27} + x_{30}$	{20,27,214}	235
$1 + x_2 + x_5 + x_6 + x_{15} + x_{22} + x_{25} + x_{31}$	{5,11,61}	225
$1 + x_2 + x_3 + x_5 + x_6 + x_7 + x_{10} + x_{11} + x_{14} + x_{18} + x_{20} + x_{26} + x_{27} + x_{31}$	{20,27,214}	230
$1 + x_0 + x_5 + x_8 + x_{10} + x_{14} + x_{17} + x_{18} + x_{26} + x_{31}$	{0,16,57}	248
$1 + x_{13} + x_{14} + x_{17} + x_{23} + x_{24} + x_{25} + x_{26} + x_{29} + x_{31}$	{0,16,57}	254
$x_4 + x_{15} + x_{17} + x_{18} + x_{22} + x_{23} + x_{24} + x_{25} + x_{27} + x_{31}$	{18,20,46}	248
$1 + x_{31}$	{18,20,46}	249

Since the largest cube size is 3 and there are 32 secret bits to solve for, the complexity of the attack is 2^8 compressions. In terms of real-world timing, a search for a set of maxterms like those listed above takes us about 15 seconds on a desktop PC, and the online attack is practically instantaneous.

We also found the following 7-cube on 20 rounds. (We will discuss why we stopped our full attack at 19 rounds in the analysis section.)

Table 6.2: Maxterm for 20 round ESSENCE compress $G_{256}(IV, M, 20)$

Superpoly Polynomial	Cube Indices	Output Bit Index
$1 + x_0 + x_1 + x_9 + x_{17} + x_{27} + x_{28} + x_{31}$	{0,1,20,27,28,29,64}	227

As was noted previously, we restricted the secret input size to 32 bits in order to ease analysis. The attack still works on larger secret sizes, albeit with slightly higher complexity. For example, we have successfully attacked 18 round ESSENCE compress using a configuration with 64-bit secret and 192-bit public and obtained results similar to the attack on 19 rounds detailed above.

6.1.3 Cube attack on reduced-round ESSENCE

We attacked a reduced-round variant of the complete 256 bit ESSENCE hash function. Due to its 0-depth hash tree construction, this is more difficult than attacking just the compression function as the message block effectively gets compressed twice. We used the naive HMAC setup as was applied to the compression function and achieved a complete cube attack on 9 rounds. Once again the goal was a partial pre-image attack to recover the 32 secret input bits. The maxterm cube indices and superpolys used in the attack are below:[2]

Table 6.3: Maxterms for 9 round ESSENCE

Superpoly Polynomial	Cube Indices	Output Bit Index
$1 + x_3$	{2,21}	225
x_4	{0,19}	235
$x_1 + x_3 + x_6$	{21,24}	225
x_9	{15,28}	229
$x_2 + x_9$	{2,11}	252
x_{11}	{8,22}	235
$x_2 + x_3 + x_9 + x_{14}$	{2,11}	238
$1 + x_4 + x_{18}$	{0,19}	238
$1 + x_6 + x_9 + x_{20}$	{9,20}	225
$1 + x_4 + x_{21}$	{0,19}	230
Continued on the next page ...		

[2]Once again more such maxterms and superpolys exist, but they are not necessary for a complete attack.

Table 6.3: (continued)

$x_1 + x_3 + x_9 + x_{12} + x_{23}$	{9,23}	235
$1 + x_6 + x_{24}$	{1,4}	231
$x_{13} + x_{24}$	{2,13}	254
$x_5 + x_{12} + x_{24}$	{5,26}	233
$1 + x_{28}$	{2,13}	252
$1 + x_1 + x_6 + x_{24} + x_{29}$	{1,4}	253
$x_5 + x_8 + x_9 + x_{11} + x_{15} + x_{19} + x_{22} + x_{24} + x_{25} + x_{26} + x_{27} + x_{29}$	{9,60}	235
$1 + x_4 + x_5 + x_6 + x_{10} + x_{11} + x_{13} + x_{15} + x_{16} + x_{18} + x_{19} + x_{23} + x_{24} + x_{27} + x_{29} + x_{30}$	{19,44}	252
$1 + x_0 + x_2 + x_9 + x_{10} + x_{11} + x_{12} + x_{14} + x_{15} + x_{16} + x_{17} + x_{20} + x_{21} + x_{22} + x_{26} + x_{27} + x_{28} + x_{30}$	{22,45}	254
x_{30}	{8,22}	254
$1 + x_3 + x_{25} + x_{30}$	{3,9}	254
$1 + x_0 + x_2 + x_3 + x_6 + x_7 + x_9 + x_{10} + x_{11} + x_{17} + x_{20} + x_{21} + x_{22} + x_{26} + x_{30}$	{0,19}	225
$x_0 + x_1 + x_2 + x_3 + x_4 + x_6 + x_7 + x_9 + x_{10} + x_{11} + x_{17} + x_{18} + x_{20} + x_{21} + x_{22} + x_{26} + x_{30}$	{0,19}	248
$x_0 + x_1 + x_2 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10} + x_{12} + x_{15} + x_{18} + x_{22} + x_{23} + x_{24} + x_{25} + x_{26} + x_{27} + x_{28} + x_{30}$	{0,18}	235
$1 + x_0 + x_1 + x_2 + x_5 + x_{12} + x_{14} + x_{15} + x_{16} + x_{18} + x_{20} + x_{24} + x_{25} + x_{29} + x_{30}$	{21,31}	235
$1 + x_0 + x_3 + x_4 + x_8 + x_{10} + x_{11} + x_{12} + x_{14} + x_{15} + x_{17} + x_{20} + x_{21} + x_{22} + x_{28} + x_{29} + x_{30}$	{34,35}	254
$x_2 + x_3 + x_4 + x_6 + x_7 + x_9 + x_{11} + x_{15} + x_{16} + x_{18} + x_{19} + x_{21} + x_{22} + x_{23} + x_{26} + x_{29} + x_{30} + x_{31}$	{1,4}	252
$x_0 + x_2 + x_3 + x_5 + x_6 + x_{14} + x_{15} + x_{16} + x_{17} + x_{18} + x_{20} + x_{24} + x_{25} + x_{29} + x_{30} + x_{31}$	{1,4}	254
$1 + x_0 + x_1 + x_2 + x_3 + x_5 + x_{12} + x_{14} + x_{15} + x_{16} + x_{17} + x_{18} + x_{20} + x_{25} + x_{29} + x_{30} + x_{31}$	{5,24}	252
$1 + x_0 + x_4 + x_6 + x_7 + x_{13} + x_{15} + x_{17} + x_{18} + x_{19} + x_{20} + x_{22} + x_{23} + x_{24} + x_{30} + x_{31}$	{24,60}	238
$1 + x_6 + x_9 + x_{11} + x_{12} + x_{13} + x_{14} + x_{15} + x_{16} + x_{17} + x_{18} + x_{19} + x_{22} + x_{25} + x_{26} + x_{27} + x_{28} + x_{31}$	{4,7}	225
$1 + x_0 + x_1 + x_8 + x_9 + x_{11} + x_{14} + x_{16} + x_{17} + x_{18} + x_{19} + x_{20} + x_{23} + x_{24} + x_{28} + x_{29} + x_{30} + x_{31}$	{20,60}	235

Since all the maxterms are of degree 2 and there are 32 secret bits to recover, the complexity of the attack is 2^7 hash computations.

6.1.4 Analysis

At this point one may wonder why we are presenting attacks of such low complexity. It seems that if attacks of such low complexity could be found then more difficult yet still feasible attacks should exist by extension to higher rounds. It turns out that the ESSENCE compression function has some very interesting algebraic peculiarities which make it difficult to predict the existence and degree of maxterms.

Analysis of the compression function

If one examines the F function in Equation 6.2 that is used in the ESSENCE compression function, one will notice that it has terms of a wide variety of degrees. For the maximum degree d of any of its inputs, the maximum degree of F is $7d$. At the same time, four of its seven inputs also exist in its resulting polynomial as linear terms. When a cube attack is performed, a few public input bits are used for the cube and the rest are all fixed to some value (in our implementation, zero). This fixing of the remaining public input bits tends to obliterate a lot of higher terms if the cubes being used are small. As a result, if small maxterms exist they will be found even in the presence of much larger potential maxterms. Because the structure of F creates this diversity of degree in terms, the degree of maxterms available for a cube attack is frequently far below the maximum degree. (Case in point: In our attack against $G_{256}(IV, M, 19)$ we solely used maxterms of degree 2 and 3, yet we found maxterms of up to degree 9, and it is likely that the degree of the output entire polynomial was considerably higher.) The reason we stopped our full attack at 19 rounds has to do with the existence and frequency of these low degree maxterms.

Before continuing, a little more notation should be laid down. In the $G_{256}(R, K, n)$ compression function (see Algorithm 6.1.1), there are two E permutations in use. We will refer to the E permutation that updates R' as E_R and the E permutation that updates K' as E_K. Furthermore we will refer to 8 registers used in E_R as $r0, \ldots, r7$ and the eight registers used in E_K as $k0, \ldots, k7$. We note that the E permutation follows a feedback shift register design (see Equation 6.1). In order to track registers as they move through the shift register, we will use a subscript that indicates how many times a register has passed the end of the shift register. So all registers in E_R start as $r0_0, \ldots, r7_0$. After eight rounds, they are $r0_1, \ldots, r7_1$, and after

10 rounds they are $r0_1, \ldots, r5_1, r6_2, r7_2$.

Consider now the structure of the ESSENCE compression function $G_{256}(R, K, n)$. The initialization vector is fed into E_R and sets the values of $r0_0, \ldots, r7_0$. The message block is fed into E_K and sets the values of $k0_0, \ldots, k7_0$. In our partial preimage cube attack, the secret value is $k0_0$ and $k1_0, \ldots, k7_0$ are the public value. The E_K permutation is cycled and is fed into E_R as the key parameter.

Attacking 8 to 16 rounds

Now consider the state of G after 8 rounds. From the definition of the E permutation, we know that for all $0 \leq i \leq 7$ we have ki_0 as a linear term of ri_1. Thus, a cube attack against $G_{256}(R, K, 8)$ succeeds by using a single maxterm of degree 0 (i.e. fixing all the public input bits) on the output bits corresponding to $r0_1$.

Now consider the state of G after 16 rounds. $r7_2$ contains $k7_1$ as a linear term which in turn contains $L(k0_0)$ as a linear term. Therefore, those bits of $k0_0$ which are not removed by the L function are still accessible as linear terms of $r7_2$ and can also be obtained through a maxterm of degree 0. The remaining bits of $k0_0$ must be obtained through the mixing done by the F function.

At this juncture, it is worth pointing out two things. First, the only part of the E permutation that can raise the degree of its resulting polynomial is the F function; all other components of E are linear. Secondly, the only part of E that can create a polynomial dependent on multiple bits from one register is the L function; all other components of E are bitwise operators.

After 16 rounds of G, $r7_2$, $r6_2$, and $r5_2$ contain maxterms of degree 1 that yield a linear combination of the bits of $k0_0$. This can only be accomplished through the work of the F and L functions. Thus, it is only after 16 of the 32 rounds in the ESSENCE compression function that some of the bits of $k0_0$ are no longer explicitly included as linear terms in the result, and only after 17 rounds (when $r7_2$ becomes $r7_3$) that none of the bits of $k0_0$ from its original linear inclusion are present as linear terms.

Attacking beyond 16 rounds

At this point our access to the bits of $k0_0$ is through the mixing of F and L. As we advance one round at a time beyond 16 rounds, it becomes increasingly

more difficult to find maxterms with corresponding superpolys of the bits of $k0_0$. After 19 rounds, $r7_2$, $r6_2$, and $r5_2$ have all passed the end of the shift register. We can now find maxterms of degree 2 and 3 in $r4_2$. After twenty rounds, we can only find maxterms as low as degree 7 in $r3_2$ with great difficulty.

We conjecture that this is due to the nature of the F function, and that not all registers have corresponding polynomials of equal degree. The F function takes all registers except the one passing the end of the shift register as input. As registers pass the end of the shift register and wrap around, they are updated with a new application of F which increases their degree. However, because F takes all the other registers as inputs, registers toward the end of an 8 round cycle are dependent on more registers that already passed the end of the shift register and wrapped and therefore have higher degree. Therefore, the degree the registers' corresponding polynomials increases from $r7$ at the lowest up to $r0$ at the highest.

As we have already noted, the structure of F lends itself to the existence of maxterms with degrees well below the degree of the polynomial they are found in. We conjecture that as we move along registers ri_2 with increasingly smaller i the input of registers with higher degree into F pushes the minimal degree maxterms into higher and higher degrees. As such, it is difficult to say how well the attack can scale theoretically to higher numbers of rounds. Because F creates such low degree terms, the attack will always depend on lower degree terms even though the degree of the polynomial as a whole would be far too large to successfully attack its maximal degree terms with a cube attack.

We therefore cannot conclude what the theoretical limit of a successful cube attack against the ESSENCE $G_{256}(R, K, n)$ compression function would be at this time. Further in-depth analysis of the F function of the E permutation is necessary to determine the nature of the existence and movement of low degree maxterms.

Analysis of the hash function

Finally, we would point out that even if a successful attack against the compression function could be mounted, this still might not compromise the ESSENCE hash function as a whole. Because of the 0-depth hash tree design incorporated in the official configuration of ESSENCE submitted to NIST, a message block is effectively compressed twice, once for the "chunk" it is a

part of and once again when that chunk's hash is compressed along with the final block to form the resulting hash.

The result of this is that in our experience a successful attack on the ESSENCE hash function requires the reduced-round variant to have half as many rounds as a corresponding reduced-round attack on the ESSENCE compression function. It should be noted that this is merely an observation based on empirical evidence, and we do not have theoretical analysis to back this up; it may very well be even more difficult to translate an attack on the compression function to the whole hash on higher rounds than we have attempted thus far.

6.1.5 Related work

Mouha, Thomsen, and Turan observed that the ESSENCE compression function was distinguishable from a random function in [24], specifically finding slid pairs (input pairs which produce states that are one step apart) and fixed points for the compression function.

We are not aware of any other written analysis of ESSENCE that has been released at this time.

6.2 Attack on Reduced-Round Keccak

6.2.1 The Keccak hash function

Keccak is another candidate submitted to NIST's SHA-3 hash competition [9]. Its distinguishing feature is that it is built using a "sponge construction" [7] in which the message is "absorbed" into a sponge piece by piece, and then the output hash is "squeezed" out of the sponge.

A complete mathematical definition of Keccak is beautifully and concisely given in [8] in the first two pages. We therefore refer the reader to [8] for details on Keccak's construction. However, to provide context for the analysis of our cube attack, we will reproduce the definition of the KECCAK-$f[b]$ permutation which serves as Keccak's compression function. It is defined as a permutation on a state a which is a three-dimensional array of elements in GF(2). The definition is as follows [8]:

> KECCAK-$f[b]$ is an iterated permutation, consisting of a sequence of n_r rounds R, indexed with i_r from 0 to $n_r - 1$. A round

consists of five steps:

$$R = \iota \circ \chi \circ \pi \circ \rho \circ \theta, \text{ with}$$

$$\begin{array}{lll}
\theta: & a[x][y][z] & \leftarrow a[x][y][z] + \sum_{y'=0}^{4} a[x-1][y'][z] + \sum_{y'=0}^{4} a[x+1][y'][z-1] \\
\rho: & a[x][y][z] & \leftarrow a[x][y][z-(t+1)(t+2)/2], \\
 & & \text{with } t \text{ satisfying } 0 \leq t < 24 \\
 & & \text{and } \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \text{ in } GF(5)^{2\times 2}, \\
 & & \text{or } t = -1 \text{ if } x = y = 0 \\
\pi: & a[x][y] & \leftarrow a[x'][y'], \text{with } \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix} \\
\chi: & a[x] & \leftarrow a[x] + (a[x+1]+1)a[x+2] \\
\iota: & a & \leftarrow a + RC[i_r]
\end{array}$$

The additions and multiplications between the terms are in GF(2).

It should be noted that the $RC[i_r]$ added in the ι function is a round constant. For more details on the KECCAK-$f[b]$ permutation, see [8, §1].

6.2.2 Cube attacks

To test Keccak's resistance to cube attacks, we reduced the number of rounds n_r and attempted to execute a cube attack against the 224-bit hash size version of Keccak. The standard number of rounds for Keccak is 18. We successfully found cubes on reduced-round variants of Keccak of up to 4 rounds.

Since cube attacks are structured to recover some kind of secret, we configured 224-bit Keccak as a very naive HMAC. We created a 224 bit message by concatenating a 112 bit secret value and a 112 bit public value and then hashing it[3]. This configuration was then subjected to a cube attack.

[3]Recall that in our analysis of ESSENCE we only restricted the secret input to 32 bits in order to ease analysis. In this case, such a restriction would not affect our analysis, and so we simply evenly divide the input between secret and public bits.

For 4 round 224-bit Keccak, we found 112 cubes with linearly independent superpolys on the secret input bits. The full cube search and attack took approximately 10 minutes on a desktop PC. This could have been sped up by searching for fewer cubes and brute force guessing some of the secret bits.

In Table 6.4 we give our precomputation of the attack. In each row we give a superpoly $p_{S(I)}$, the cube indices I that define the corresponding maxterm t_I, and the output bit the superpoly resides on. Note that because we attacked all output bits in parallel, a single maxterm may be associated with multiple superpolys on multiple output bits. In the superpoly polynomials, an x_j variable refers to bit j of the secret input. Likewise, a cube index i refers to bit i of the public input. Note that these bit indices are indices into the secret and public inputs, not indices into the hash input, so both secret and public bit indices start from 0. To convert this to hash input bit indices using our partial pre-image attack configuration, the secret bit indices would remain the same and the public bit indices would be increased by 112.

Finally, we should note that the maxterms given in this table (via their cube indices) are simply the first ones we found that had linearly independent superpolys; we did a random walk and every time we found a maxterm with a superpoly that was linearly independent to those we already had, we added it to our list. There are certainly more such maxterms, but only these were necessary to mount an attack.

Table 6.4: Maxterms for 4 round 224-bit Keccak

Superpoly Polynomial	Cube Indices	Output Bit Index
$1 + x_0$	{0,31,36,46,62,65,77,80,84,91,94,110}	202
$1 + x_1$	{1,15,20,24,25,35,57,58,73,80,95,106}	200
$1 + x_2$	{1,7,12,35,40,44,66,82,92,93,104,106}	168
$1 + x_4$	{7,10,11,19,38,43,46,52,66,70,77,83}	166
x_5	{12,17,19,23,43,50,69,78,83,90,104}	177
$1 + x_6$	{4,9,14,23,24,37,53,71,86,90,103,104}	3
x_7	{6,14,16,23,30,36,43,64,87,101,105,111}	0
x_8	{14,23,30,41,45,59,60,64,73,85,90,106}	191
$1 + x_9$	{0,22,29,39,43,44,62,64,70,89,99}	97
x_{10}	{3,9,17,19,25,28,40,50,58,65,88}	173
x_{11}	{0,31,36,46,62,65,77,80,84,91,94,110}	55
$1 + x_{12}$	{9,11,26,29,40,80,84,91,95,106,109,110}	163
$1 + x_{17}$	{4,9,14,23,24,37,53,71,86,90,103,104}	162
$1 + x_{19}$	{6,14,16,23,30,36,43,64,87,101,105,111}	94
$1 + x_{21}$	{1,3,8,12,41,60,62,67,68,73,84,98}	212
$1 + x_{22}$	{3,6,22,36,42,44,45,49,54,66,84,104}	45
Continued on the next page ...		

Table 6.4: (continued)

x_{24}	{5,10,13,28,39,41,46,48,58,62,76,77}	108
x_{26}	{12,17,19,23,43,50,69,78,83,90,104}	43
$1 + x_{27}$	{4,9,14,23,24,37,53,71,86,90,103,104}	27
x_{28}	{17,30,35,37,41,45,62,70,76,80,105}	118
$1 + x_{29}$	{38,40,59,61,66,80,84,92,98,104,111}	71
$1 + x_{18} + x_{31}$	{7,10,11,19,38,43,46,52,66,70,77,83}	61
$1 + x_{31}$	{27,31,33,52,54,59,60,80,83,92,108,111}	20
$1 + x_{32}$	{1,5,9,13,15,42,47,49,54,80,83,109}	110
$1 + x_{33}$	{20,36,39,40,42,55,56,60,61,71,99}	149
x_{34}	{18,21,27,43,51,62,69,77,88,89,98,103}	21
x_{35}	{6,14,16,23,30,36,43,64,87,101,105,111}	43
$1 + x_{36}$	{16,30,47,52,75,76,77,82,85,90,109,110}	1
$1 + x_{37}$	{2,6,26,41,57,68,89,95,99,101,110}	95
x_{38}	{0,5,14,19,38,43,44,52,74,79,91,92}	44
$1 + x_{42}$	{27,31,33,52,54,59,60,80,83,92,108,111}	43
x_{43}	{1,3,8,12,41,60,62,67,68,73,84,98}	168
x_{47}	{6,9,14,33,42,57,65,66,73,74,75,81}	9
x_{49}	{20,36,39,40,42,55,56,60,61,71,99}	15
$1 + x_{51}$	{4,9,14,23,24,37,53,71,86,90,103,104}	166
$1 + x_{52}$	{4,9,14,23,24,37,53,71,86,90,103,104}	161
$1 + x_{53}$	{3,9,17,19,25,28,40,50,58,65,88}	167
$x_{16} + x_{54}$	{8,9,14,25,52,53,54,61,66,67,69,92}	152
$1 + x_{54}$	{1,5,10,26,27,53,55,57,77,89,91,107}	179
$1 + x_{48} + x_{54}$	{1,3,29,43,56,58,79,82,84,85,97,100}	50
x_{55}	{1,7,11,14,18,23,60,79,91,100,101,109}	22
x_{56}	{3,6,22,36,42,44,45,49,54,66,84,104}	155
$x_{31} + x_{53} + x_{57}$	{0,10,13,18,38,64,81,88,90,92,94}	144
$1 + x_{58}$	{9,11,13,16,42,44,58,67,68,78,94,109}	111
$1 + x_9 + x_{41} + x_{60}$	{4,11,26,38,45,53,59,60,61,81,89,107}	33
$1 + x_{61}$	{4,8,17,24,29,44,56,65,76,85,103,110}	97
$1 + x_{62}$	{4,8,17,24,29,44,56,65,76,85,103,110}	110
$1 + x_{63}$	{13,18,20,33,42,62,63,87,102,107,108}	47
$1 + x_{64}$	{2,17,38,42,46,55,59,66,75,96,100,107}	80
$1 + x_{65}$	{5,6,16,22,29,35,43,66,73,76,102,106}	168
$x_{41} + x_{65}$	{7,25,50,52,53,54,74,78,79,87,100}	160
$1 + x_{66}$	{14,23,30,41,45,59,60,64,73,85,90,106}	77
$1 + x_{10} + x_{39} + x_{56} + x_{68}$	{3,8,23,27,30,33,41,57,67,85,91,100}	14
$1 + x_6 + x_{69}$	{14,23,30,41,45,59,60,64,73,85,90,106}	87
x_{70}	{9,12,18,33,65,68,83,90,97,101,103}	57
$x_{68} + x_{71}$	{16,24,26,38,48,50,58,64,84,91,93}	138
x_{72}	{0,31,36,46,62,65,77,80,84,91,94,110}	96

Continued on the next page ...

Table 6.4: (continued)

$1 + x_{73}$	{9,12,16,17,20,42,45,58,103,107,109}	133
$1 + x_{10} + x_{44} + x_{73}$	{2,4,6,26,32,45,46,47,53,75,89,110}	86
$1 + x_{74}$	{4,9,14,23,24,37,53,71,86,90,103,104}	41
$1 + x_{75}$	{0,5,14,19,38,43,44,52,74,79,91,92}	183
x_{77}	{4,9,14,23,24,37,53,71,86,90,103,104}	79
$1 + x_2 + x_3 + x_{13} + x_{77}$	{8,9,11,34,41,50,58,68,75,80,85,102}	115
x_{78}	{4,9,14,23,24,37,53,71,86,90,103,104}	11
$1 + x_{79}$	{17,30,35,37,41,45,62,70,76,80,105}	123
$x_{15} + x_{79}$	{1,5,9,13,15,42,47,49,54,80,83,109}	56
$1 + x_{81}$	{14,33,37,40,47,52,78,79,81,84,93,100}	80
$1 + x_{82}$	{14,23,30,41,45,59,60,64,73,85,90,106}	99
$x_{14} + x_{36} + x_{78} + x_{82}$	{1,3,8,12,41,60,62,67,68,73,84,98}	181
$x_{20} + x_{83}$	{4,9,14,23,24,37,53,71,86,90,103,104}	36
$1 + x_{85}$	{27,30,51,52,54,76,82,87,96,99,111}	137
$1 + x_{23} + x_{81} + x_{86}$	{13,33,52,55,58,72,73,89,96,104,109}	110
x_{86}	{4,9,14,23,24,37,53,71,86,90,103,104}	113
$1 + x_8 + x_{59} + x_{86}$	{27,31,33,52,54,59,60,80,83,92,108,111}	21
$x_9 + x_{51} + x_{87}$	{4,11,26,38,45,53,59,60,61,81,89,107}	123
$x_{25} + x_{88}$	{27,31,33,52,54,59,60,80,83,92,108,111}	25
$1 + x_{88}$	{4,5,24,28,44,62,65,73,80,84,93,110}	175
x_{89}	{13,18,20,33,42,62,63,87,102,107,108}	154
$1 + x_{90}$	{14,23,30,41,45,59,60,64,73,85,90,106}	148
$x_{28} + x_{91}$	{20,43,45,52,64,83,87,94,97,98,100}	85
$1 + x_{92}$	{0,7,19,21,41,42,52,56,61,66,80,83}	86
$x_{77} + x_{93}$	{13,33,52,55,58,72,73,89,96,104,109}	6
$1 + x_{61} + x_{80} + x_{93}$	{4,9,14,23,24,37,53,71,86,90,103,104}	198
$x_{30} + x_{93}$	{27,30,51,52,54,76,82,87,96,99,111}	162
$1 + x_{94}$	{27,31,33,52,54,59,60,80,83,92,108,111}	73
$1 + x_{95}$	{3,9,17,19,25,28,40,50,58,65,88}	66
$1 + x_{96}$	{4,9,14,23,24,37,53,71,86,90,103,104}	212
$x_{75} + x_{97}$	{4,9,14,23,24,37,53,71,86,90,103,104}	71
x_{98}	{27,31,33,52,54,59,60,80,83,92,108,111}	101
$1 + x_{46} + x_{64} + x_{99}$	{0,5,36,37,55,62,72,80,81,88,100}	185
$x_{23} + x_{100}$	{6,12,19,20,58,62,71,75,78,97,102,103}	141
$x_{40} + x_{100}$	{5,6,16,22,29,35,43,66,73,76,102,106}	25
$1 + x_{71} + x_{100}$	{12,15,27,29,31,39,44,46,50,74,84,106}	29
$x_{28} + x_{38} + x_{101}$	{4,9,14,23,24,37,53,71,86,90,103,104}	177
$1 + x_{50} + x_{101}$	{27,31,33,52,54,59,60,80,83,92,108,111}	57
$1 + x_{102}$	{15,25,30,39,47,49,57,77,80,82,107}	176
$1 + x_{84} + x_{102}$	{16,24,26,38,48,50,58,64,84,91,93}	139
$1 + x_{83} + x_{102}$	{1,29,41,48,50,54,56,77,84,85,95}	147

Continued on the next page ...

Table 6.4: (continued)

$1 + x_{13} + x_{103}$	{7,10,11,19,38,43,46,52,66,70,77,83}	133
x_{103}	{4,9,14,23,24,37,53,71,86,90,103,104}	42
x_{104}	{4,9,14,23,24,37,53,71,86,90,103,104}	90
x_{105}	{1,31,46,49,52,59,61,64,89,99,103}	79
$1 + x_{106}$	{27,31,33,52,54,59,60,80,83,92,108,111}	192
$1 + x_{76} + x_{106}$	{1,15,20,24,25,35,57,58,73,80,95,106}	219
$1 + x_{100} + x_{107}$	{9,12,16,24,25,26,31,38,40,68,74,84}	190
$1 + x_{108}$	{18,21,27,43,51,62,69,77,88,89,98,103}	193
$1 + x_{45} + x_{108}$	{9,11,13,16,42,44,58,67,68,78,94,109}	26
x_{109}	{7,10,11,19,38,43,46,52,66,70,77,83}	215
$x_{46} + x_{109}$	{12,17,19,23,43,50,69,78,83,90,104}	138
$1 + x_{110}$	{4,8,17,24,29,44,56,65,76,85,103,110}	201
$1 + x_{111}$	{9,11,13,16,42,44,58,67,68,78,94,109}	60
$1 + x_{67} + x_{102} + x_{111}$	{9,11,13,16,42,44,58,67,68,78,94,109}	205

These maxterms were used in an attack which was successful in fully recovering the secret. Since the largest maxterm we used was of degree 12 and all 112 secret bits were attacked (i.e. no brute force guessing was used), the complexity of the attack is less than 2^{19} hashing operations.

6.2.3 Analysis

When experimenting with our cube attack, we noticed that when we increased the number of rounds in the reduced-round variant of Keccak we were attacking the maximum degree of the maxterms found seemed to roughly double for each round. An examination of the KECCAK-$f[b]$ round function casts some light on this.

If one examines the functions which compose the round function, one will see that all but one of them is linear in GF(2). θ and ι are linear sums. ρ and π perform copies within the state array which do not affect the degree of the underlying polynomials; they just move the polynomials around within the state. χ is the only nonlinear member of the round function due to the term $(a[x+1]+1)a[x+2]$. Presuming $a[x+1]$ and $a[x+2]$ contain polynomials of the state that are of maximum degree, χ could at most double the degree of the state's corresponding polynomials. So one round of KECCAK-$f[b]$ can at most double the degree of any polynomial it receives in its state. Therefore, the maximum degree of the KECCAK-$f[b]$ output polynomials is 2^{n_r} where n_r is the number of rounds.

This matches our empirical observations. Since the Keccak sponge function already contains a full hash output worth of data after absorbing the message, the "squeeze" phase is never performed in an ordinary hash. (And in fact, the reference implementation never calls the Squeeze function in the process of hashing a message.) Since our message always fit in one message block, Keccak only ran the KECCAK-$f[b]$ permutation on it once, and as such we were almost directly interacting with the KECCAK-$f[b]$ permutation. So when we increased the number of rounds in the reduced-round variant we were attacking, the degree of the maxterms roughly doubled because the maximum degree of the Keccak output polynomials had doubled. A similar observation was made by the Keccak design team in [9, §5.9.3.2].

If the maximum degree of Keccak's output polynomials is 2^{n_r}, then a cube attack would only be practical against up to 7 rounds of 224 or 256 bit Keccak. Since the official configuration of Keccak submitted to NIST has 18 rounds, this is well within the margin of safety.

6.2.4 Related work

Aumasson and Khovratovich released the first third-party written analysis of Keccak in [6]. They applied a number of automated tools to the KECCAK-$f[1600]$ permutation, attempting to solve the "constrained input constrained output" problem and using cube testers[4] to explore the KECCAK-$f[1600]$ permutation's algebraic properties. Their work does not include any kind of partial pre-image attack, but instead focuses on making observations on Keccak's structure. We feel that their work and ours complement each other, and we recommend their paper to anyone interested in pursuing further analysis of Keccak.

We are not aware of any other written analysis of Keccak that has been released at this time.

6.3 Preliminary Results for Other Hashes

Aside from ESSENCE and Keccak, we also made brief attempts at applying cube attacks to other hash functions to see if there might be the potential for an attack worth researching. The configuration was similar to what was

[4] See [5] for more information on cube testers.

done with ESSENCE and Keccak: A simple partial pre-image attack against the hash configured to be at a reduced strength.

Blake We were able to find cubes of size 4-10 on Blake reduced to 1 round. Since Blake has 10 rounds in its official configuration, we did not consider the discovery promising enough to pursue.

CubeHash Against CubeHash1/16-224 we found nothing. Since that is only 1 round of 10 steps and the official CubeHash configuration is CubeHash8/1, which has 8 rounds, we did not pursue the matter further.

Skein We found no cubes when attacking 8 round Skein, and the official configuration has 72 rounds.

MCSSHA3 Our results for MCSSHA3 were peculiar. To begin, we attacked only the initialization and update phases of MCSSHA3, leaving off the finalization steps. We immediately found maxterms for any degree we tested on (including degree 0), yet all their corresponding superpolys only contained variables for bits 0, 1, and 2 of the secret. This was against full round MCSSHA3, i.e. we did not weaken it in any way except for removing the finalization phase. We found these results rather puzzling and did not pursue them at the time due to our more promising discoveries with ESSENCE and Keccak.

This presents an opportunity for further research which may prove enlightening with regard to the algebraic structure of MCSSHA3.

Chapter 7

Related Work

There are a few other examples of cube attacks being applied to hash functions that deserve mention.

MD6 attacks The original discoverers of cube attacks have collaborated with the MD6 design team to test the resilience of MD6 to cube attacks [31]. Thus far the results are "*very* preliminary and tentative," but they have determined that it appears cube attack techniques can be used to distinguish MD6 from a random function at up to 15 rounds. There is the possibility that there exists a cube attack which could recover the MD6 key in the same number of rounds.

This should be considered by no means alarming since for the smallest digest size NIST requires (224 bits), MD6 has 96 rounds.

Cube testers Jean-Philippe Aumasson in collaboration with Itai Dinur, Willi Meier, and Adi Shamir devised a modification of cube attacks called "cube testers" [5]. In essence, they can be used to test a function for the feasibility of a cube attack or differentiate it from a random function. They cannot however carry out the attack. Aumasson has applied this technique in analysis of a number of SHA-3 candidates, including Shabal [4] and Keccak [6].

Chapter 8

Conclusion

Having now explained cube attacks and demonstrated their application to hash functions, there are a few closing observations we would like to make.

8.1 Effectiveness of Pure Cube Attacks

As can be observed from our results, cube attacks aren't powerful in all cases. While we did give two examples of their successful use in reduced-round attacks, there were even more hashes which they did nothing against.

A cube attack's success depends on the cryptosystem under attack having a low degree. However, today most cryptosystems are based on a design that uses rounds. This tends to drive up degree and quickly make a cube attack impractical, except in the rare cases where the rounds do very little to increase the degree and there aren't very many of them. Hash functions tend to have a considerable number of rounds though, and this tends to limit the usefulness of a pure cube attack to corner cases.

8.2 New Areas for Research

This should not be taken as a denouncement of cube attacks. They are a fairly recent discovery and like any other discovery there is still much room to enhance and improve them.

For the time being, one area that may be worth pursuing is hybrid attacks involving cube attacks, as were briefly discussed earlier in this thesis.

Researching ways to combine cube attacks with existing well understood attacks in some kind of meet-in-the-middle fashion could lead to a new variety of very powerful attacks.

In short, while pure cube attacks as we currently understand them may not yet be cryptosystem killers, there is considerable potential for enhancement both through studying the details of how cube attacks interact with the internals of various cryptosystems as well as studying their use in hybrid attacks. So far they have already been used to show weaknesses in hash functions and stream ciphers and new developments and improvements in our understanding of them are already being made. Their future as a cryptanalytical technique appears promising indeed.

Bibliography

[1] SHA-1 collision search. http://boinc.iaik.tugraz.at/. Retreived on 4/25/2009.

[2] NIST SHA-3 competition first round candidates. http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/submissions_rnd1.html, 2009.

[3] NIST SHA-3 competition round one webpage. http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/index.html, 2009.

[4] Jean-Philippe Aumasson. On the pseudorandomness of Shabal's keyed permutation. Official comment on the NIST Hash Competition, 2009.

[5] Jean-Philippe Aumasson, Itai Dinur, Willi Meier, and Adi Shamir. Cube testers and key recovery attacks on reduced-round MD6 and Trivium. *Fast Software Encryption. Springer, Heidelberg*, 2009.

[6] Jean-Philippe Aumasson and Dmitry Khovratovich. First analysis of Keccak. http://131002.net/data/papers/AK09.pdf, 2009.

[7] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. In *Ecrypt Hash Workshop*, 2007. http://csrc.ncsl.nist.gov/groups/ST/hash/documents/JoanDaemen.pdf.

[8] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak specifications. Submission to NIST, 2008. http://keccak.noekeon.org/Keccak-specifications.pdf.

[9] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak sponge function family main document. Submission to NIST (updated), 2009.

[10] John Black and Martin Cochran. A study of the MD5 attacks: Insights and improvements. In *In Fast Software Encryption*, pages 262–277. SpringerVerlag, 2006.

[11] Philip Hawkes Cameron McDonald and Josef Pieprzyk. SHA-1 collisions now 2^{52}. In *Rump Session of EuroCrypt '09*. Annoucement, 2009.

[12] Christophe De Canniere, Florian Mendel, and Christian Rechberger. Collisions for 70-step SHA-1: On the full cost of collision search. *Lecture Notes in Computer Science*, 4876, 2007.

[13] Christophe De Canniere and Christian Rechberger. Finding SHA-1 characteristics: General results and applications. *Lecture Notes in Computer Science*, 4284, 2006.

[14] Itai Dinur and Adi Shamir. Cube attacks on tweakable black box polynomials. Cryptology ePrint Archive, Report 2008/385, 2008. `http://eprint.iacr.org/`.

[15] Hans Dobbertin. Cryptanalysis of MD5 compress. In *Rump Session of EuroCrypt '96*. Annoucement, 1996.

[16] Dan Kaminsky. MD5 to be considered harmful someday. Cryptology ePrint Archive, Report 2004/357, 2004. `http://eprint.iacr.org/`.

[17] Vlastimil Klima. Tunnels in hash functions: MD5 collisions within a minute. Cryptology ePrint Archive, Report 2006/105, 2006. `http://eprint.iacr.org/2006/105`.

[18] Arjen Lenstra and Benne de Weger. On the possibility of constructing meaningful hash collisions for public keys. In *ACISP*, volume 3574, pages 267–279. Springer, 2005.

[19] Arjen Lenstra, Xiaoyun Wang, and Benne de Weger. Colliding X.509 certificates. In *Proceedings of ACISP*, 2005.

[20] Mangojuice. Merkle-Damgård construction illustration. `http://en.wikipedia.org/wiki/File:Merkle-Damgard.png`, 2006.

[21] Jason Worth Martin. ESSENCE: A candidate hashing algorithm for the nist competition. Submission to NIST, 2008.

[22] Jason Worth Martin. ESSENCE: A family of cryptographic hashing algorithms. Submission to NIST, 2008.

[23] Alfred Menezes, Paul Van Oorschot, and Scott Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[24] Nicky Mouha, Søren Thomsen, and Meltem Sönmez Turan. Observations of non-randomness in the essence compression function. `http://www.mat.dtu.dk/people/S.Thomsen/essence/Essence-obs.pdf`, 2009.

[25] National Institute of Standards and Technology. Tentative timeline of the development of new hash functions. `http://csrc.nist.gov/groups/ST/hash/timeline.html`, November 2007.

[26] National Institute of Standards and Technology. Cryptographic hash algorithm competition. `http://csrc.nist.gov/groups/ST/hash/sha-3/index.html`, January 2008.

[27] Department of Commerce: National Institute of Standards and Technology. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. *Federal Register*, 72(212):62212 – 62220, November 2007. Docket No.: 070911510-7512-01.

[28] National Institute of Standards and Technology. FIPS 180-1: Secure hash standard. Technical report, United States Department of Commerce, 1995.

[29] National Institute of Standards and Technology. FIPS 180-2: Secure hash standard. Technical report, United States Department of Commerce, 2002.

[30] Ron Rivest. The MD5 message-digest algorithm. Technical report, IETF, 1992.

[31] Ronald Rivest. The MD6 hash function – A proposal to NIST for SHA-3. Submission to NIST, 2008.

[32] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition*. Wiley, October 1995.

[33] Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. MD5 to be considered harmful today: Creating a rogue CA certificate. `http://www.win.tue.nl/hashclash/rogue-ca`, 2008.

[34] Marc Stevens. On collisions for MD5. Master's thesis, Eindhoven University of Technology, 2007.

[35] Marc Stevens, Alex Sotirov, Jake Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. Cryptology ePrint Archive, Report 2009/111, 2009. `http://eprint.iacr.org/`.

[36] Douglas Stinson. *Cryptography Theory and Practice*. Discrete Mathmatics and its Applications. Chapman & Hall/CRC, third edition, 2006.

[37] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive, Report 2004/199, 2004. `http://eprint.iacr.org/2004/199`.

[38] Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, and Xiuyuan Yu. Cryptanalysis of the hash functions MD4 and RIPEMD. In *Advances in Cryptology–EUROCRYPT*, volume 3494, pages 1–18. Springer, 2005.

[39] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. *Lecture Notes in Computer Science*, 3621, 2005.

[40] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. *Lecture Notes in Computer Science*, 3494, 2005.

[41] Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin. Efficient collision search attacks on SHA-0. *Lecture Notes in Computer Science*, 3621:1–16, 2005.