ANALYSIS OF ARX ROUND FUNCTIONS IN SECURE HASH FUNCTIONS

by Kerry A. McKay

B.S. in Computer Science, May 2003, Worcester Polytechnic Institute
M.S. in Computer Science, May 2005, Worcester Polytechnic Institute

A Dissertation submitted to

the Faculty of
The School of Engineering and Applied Science
of The George Washington University
in partial satisfaction of the requirements
for the degree of Doctor of Science

May 15, 2011

Dissertation directed by

Poorvi L. Vora
Associate Professor of Computer Science

The School of Engineering and Applied Science of The George Washington University certifies that Kerry A. McKay has passed the Final Examination for the degree of Doctor of Science as of March 25, 2011. This is the final and approved form of the dissertation.

ANALYSIS OF ARX ROUND FUNCTIONS IN SECURE HASH FUNCTIONS

Kerry A. McKay

Dissertation Research Committee:

Poorvi L. Vora, Associate Professor of Computer Science,
Dissertation Director

Gabriel Parmer, Assistant Professor of Computer Science,
Committee Member

Rahul Simha, Associate Professor of Engineering and Applied Science,
Committee Member

Abdou Youssef, Professor of Engineering and Applied Science,
Committee Member

Lily Chen, Mathematician, National Institute of Standards and
Technology, Committee Member

Dedication

To my family and friends, for all of their encouragement and support.

Acknowledgements

This work was supported in part by the NSF Scholarship for Service Program, grant DUE-0621334, and NSF Award 0830576.

I would like to thank my collaborators Niels Ferguson and Stefan Lucks for our collaborative research on the analysis of the CubeHash submission to the SHA-3 competition. I would also like to thank the entire Skein team for their support.

Abstract

Analysis of ARX Round Functions in Secure Hash Functions

A new design paradigm for symmetric-key design primitives, such as hash functions and block ciphers, is the Addition-Rotation-XOR (ARX) paradigm. ARX functions rely on the combination of addition modulo 2^n, word rotation and exclusive-or to increase the difficulty of applying traditional linearity-based attacks. This work provides contributions in the analysis of ARX functions.

This dissertation introduces a new analytic technique, pseudo-linear cryptanalysis, which takes advantage of linear properties of ARX-functions over the groups $\mathbb{Z}_2^n$ and $\mathbb{Z}_{2^n}$. This is in contrast to traditional linear analysis, which has largely focused on linearity over $\mathbb{Z}_2$. Pseudo-linear cryptanalysis can be used on any ARX-based symmetric primitive, and is particularly useful for block ciphers and iterative hash functions containing round functions. The dissertation also presents a variant that can be used for differential attacks, and extends the branch number diffusion metric for ARX ciphers that use large words.

Secure hash functions are among those primitives that may be built on ARX-functions. The National Institute of Standards and Technology is currently in the process of selecting the next US standard secure hash algorithm, SHA-3, which will be used in everyday applications such as secure online sessions and password-based authentication. Two of the five finalists in the SHA-3 competition are based on ARX functions. This dissertation applies pseudo-linear cryptanalysis, truncated differentials, and new ideas for computing branch numbers to SHA-3 finalist Skein. It also presents improved attacks on second-round SHA-3 candidate CubeHash as well as structural analysis of its symmetry classes.

Table of Contents

List of Figures

List of Tables

Chapter 1 – Introduction

This dissertation analyzes weaknesses of secure hash functions built on the popular Addition-Rotation-XOR (ARX) block cipher primitive, and presents approaches to minimize security vulnerabilities.

The secure hash function is an important security primitive for the purpose of ensuring data integrity. Loosely speaking, a secure hash function compresses data in a manner that is difficult to invert. More concretely, given an element from the range of the secure hash function, it should be infeasible to find a corresponding element of the domain. Further, it should be infeasible to find any two elements of the domain that map to the same element in the range. The properties of secure hash function designs are in focus at the moment because NIST is in the process of choosing a new addition to the Federal Information Processing Systems (FIPS) standard for secure hash functions following the discovery of a projected vulnerability in the current standard.

1.1 Background

The Addition-Rotation-XOR (ARX) primitive has recently been used to design block ciphers, which, in turn, have been used for secure hash function designs. Block ciphers provide confidentiality by transforming one string (message) into another (ciphertext) of the same length. That is, a block cipher provides a permutation (one-to-one, onto function) of the space of all messages. The message is first partitioned into fixed-size blocks (it may need to be padded so that the padded message length is a multiple of block length), and then the encryption algorithm is applied to each block. The permutation is determined by the key, and it should be infeasible to discover or guess

the key. It should also be infeasible to relate a particular input with its output without knowledge of the key. The permutation is achieved by multiple iterations of a *round function*. These round functions often appear as components in hash functions.

Given a set of plaintexts and corresponding ciphertexts encrypted with a fixed key, it is possible to discover the key as follows. Try every key (brute force attack). For each key, check if each plaintext, on encryption, gives the corresponding ciphertext. If not, reject the key and go on to the next one. This attack, is, however, exponential in the length of the key, and is hence considered infeasible. Thus, cryptanalysis attempts to find relationships among a smaller group of key bits and bits of the plaintext and ciphertext so that a brute force attack may focus on smaller groups of key bits, resulting in a considerable reduction in attack complexity.

Linear cryptanalysis is an analytical technique that approximates the pseudorandom permutation for a smaller number of bits than the block size. The approximations involve bits of plaintext, ciphertext, and key. For example, consider a cipher with 64 bits of state and key. If k_h is a bit of key, p_i is a bit of plaintext, and c_j is a bit of ciphertext, a linear approximation may look like $p_4 \oplus p_8 \oplus c_{15} \oplus c_{16} = k_{23} \oplus k_{42}$. That is, the expression uses plaintext bits at index 4 and 8, ciphertext bits 15 and 16, and key bits 23 and 42. If this approximation has a large bias (occurs with probability significantly different from $\frac{1}{2}$), then an adversary can use it to discover the key with reduced effort. We write the probability that the approximation is true as $Pr[p_4 \oplus p_8 \oplus c_{15} \oplus c_{16} = k_{23} \oplus k_{42}] = \frac{1}{2} \pm \alpha$, where $|\alpha|$ is the bias.

To discover information about the key, the adversary collects a large set of plaintexts and their corresponding ciphertexts obtained by encryption with the target key. For all possible values of k_{23} and k_{42}, a and b, the adversary tries the equation $p_4 \oplus p_8 \oplus c_{15} \oplus c_{16} = a \oplus b$ on each plaintext-ciphertext pair and stores how often the equation is true in $correct_{ab}$. From each count, she subtracts the number of times one

expects the approximation to be true at random in order to obtain the bias. That is, $bias_{ab} = |correct_{ab} - \frac{pairs}{2}|$, where $pairs$ is the number of plaintext-ciphertext pairs. For each pair, the probability of guessing the parity correctly at random is $\frac{1}{2}$, hence $\frac{1}{2} \times pairs$ must be subtracted from the adversary's counter, $correct_{ab}$, to obtain the bias of the expression. The a, b values that produce the largest biases are the most likely to be the correct values for the key bits k_{23} and k_{42}.

The ability to partition the key into independent parts allows an adversary to reduce the complexity of guessing the secret key. For example, let us consider a cipher that contains 64 bits of key. If a sufficient linear approximation exists that involves only 32 bits of key, then the expected amount of work to get the key is reduced from 2^{63} to $2^{32} + 2^{32} = 2^{33}$ guesses.

Differential cryptanalysis is based on differences between pairs of plaintext and pairs of ciphertext. The relationship between a pair of plaintexts, (P, P'), where P' has a known difference to P, leads to a particular difference in the ciphertext pair (C, C'). In other words, it deals with the probability that a particular difference in (P, P') leads to a particular difference in (C, C'). For example, let $P' = P \oplus \Delta$, $C' = C \oplus \delta$. Then the probability that the difference Δ in the plaintexts leads to a difference of δ in the ciphertexts is written $Pr[\Delta \rightarrow \delta] = \alpha$. All bits of state are used for the general attack, but only a subset of the bits are needed for a truncated differential attack.

Hash function designs that use block ciphers typically rely on block cipher security properties for the security of the hash function. Thus, block ciphers used in hash function, like all other block ciphers, are expected to be resilient to attacks such as linear and differential cryptanalysis.

1.2 The SHA-3 Competition

The National Institute of Standards and Technology (NIST) has invited proposals for the next standard of the Secure Hash Algorithm (SHA-3) [46] from the international community of cryptologists. Following the success of the Advanced Encryption Standard competition [45] for the current block cipher standard, the SHA-3 selection process is also in the form of a competition. The proposals were submitted in October 2008. Currently, the research community is analyzing the various submissions to determine security vulnerabilities and to understand the efficiency properties of hardware and software implementations of the candidates. The open design and analysis process increases the assurance that the selected algorithm will meet security and efficiency requirements. A considerable amount of time and effort goes into standardizing an algorithm, and even more goes into algorithm transition in the industry. Thus, it is important that a strong algorithm is selected, ensuring long-term security and a smooth transition.

NIST received 64 submissions for the SHA-3 competition, of which 51 advanced to Round One and 14 to Round Two. There are currently five algorithms in the final round, two of which are ARX designs.

1.3 Contributions

Several contributions are presented in this dissertation. First, we present a new cryptanalytic technique that extends linear cryptanalysis which is defined over the over exclusive-or operation (i.e. $\mathbb{Z}_2$) to ARX-based round functions that perform addition over $\mathbb{Z}_2^n$ and $\mathbb{Z}_{2^n}$ where n is large (for example, $n = 64$) [41]. From the point of linear cryptanalysis, each bit of an ARX-based output is related to the

XOR of input bits and a carry, which is not a linear function of the input bits. Hence, the carry function provides non-linearity when addition is performed modulo 2^n for $n > 1$. Thus functions based on addition modulo 2^n for $n > 1$ are resilient to traditional linear cryptanalysis. However, we demonstrate that our technique, pseudo-linear cryptanalysis, can exploit the fact that addition modulo 2^n can be approximated with a function that is linear over larger strings of input bits (and not over single bits). Pseudo-linear cryptanalysis is particularly useful in analyzing ARX-based block ciphers. It can also be used in conducting preimage and collision attacks on hash functions.

We present results obtained by applying pseudo-linear cryptanalysis to Threefish, the block cipher used in the SHA-3 candidate Skein [23] hash function proposal. This technique is applied to two Threefish variants: Threefish-256 and Threefish-512.

A variation on pseudo-linear cryptanalysis can be used to reduce the effort involved in truncated differential attacks. An example application is provided for four rounds of Threefish-512.

We discuss potential pitfalls that could arise from a direct application of the branch number to ARX functions – particularly those that use large word sizes. An alternative method for applying the branch number metric is provided, as well as an alternative metric specific to ARX functions.

Finally, we present an improved analysis of the symmetric properties of the round function of SHA-3 candidate CubeHash [7]. Our CubeHash analysis [22] leads to the current best preimage and multicollision attacks known on it, and also shows that the symmetry classes have an algebraic structure.

Chapter 2 – Background and Previous Work

Before discussing our analysis of candidate algorithms, it is important to set the stage. We will be analyzing the secure hash functions both in terms of their internal components, and as a whole. The component of primary concern in this work is the pseudorandom permutation that is, or closely resembles, a block cipher. For clarity, we first discuss block ciphers and previous work in their analysis. After that, we consider attacks on secure hash functions in their entirety.

This chapter covers four main topics:

- brief overview of block ciphers
- previous work on block ciphers
- brief overview of iterated secure hash functions
- previous work on the SHA-3 candidates of interest

2.1 Brief Overview of Block Ciphers

The essential idea of encryption is to take a message m and map it to another message y such that the connection between m and y cannot be discerned without access to the encryption key. Let $Encrypt_k()$ represent the function that performs this pseudorandom permutation, using parameter k – the key. Then $y = Encrypt_k(m)$ and the permutation – and therefore y – is different for each distinct value of k, assuming m is fixed.

Unlike popular asymmetric (public key) cryptosystems, block ciphers are not generally based on hard number-theoretic problems. They are based upon Shannon's principles of *diffusion* and *confusion* [52] for secure ciphers. Diffusion requires that

the statistical properties of the cipher be hidden so that a large amount of data must be collected, and intensive analysis performed, in order to break the cipher. High diffusion prevents an adversary from isolating a subset of bits and using statistics to gain information about the key with accuracy better than random guessing. Confusion requires that the relationship between the key and ciphertext be complex.

Block ciphers partition data into fixed-size blocks, and run the encryption or decryption algorithm on each block, independently. There are several *modes of operation* that link that multiple blocks of an encrypted message together in various ways. For the purposes of this work, we only consider the encryption/decryption operation on a single block.

Most block ciphers are iterative functions. There is a simple function, called a *round function*, which provides confusion and diffusion. The round function is called repeatedly, increasing the amount of confusion and diffusion with each iteration. The key is used to generate a *key schedule* – a series of *round keys* that are applied to the state in each round, or in a subset of rounds. We call this application of the round key *key injection*. Intuitively, more rounds means more confusion and diffusion, and therefore, higher security. However, this is not always true, as we will describe later in section 2.2.4.

In this section, we provide a high-level view of block cipher constructions. This is necessary to understand how attacks work and why a cryptanalytic attack on one construction does not necessarily translate to another.

2.1.1 Substitution-Permutation Networks

A substitution-permutation network (SPN) provides confusion with *substitution boxes* (S-boxes). This simple primitive replaces a value with another value, such that the

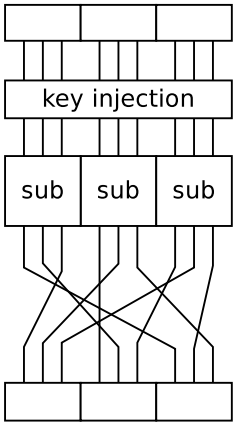


Figure 2-1: Sample SPN Round

relationship between the input and output is difficult to discover. Specifically, these must not have linear properties. Diffusion is provided by permutation. The state is partitioned into smaller pieces, and the pieces are moved to different positions in the state according to the permutation. This allows bits to spread their influence across more bit positions in the state. A simple SPN round is depicted in figure 2-1.

2.1.2 Feistel Ciphers

A Feistel cipher has two defining properties: First, encryption and decryption operations are identical (although the key schedule may be reversed). Second, only a portion of the state is modified in each round. The state is broken into left and right partitions, L_i and R_i, and a round operates on one of these partitions while leaving the other unmodified. At the end of the round function, L_i and R_i are swapped so that the next round modifies the other partition. Typically, Feistel ciphers are balanced (the size of each partition is equal), but a cipher can be unbalanced as well.

A general Feistel round is shown in figure 2-2. The round can be written as $R_i = g(L_{i-1}, f(R_{i-1}))$, $L_i = R_{i-1}$. The key would be used as a parameter for $f(x)$ or $g(x)$.

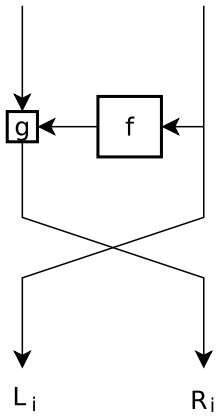


Figure 2-2: General Feistel Round

2.1.3 ARX

Addition-Rotation-Xor (ARX) ciphers are a newer cipher construct that combine addition modulo 2^n ($\boxplus$) and exclusive-or ($\oplus$). The rotation operation is also used to provide diffusion. An ARX cipher is not necessarily distinct from the previous two constructions – it simply describes the operations that comprise the round function. An ARX cipher may be a SPN or Feistel cipher. There are several important properties that arise with this particular combination of operations.

- a round function is comprised of *two* binary operations over the same set, $\mathbb{Z}_2^n$
  - addition forms the group $\mathbb{Z}_{2^n}$
  - exclusive-or forms the group $\mathbb{Z}_2^n$
- addition modulo 2^n and exclusive-or do not distribute, making it difficult to simplify relations (provides confusion)
  - $(a \oplus b) \boxplus c \nRightarrow (a \boxplus c) \oplus (b \boxplus c)$
  - $(a \boxplus b) \oplus c \nRightarrow (a \oplus c) \boxplus (b \oplus c)$
- addition provides diffusion and nonlinearity via the carry function

- rotation provides diffusion by causing bits to rely on bits in different positions

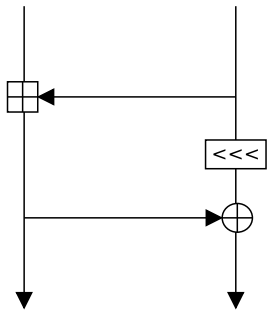


Figure 2-3: Example ARX round

Examples of ARX ciphers include Akelarre [25], RC5 [51], and Threefish [23]. Skein [23], BLAKE [5], CubeHash [7], and Blue Midnight Wish [26] are ARX-based hash functions that were candidates in the second round of the SHA-3 competition. Skein [23] and BLAKE [5] are among the five finalists.

This work focuses on the analysis of ARX functions.

2.2 Previous Cryptanalysis of Block Ciphers

Several classes of attacks on block ciphers have been seen over the years, and all are still relevant. The two most historically significant attacks are *linear cryptanalysis*[40] and *differential cryptanalysis*[8]. Both of these attacks demonstrated serious weaknesses in the Data Encryption Standard (DES)[43], which led to its replacement by 3-DES[44], and later the Advanced Encryption Standard (AES)[45].

A *distinguisher* is a function or method that allows an adversary to determine, with probability significantly distinct from guessing at random, whether a string is the result of encryption/decryption or a random string. Because the security of an encryption algorithm relies on its appearance as a random function, the existence of a feasible distinguisher often implies that the cipher is insecure. A distinguisher or the

combination of a set of distinguishers is the foundation of most cryptanalytic attacks. (Other attacks do exist that build upon properties of algorithm implementation, but these are specific to implementations and generally do not extend to the algorithm itself.)

The significance of a proposed attack is judged by the probability of success, the computational and communication complexities, and the adversarial model. An adversarial model captures the assumed capabilities or abilities of the adversary. For example, the known plaintext model assumes an adversary with knowledge of plaintexts and their corresponding ciphertexts when encrypted with some unknown key, but the adversary can only observe. The chosen plaintext model captures an adversary who has the ability to request the encryption of plaintext of her choice under an unknown key and receive the corresponding ciphertext. The probabilities of success in an attack should be significantly greater than that of a guess made at random, and the computational and communication complexities should be considerably smaller than those of brute force attacks that are oblivious of the structure of the cipher. Further, a weaker adversarial model implies a better attack, because the adversary is less constrained.

2.2.1 Linear Cryptanalysis

Linear cryptanalysis consists of approximating a function using linear expressions. This method was created to attack DES, and as such, was designed for approximating functions using XOR, or addition modulo 2. An approximation of the relationship among plaintext, ciphertext and key bits would be of the form $p_1 \oplus p_2 \oplus \ldots \oplus p_n \oplus c_i \oplus \ldots \oplus c_m = k_1 \oplus \ldots \oplus k_s$, where each p_i is a bit of plaintext, each c_i is a bit of ciphertext, and each k_i is a bit of the key.

For an approximation to be useful, it must be true with a probability significantly distinct from $\frac{1}{2}$, which is the probability with which any linear expression in independent bits would be true. That is, the bias of the expression (difference between this probability and $\frac{1}{2}$) should be significant. Thus the approximation is true with probability $\frac{1}{2} \pm \epsilon$, where $\frac{1}{2}$ is what is expected at random, and ϵ represents the bias. When an equation with a suitably large bias is found, the approximation can be used to determine a portion of the key under the known plaintext model. That is, an adversary who can obtain a set of ciphertexts and corresponding known plaintexts encrypted using a single unknown fixed key can proceed as follows.

1. Find a linear expression that approximates $r - 1$ rounds of the cipher
2. Obtain a collection of (plaintext, ciphertext) pairs encrypted under the same key. The number of pairs needed depends on the bias of the linear expression.
3. For each possible value of the candidate key values:
   (a) Decrypt one round
   (b) Calculate the number of times the equation holds true with the pairs
   (c) The values of the key that produce the results with the greatest bias are most likely to be the true key values

This attack, unlike most others, does not require that plaintexts satisfy a certain distribution or pattern. Any plaintext-ciphertext pairs, chosen uniformly at random, suffice. Therefore it does not require a very strong adversary, and is generally considered a practical and devastating attack.

Extensions

Results can be improved using multiple linear approximations to perform linear cryptanalysis [29]. In particular, Kaliski et al. showed that multiple linear approximation may exist over the same key bits, with different plaintext and ciphertext bits. By using multiple linear expressions using the same key bits, the number of pairs needed to distinguish ciphertext from a random permutation was significantly reduced. From another point of view, the success rate for a fixed number of plaintext-ciphertext pairs increases when multiple expressions are used.

In recent years, the concept of linear cryptanalysis has been generalized to encompass non-binary ciphers [6]. In particular, it was generalized to work over the group $\mathbb{Z}_m^r$, where $m \geq 2$ and $r \geq 1$. The theory also includes the combination of multiple group structures over the same set. For example, it applies to a cipher that combines operations in $\mathbb{Z}_2^8$ and $\mathbb{Z}_{256}$. An example using these groups was presented, but in the analysis the adversary changed the view of the cipher to only use addition modulo 2^8. That is, the additions modulo 2, which appeared only during key injection, were dealt with by changing the adjacent substitution boxes to keyed substitution boxes. Once the exclusive-or operations were absorbed into S-boxes, the view of the cipher no longer contained addition modulo 2 – all additions were modulo 2^8.

2.2.2 Differential Cryptanalysis

Differential cryptanalysis examines pairs of plaintexts with a fixed difference, obtaining a relationship between the corresponding ciphertexts [8]. Consider two plaintexts, x and $x' = x \oplus \Delta_{in}$ (where $\oplus$ represents bitwise XOR), and their encryptions under key k, $y = E_k(x)$ and $y' = E_k(x')$. Let $\Delta_{out} = y \oplus y'$. If there exists an input difference Δ_{in} that leads to an output difference Δ_{out} with high bias, then the encryption

function can be distinguished from a random function. The pair (Δ_{in}, Δ_{out}) is referred to as a *characteristic*, and written ($\Delta_{in} \rightarrow \Delta_{out}$). A pair ($x$, x') that follows the characteristic is called a *right pair*.

This type of attack assumes a stronger adversary than linear cryptanalysis. In order to perform a differential attack, the plaintexts must fit the difference pattern. Therefore, these attacks are typically performed by an adversary that has the ability to select plaintext messages and obtain their encryptions.

Algorithms for finding optimal exclusive-or differences for addition modulo 2^n with complexity less than that of an exhaustive search have been studied [38]. It has also been shown that solving differential equations of addition can be accomplished in polynomial time [50]. Although the original attack description uses exclusive-or as the difference function, other forms of addition could be used as well [39].

2.2.3 Variations of Linear and Differential Cryptanalysis

A relationship between linear and differential cryptanalysis has been shown [17]. In particular, an attack with one method implies an attack with the other because differences tend to propagate linearly. It is therefore important to study both techniques, even if the goal is only to apply one. The following sections describe newer methods that have evolved from linear and differential attacks.

Differential-Linear Cryptanalysis

Linear and differential methods have been combined into a single attack [37], called *Differential-Linear Cryptanalysis*. The attack begins with a differential characteristic of high probability that is used over the first s rounds. After that, a linear distinguisher is used to approximate the remaining t rounds. This results in a distinguisher

over $s + t$ rounds of an iterated encryption function where there was not one possible using only linear or differential methods. This attack was demonstrated on DES.

Truncated Differentials

The main distinction between a differential characteristic and a truncated differential characteristic [34] is the set of bits used to compute the difference. Differential cryptanalysis requires a difference to be satisfied in all n-bits of data (the entire data block), while truncated differentials only require a difference to be satisfied in m bits, where $m \leq n$. It is therefore a generalization of differential cryptanalysis.

Higher Order Differentials

Higher order differentials [34] do not restrict differences to linear relationships. The differentials used in differential cryptanalysis reflect the first order derivative described in [36]. Higher order differentials extend the concept of differential cryptanalysis to allow for i-th order derivatives as well.

Boomerang Attacks

Sometimes several differential characteristics can be found on a function, where each is too short to cover the entire function, but together they do achieve coverage. These shorter differentials can be used to perform a *boomerang attack*. Let E be the entire encryption function, E_0 be a cipher with reduced rounds which has a differential characteristic in the encryption direction. Let E_1 be the remaining rounds of the encryption function with a differential characteristic in the decryption direction. Then write the encryption function as $E = E_0 \circ E_1$.

The idea behind this method is that if an adversary "throws" an input difference at the cipher, the same difference will "come back" with a certain probability.

An abstract view of this type of attack is shown in figure 2-4. The distinguisher begins on the left, with plaintexts P and P' such that $P \oplus P' = \Delta$. Both are encrypted, $C = Encrypt_k(P)$ and $C' = Encrypt_k(P')$. New ciphertexts are derived such that $D = C \oplus \nabla$ and $D' = C' \oplus \nabla$, and are decrypted to produce $Q = Decrypt_k(D)$ and $Q' = Decrypt_k(D')$. The bias is then computed for how often the equation $Q \oplus Q' = \Delta$ holds.

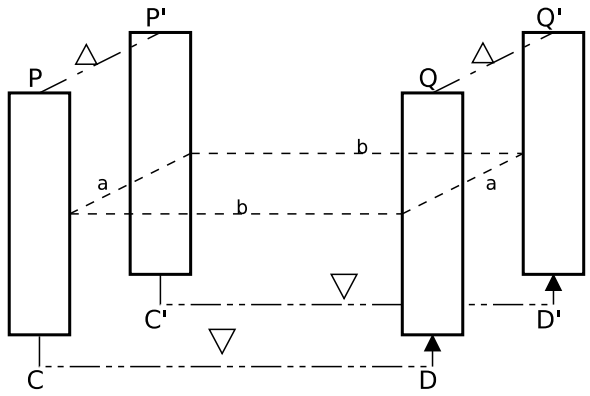


Figure 2-4: Simplified view of a boomerang attack

In figure 2-4, a is the difference between $E_0(P)$ and $E_0(P')$, and b is the difference between $E_1^{-1}(C)$ and $E_1^{-1}(D)$, where E_1^{-1} represents the decryption of E_1.

Related-key boomerang attacks have been applied to Threefish, the block cipher in Skein's [23] compression function [18, 4].

Impossible Differentials

Differential attacks seek out characteristics with higher probability than expected. In contrast, *impossible differentials* are characteristics that *can not* occur. Suppose an adversary has a characteristic $(\Delta_a \rightarrow \Delta_b)$ that occurs with probability 0, and a right pair. When the adversary guesses the relevant key bits and tests the differential properties, any guess that results in the target difference can immediately be eliminated since the correct key value will never produce that difference.

Rotational Cryptanalysis

Rotational cryptanalysis [32] is a new differential technique. It is based on two properties that always hold, independent of the cipher.

$$Pr[\overrightarrow{X \oplus Y} = \overrightarrow{X} \oplus \overrightarrow{Y}] = 1$$

$$Pr[\overrightarrow{X \boxplus Y} = \overrightarrow{X} \boxplus \overrightarrow{Y}] = \frac{1}{4}(1 + 2^{r-n} + 2^{-r} + 2^{-n})$$

where $\overrightarrow{X}$ is a right rotation of n-bit word X by r positions and $\boxplus$ is addition modulo 2^n. A pair $(X, \overrightarrow{X})$ is called a rotational pair for a given r, $r \leq n$. A distinguisher is successful if the rotational property is preserved ($Encrypt(\overrightarrow{X}) = \overrightarrow{Encrypt(X)}$) throughout the cipher with probability significantly different from random.

This attack requires a very strong adversary who has the ability to manipulate key and data pairs. As such, the meaning and applicability of this type attack is currently being evaluated in the cryptologic community. If a block cipher is part of a hash function, where keys are derived from some portion of the state, this could be a valid attack. As an attack on a standalone block cipher, however, the model may not be as valid.

2.2.4 Other Attacks

Mod n Cryptanalysis

Mod n cryptanalysis was first discovered by Kelsey et al., presented in [30]. It is a partitioning attack that exploits weaknesses in ciphers based on addition and rotation. The key observation in [30] is that addition modulo 2^n, where n is the word length in bits, can be partitioned into a smaller word size. By performing addition modulo 3 instead of modulo 2^n, the authors of [30] could distinguish the input to the penultimate

round of a cipher from random.

Consider two n-bit words, a and b. Because addition is preformed modulo 2^n, the sum is described as

$$a + b \bmod 2^n = \begin{cases} a + b & \text{if there was no carry out} \\ a + b - 2^n & \text{if there was a carry out} \end{cases}$$

In [30], the authors followed addition modulo 3 throughout the round functions of a cipher. This has the following properties:

$$(a + b \bmod 2^n) \bmod 3 = \begin{cases} a + b \bmod 3 & \text{if there was no carry out} \\ a + b - 1 \bmod 3 & \text{if there was a carry out} \end{cases}$$

If the adversary has additional knowledge of the carry function, she can approximate a value with high accuracy. An example was presented where the four most significant bits of one of the addends is all ones. In this scenario, $Pr[\text{no carry out}] = 0.02$ and $Pr[\text{carry out}] = 0.98$. Clearly, this additional knowledge leads to a high chance of guessing the correct value of the sum modulo 3.

This technique was applied to modified versions of several ciphers, most notably RC5 [51]. The modified version replaced all exclusive-or operations with addition modulo 2^{32}, and was called RC5P. The attack was performed using the chosen plaintext adversarial model, allowing the adversary to select plaintext values that would maximize the bias after m rounds. The attack could successfully distinguish the correct input from incorrect input with effort much less than required by brute force.

The attack did not work against the original RC5, since exclusive-or is difficult to approximate modulo 3.

This demonstrated that RC5 (and similarly-constructed ciphers) depends on the mixture of addition and exclusive-or for security. A round function created with addition modulo 2^n may be susceptible to mod n cryptanalysis, a round function created with exclusive-or may be susceptible to linear cryptanalysis, but the alternation of the two addition operations appears to prevent both attacks.

Integral Cryptanalysis

Integral cryptanalyis[35] relies on sums of sets of ciphertexts. In this attack, a particular set of plaintexts is encrypted such that a subset of the bits take on all possible values. As an example, consider the attack on reduced-round AES. AES groups the data in a state into bytes. For this attack, the adversary needs to obtain a set of ciphertexts that were generated by a set of plaintexts where all bytes of the message are the same, except one. The byte that differs takes on all 256 values over the set, yielding 256 different ciphertext values.

This attack works quite well for ciphers that break the state into small partitions, such as bytes. It is unclear how this could be efficiently applied to ciphers that use large partitions.

If an adversary is extremely lucky, she might be able to collect a set of plaintext-ciphertext pairs that satisfy this attack. Realistically, however, this attack belongs to a stronger adversarial model where the adversary can choose the plaintexts and obtain their encryptions.

Slide Attacks

Slide attacks take advantage of identical properties in cipher built from repeated iterations of the same round function, $f(x)$ [9]. It relies on the existence of "slid pairs", plaintexts (P, P') and corresponding ciphertexts (C, C'), such that $f(P) = P'$

and $f(C) = C'$. The structure of a slid pair is shown in figure 2-5. When a slid pair is found, it reduces the cipher from multiple iterations of the round function to 1. Round functions are typically weak on their own, and are susceptible to other techniques such as linear and differential cryptanalysis.

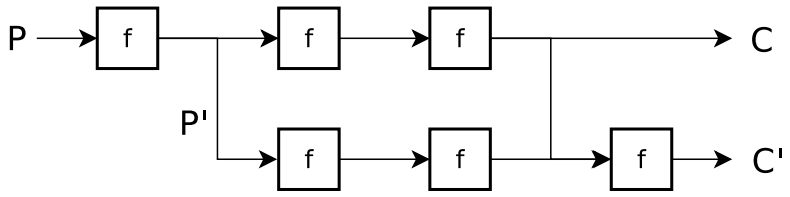


Figure 2-5: Slid pair for three rounds of f

For Feistel ciphers, it is fairly straightforward to find a slid pair. This is because after 1 round, only half of the of the state is modified. Thus if a pair (P, P') is found such that they match in the portion of the state that was not modified, they are likely to be a slid pair. Variations of this method exist to address situations where the adversary must slide over multiple rounds [10], but will not be described here.

These attacks have forced modern cipher designers to design functions such that each iteration is different. The most common way to accomplish this is to add a constant to an operation in each round, such as making it part of the subkey, where the constant acts as a counter. Since the constant is different in each round, no two rounds are exactly the same.

Weak keys

An encryption algorithm does not need a distinguisher or key recovery attack in order to be considered too insecure for use. The existence of *weak keys* in a cipher is enough to dissuade its use. Weak keys are a subset of the keyspace that produce undesirable behavior in the cipher. Often, this occurs in the form of multiple keys generating the same permutation.

To illustrate this concept, let us consider a cipher where half of the keys map message m to ciphertext c_1, and the other half map m to c_2. Let the key size be 128 bits. Then, on knowing the ciphertext corresponding to m, an attacker has a search space 2^{127} possible keys rather than 2^{128} and the cipher provides one less bit of security than it appears without those weak keys.

Weak keys were discovered in Akelarre[25] and TEA [57], among others.

2.2.5 Linear Properties of Addition

Linearization techniques are not limited to block ciphers – they have also been successfully applied to *stream ciphers* and *hash functions*. A stream cipher generates a pseudorandom bit string that is combined with the plaintext via a simple operation, such as exclusive-or. A hash function takes a variable-length message and converts it to a fixed-sized output such that it is extremely difficult to (1) find two messages that map to the same output and (2) find a message that maps to a particular output. Addition and ARX constructs are also found in these functions. As a result of its now widespread use in cryptology, linear properties of addition modulo 2^n have become an interesting area of research. Several properties of addition have been shown and used in cryptanalysis.

Bias of the carry function

Consider $A + B = A \oplus B \oplus carry(A, B)$, where $carry(A, B)$ is a binary vector representing the positions in an addition where a carry occurs. For example, let $A = 2$ and $B = 3$. Then $2+3 = 10_2 \oplus 11_2 \oplus 100_2 = 101_2 = 5$, where 100_2 is the carry vector. Intuitively, it may feel that it is equally likely for any bit in the carry function to be a 1 or a 0 when the two addends are selected at random from a uniform distribution.

However, this is not the case – the carry function is biased [58]. When the two addends are selected at random, the probability of a carry at the n^{th} least significant bit is $\frac{1}{2} + 2^{-n-1}$ for $n \geq 1$ [58]. This bias allowed Wu and Preneel to form linear relationships in the ABCv2 stream cipher that held with high bias when keys had certain properties [58]. Using these relationships, they derived a method to identify weak keys in the stream cipher.

Multiple additions

It is fairly easy to reason about addition and the carry distribution when the addends are selected at random from a uniform distribution. Consider three random addends, x, y, and z, where an approximation of the sum $x \boxplus y \boxplus z$ is desired. The approximation $x \boxplus y$ no longer belongs to the random distribution, so it is difficult to reason about this sum added to z. This is due to a high level of dependence between subsequent additions [49]. When two additions are performed in direct succession, the approximation should consider it as an addition with three inputs rather than two independent additions, each with two inputs [49].

The probability distribution of the carry function in integer addition with an arbitrary number of inputs has been investigated[53]. For n inputs, Staffelbach et al. constructed a $n \times n$ transition matrix for the carry function and derived Eigenvalues and Eigenvectors on the transition matrix. Using this technique, they concluded that asymptotically the carry is balanced for even n and biased for odd n.

Carry distribution of neighboring bits

In the cryptanalysis of the stream cipher NLSv2[19], a very particular instance of the carry function – the carry distribution of two neighboring bits during addition modulo 2^{32} – was studied. In this analysis, several properties are described. First,

let $Carry(x, y)_i$ represent the carry bit out of index i that results from $x \boxplus y$. Then, the carry bit out of index i can be expressed as

$$Carry(x, y)_i = x_i y_i \oplus \sum_{j=0}^{i-1} x_j y_j \prod_{k=j+1}^{i} [x_k \oplus y_k], i = 0, 1, \ldots, 30 \tag{2.1}$$

Given a linear selection mask Γ_i, where the mask is 1 at bits i and $i + 1$ and 0 everywhere else, the equations

$$\Gamma_i \cdot Carry(x, y) = 0 \tag{2.2}$$

and

$$\Gamma_i \cdot (x \boxplus y) = \Gamma_i \cdot (x \oplus y), i = 0, \ldots, 30 \tag{2.3}$$

hold with constant probability $\frac{3}{4}$ [19].

Approximations of addition modulo 2^n

Nyberg and Wallén have made considerable contributions in linear approximations of addition and understanding the carry function [49, 54, 55, 56, 48, 47]. In [49], they provide the following approximation for addition modulo 2^n.

Consider k n-bit integers, denoted $x_1 \ldots x_k$. Then $x_1 \boxplus \ldots \boxplus x_k$ is a function from the vector space $(\mathbb{F}_2^n)^k$ to $\mathbb{F}_2^n$. The addition can now be approximated by inner products and exclusive-or. If $u \in \mathbb{F}_2^n$ and $w_i \in \mathbb{F}_2^n$, $i \in \{1, \ldots k\}$ are mask vectors, then addition can be approximated by

$$u \cdot (x_1 \boxplus \cdots \boxplus x_k) = (x_1, \ldots, x_k) \cdot (w_1 \cdots w_k) \tag{2.4}$$

A method to search for appropriate values of u and w_i was investigated, but was

found to be difficult when addition had three inputs.

2.3 Brief Introduction to Iterated Hash Functions

The purpose of a hash function is to produce a fixed-length representation of a variable-length message. Most well-known secure hash functions in use today are *iterated hash functions*. The heart of any iterated hash algorithm is its *compression function*, which comprises two parts: (1) a method of inserting a message block into the state, and (2) a pseudorandom permutation. The compression function takes at least two values – a message block and a chaining value (hash state) [42]. There may be more input arguments depending on the compression function. This function is called for each message block, and thus the resulting secure hash function is an iterative function.

The operation of a generalized iterated hash function can be broken down into the following steps:

initialization This is the process that produces the first state value of the compression function, H_0. This depends only on the algorithm in use, and is independent of the message. We also refer to H_0 as the *initialization vector*, or IV.

preprocessing Various preprocessing transformations may be applied to the message prior to hashing. These include padding the message to be a multiple of b bits, where b is the block size in bits, and splitting the resulting message into k b-bit blocks.

message processing This stage makes repeated calls to the compression function – one for each message block. The compression function takes different input parameters, depending on how the hash function was constructed. At a minimum,

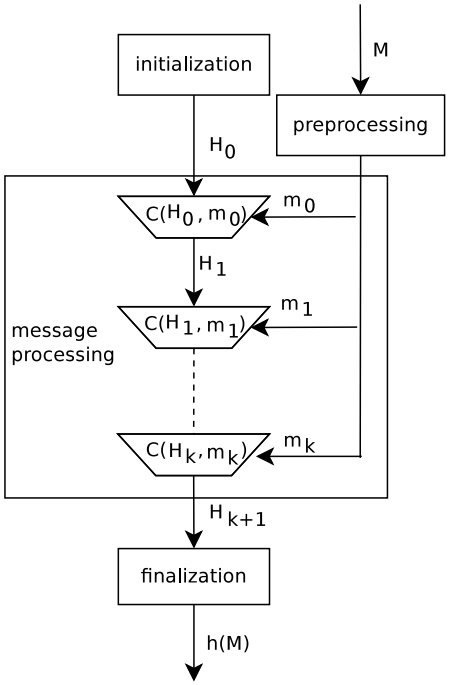


Figure 2-6: A generalized iterated hash function

it takes a chaining value and a message block as input.

finalization After the message has been processed, an additional function applied, known as finalization. Note that this may be simply be an identity permutation.

2.3.1 Notation

H_i – chaining value after i calls to the compression function

$h(x)$ – a call to the hash function

$c(H_i, x)$ – a call to the compression function

M – a message of any length

m_i – the i^{th} block of message M

2.4 Previous Cryptanalysis of Hash Functions

Discussion up to this point has been focused on the analysis of ARX-based compression functions, and not the hash function as a whole. In this section, we will briefly discuss attacks on general hash functions. Note that an attack on the compression function of an iterated hash function can be extended to an attack on the hash function [42].

2.4.1 Preimage

A hash function should be one-way. That is, it should be computationally infeasible for an adversary to find a message M, given a hash value y, such that $H(M) = y$. NIST requires that candidates for SHA-3 provide n bits of preimage resistance for an n-bit hash [46]. That is, that the most efficient attack should require $O(2^n)$ computations of the hash function.

Preimage attacks on the first round version of SHA-3 entry CubeHash can be derived due to a symmetry property in the compression function [2]. Symmetry occurs when a property of the input also holds in the output. In CubeHash, the symmetric property is equality between 32-bit words.

Several preimage attacks have been demonstrated on weakened variants of CubeHash [33, 12]. Preimage attacks that exploit properties of CubeHash's round function (with the parameters recommended in round 1 of the competition) were presented in [3]. A statistical approach for preimage searching has also been applied [11].

2.4.2 Second Preimage

As the name implies, a second preimage attack is concerned with finding a second message that hashes to a particular value. The adversary knows the first message M,

and the resulting hash, $h(M)$. If the adversary can find a message $M', M' \neq M$, such that $h(M') = h(M)$, then they have found a second preimage.

2.4.3 Collision

A collision occurs when two messages, M and M', hash to the same value. These are unavoidable due to the pigeon-hole principle, but it should be computationally infeasible to find such collisions. In particular, the probability of achieving a collision by any means should be upper bounded by the birthday paradox: for an n-bit digest, $2^{\frac{n}{2}}$ calls to the compression function should result in a probability of roughly $\frac{1}{2}$ of finding a collision.

Several collisions have been found on reduced-versions of CubeHash [12, 21, 15, 14, 1, 16].

A linearization framework for finding collisions has also been explored [13] . This framework showed that finding a collision can be redefined as finding preimages of zero in a special function called a condition function. The condition function examines two words, A and B, and their carry word under addition modulo 2^n, C, and returns a vector of length $(n-1)(\#\ of\ additions)$.

A rotational rebound collision attack was performed on Skein [31]. This lead to the strongest cryptanalysis of Skein and Threefish to date, but did not threaten the security of the hash function. The strength of the attack was greatly reduced by changing a constant in the Threefish key schedule.

Multicollision

A multicollision [28] is a special type of collision where more than two messages will yield the same digest. Joux [28] showed that these are efficient to compute on an

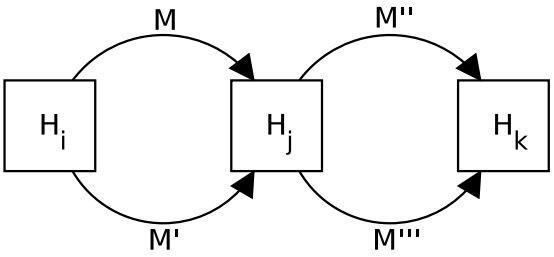


Figure 2-7: Multicollision

iterated hash function. From a state H_i, find two messages that yield state H_j (note that the two messages can have different size). Figure 2-7 shows a 4-collision, since there are four different ways to get from H_i to H_k.

Multicollisions that exploit fixed points of the CubeHash round function were presented in [3].

Near-collision

A near-collision occurs when $h(M) \approx h(M')$. The distance between the two digests is small, and is typically measured in Hamming distance. Near-collisions have been found on variants of Skein [23] and BLAKE [5] where the compression function was both linearized and the number of rounds reduced.

Pseudo-collision

A pseudo-collision (or free-start collision) is an attack on the compression function where the adversary can control both chaining values and both messages [42]. That is, the adversary is free to choose (H_a, m_a) and (H_b, m_b) such that the $c(H_a, m_a) = c(H_b, m_b)$. This type of attack is applied to the compression function, and then later extended to the hash function.

To consider this scenario in terms of attacks presented earlier in section 2.2, the idea is similar to a differential. In this case, the attacker can fix differences in both keys

and both plaintexts in an attempt to arrive at a zero difference after the compression function.

Semi-free-start collision

A semi-free-start collision on a compression function is a stronger attack than a pseudo-collision, because it removes some of the adversary's control. In particular, the adversary no longer has the ability to modify the chaining value. A semi-free-start collision occurs when $c(H_a, m_a) = c(H_a, m_b)$, where $m_a \neq m_b$.

This scenario is also similar to a differential attack, where the adversary can only control differences in either the keys *or* plaintexts, but not both.

Chapter 3 – Linear Approximation in ARX

This chapter introduces a new technique called *pseudo-linear cryptanalysis* that extends linear cryptanalysis to analyze ARX ciphers. In the next chapter we describe preliminary results obtained through its use in analyzing Threefish-256. For the dissertation, I will analyze Threefish-512.

Linear cryptanalysis traditionally results in linear equations in $GF(2)$: that is, in equations over bits. If one uses this approach to analyze addition over larger fields, such as GF(2^n), the carry introduces a large degree of non-linearity so that linear cryptanalysis is no longer a valid analytical tool.

In order to approximate additions of large strings modulo 2^n, we consider larger groupings of contiguous bits, which we call *windows*. Consider the set G of all possible windows of size w—that is, all 2^w possible bits-strings of length w. Our cryptanalysis requires the following two operations on G: bitwise exclusive-or and addition modulo 2^w. We refer to G, with the two operations, as a *pseudo-linear system*. The two operations do not distribute, and hence G, with the two operations, is not a ring.

We approximate addition modulo 2^n over windows of length $w < n$ with addition modulo 2^w. Our approximation is correct if and only if there is no carry into the bits in the window from bits preceding it. Our approach is useful for two reasons: (1) the carry has the same effect regardless of w, (2) its probability, and hence the probability of correctness of our approximation, is independent of w, and (3) the success probability of a random guess decreases as w increases. Thus the bias of our approximations (the difference between the probability of correctness of the approximation and that of a random guess) increases with w.

To illustrate why this is an improvement over traditional linear cryptanalysis, consider the following example depicted in figure 3-1. There are two n-bit words,

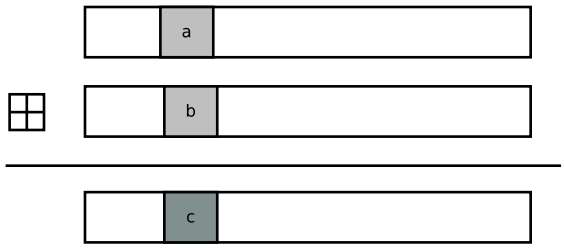


Figure 3-1: Window addition

added modulo 2^n. Suppose only the value of the dark square, labeled c, is needed. a and b are the operand windows in the same position. The square contains w bits, and c can be approximated as $a + b \bmod 2^w$. Whether the approximation is correct or not will depend on the value of the carry into the square. The probability that the carry is zero—and the approximation is correct—is approximately $\frac{1}{2}$, and approximately equal to the probability that the carry is one and the approximation is incorrect. If $w = 1$, the probability of guessing c correctly at random is also $\frac{1}{2}$ and there is no benefit to approximating c by $a \boxplus_w b$. On the other hand, if w is large, the probability of guessing the value of c correctly at random is $\frac{1}{2^w}$, and the approximation, with a correctness probability of $\frac{1}{2}$, provides considerable advantage. *Thus, by looking at windows of several bits rather than a single bit, the effect of the carry is diminished, and one is able to approximate several bits with high bias.*

Any window in an ARX function can be approximated using pseudo-linear equations. We refer to such approximations as pseudo-linear approximations.

3.1 Notation

The following notation is used in this dissertation.

$\oplus$ Bitwise exclusive-or

$\boxplus$ Addition modulo 2^n

$\boxplus_i$ Addition modulo 2^i

$\boxminus$ Subtraction modulo 2^n

$\boxminus_i$ Subtraction modulo 2^i

$\lll$ **(**$\ggg$**)** Left (right) rotation on an n-bit word

3.2 Addition Windows: Simple Analytical Results

Several properties are useful for predicting carries. We start by assuming that the n-bit words that are added come from a uniformly random distribution. Throughout this paper, words are represented with the most significant bit at position $n-1$ and the least significant bit at position 0.

Consider two n-bit words, a and b, selected uniformly at random, and a window size w. Let $part(a, s, e)$ be a function which returns the bits of a in the range $[s,\ e)$, where the range $[0,w)$ represents the least significant w bits and $s < e \leq n$. Then

Lemma 1. *The equation* $part(\mathrm{a} \boxplus \mathrm{b}, \mathrm{i}, \mathrm{i}+\mathrm{w}) = part(\mathrm{a}, \mathrm{i}, \mathrm{i}+\mathrm{w}) \boxplus_w part(\mathrm{b}, \mathrm{i}, \mathrm{i}+\mathrm{w})$ *holds with probability greater than* $\frac{1}{2}$.

Proof. If there is no carry into bit i, then the equation is true. If there is a carry into bit i, then the equation is false. For the addition of two i-bit integers, there are $2^{2i-1} + 2^{i-1}$ of 2^{2i} operand combinations that will not produce a carry in the output. Therefore the probability that the carry into bit i is 0 is $\frac{1}{2} + 2^{-i-1}$, which agrees with the bias presented in [58].

□

Corollary 1. $part(\mathrm{a} \boxplus \mathrm{b}, 0, \mathrm{w}) = part(\mathrm{a}, 0, \mathrm{w}) \boxplus_w part(\mathrm{b}, 0, \mathrm{w})$ *with probability 1.*

Proof. Because this is the least significant window, there are no carry bits into the sum. Therefore the probability of the equation being true is 1. □

Corollary 2. $part(\mathrm{a} \boxplus \mathrm{b}, \mathrm{i}, (\mathrm{i}+\mathrm{w}) \bmod \mathrm{n}) = part(\mathrm{a}, \mathrm{i}, (\mathrm{i}+\mathrm{w}) \bmod \mathrm{n}) \boxplus_w part(\mathrm{b}, \mathrm{i}, (\mathrm{i}+\mathrm{w}) \bmod \mathrm{n})$ *with probability greater than* $\frac{1}{2}$.

Proof. The key difference in this case is that the window may wrap around from the higher end of the n-bit word to the lower end. If the window does not wrap around, this is equivalent to lemma 1. If it does, then there is one carry that is not propagated; namely the carry out of the n-bit sum. To accomplish this, we split the window in two: $part(a, i, n) \boxplus_w part(b, i, n)$ and $part(a, 0, (i+w) \bmod n) \boxplus_w part(b, 0, (i+w) \bmod n)$. The first sum is true by lemma 1 and the second is true by corollary 1. □

Corollary 3. $part(\mathrm{a} \boxminus \mathrm{b}, \mathrm{i}, (\mathrm{i}+\mathrm{w}) \bmod \mathrm{n}) = part(\mathrm{a}, \mathrm{i}, (\mathrm{i}+\mathrm{w}) \bmod \mathrm{n}) \boxminus_w part(\mathrm{b}, \mathrm{i}, (\mathrm{i}+\mathrm{w}) \bmod \mathrm{n})$ *with probability greater than* $\frac{1}{2}$.

Proof. Let $a \boxminus b = c$. Then $b \boxplus c = a$, and $part(b \boxplus c, i, (i+w) \bmod n) = part(b, i, (i+w) \bmod n) \boxplus_w part(c, i, (i+w) \bmod n)$ with probability greater than $\frac{1}{2}$ by lemma 1 and corollary 2. Then we have $part(a, i, (i+w) \bmod n) = part(b, i, (i+w) \bmod n) \boxplus part(c, i, (i+w) \bmod n)$, which is the same as $part(a, i, (i+w) \bmod n) \boxminus part(b, i, (i+w) \bmod n) = part(c, i, (i+w) \bmod n)$, with probability greater than $\frac{1}{2}$. □

Lemma 2. $part(\mathrm{a} \oplus \mathrm{b}, \mathrm{i}, (\mathrm{i}+\mathrm{w}) \bmod \mathrm{n}) = part(\mathrm{a}, \mathrm{i}, (\mathrm{i}+\mathrm{w}) \bmod \mathrm{n}) \oplus part(\mathrm{b}, \mathrm{i}, (\mathrm{i}+\mathrm{w}) \bmod \mathrm{n})$ *with probability 1.*

Proof. This follows directly from the bitwise nature of exclusive-or. □

Rotations occur within a single word, and are easy to express as windows by shifting the indices. For example, $part(a \ggg c, i, (i+w) \bmod n) = part(a, (c+i) \bmod n, (c+i+w) \bmod n)$.

The addition window properties described above work quite well for window approximations where the operands are from a *uniformly random* distribution. Once an addition on this data is performed, the distribution is no longer random. It is easy to see why this is so. Consider four k-bit words sampled from a uniform distribution: a, b, a', b'. Let $c = a + b \bmod 2^k$, similarly c'. If there is a carry out of $a + b$, then c is biased towards the smaller values from 0 to $2^k - 1$. Similarly, if there is no carry out of $a + b$, c is biased towards the larger values from 0 to $2^k - 1$. For example, the maximum possible value of the sum of two k-bit integers is $2^{k+1} - 2$, and hence the sum modulo 2^k cannot achieve the value $2^k - 1$ at all if there is a carry out of the sum. Thus the carry out of $a + b$ and $a' + b'$ characterizes the distribution of c and c', and hence of the carry out of their sum.

As an example, consider two 4-bit words, a and b, and sum $c = a \boxplus b$. Let $a(i)$ be the 2-bit window with i denoting the position of the least significant bit. Suppose $w = 2$ and the goal is to approximate $c(2)$ (the approximation window has the third bit as its least significant bit). The carry into this addition window is determined entirely by the values in $a(0)$ and $b(0)$. There are 16 possible pairs of operands, 6 of which produce a carry and 10 which do not. If there was no carry into $c(2)$, then $Pr[c(0) = 0] = \frac{1}{10}$, $Pr[c(0) = 1] = \frac{2}{10}$, $Pr[c(0) = 2] = \frac{3}{10}$, and $Pr[c(0) = 3] = \frac{4}{10}$. If there was a carry into $c(2)$, then $Pr[c(0) = 0] = \frac{3}{6}$, $Pr[c(0) = 1] = \frac{2}{6}$, $Pr[c(0) = 2] = \frac{1}{6}$, and $Pr[c(0) = 3] = 0$.

Now consider $e = c \boxplus d$, where d is the sum of two other 4-bit values chosen at random. If $c(2) = a(2) \boxplus_2 b(2)$ (there was no carry), then there is a greater chance that $e(2) = c(2) \boxplus_2 d(2) \boxplus_2 1$. If $c(2) = a(2) \boxplus_2 b(2) \boxplus_2 1$ (there was a carry), then there is a greater chance that $e(2) = c(2) \boxplus_2 d(2)$. In the next chapter, we describe how our approach may be used to analyze the SHA-3 candidate Skein.

3.3 Approximations of ARX Round Functions

3.3.1 Base Approximations

Using the windowing method described above, it is straightforward to create an approximation for an ARX round function by following the windows. An approximation that simply follows the windows is equivalent to assuming that the carry into all windows is 0. We refer to this approximation as the *base approximation*.

Although the carry is biased, one still expects a carry for approximately half of the window additions. That is, every other round of approximated window additions, one expects a carry into the window. To allow for a carry approximately every other round of approximation, we use *carry patterns* and *offsets*.

3.3.2 Carry Patterns

A carry pattern is a series of carry values, $c_i \in \{0, 1\}$, where each i denotes an approximated addition window that may have a carry into it. Stated differently, there is a c_i for every approximated addition window that does not start with the least significant bit of the word. One can construct multiple carry patterns that overlay the base approximation such that 1 is added to approximated addition window i if $c_i = 1$.

Let $C^j = (c_0 \dots c_{m-1})$ represent the carry pattern for m approximated addition windows. Then each base approximation overlaid with a distinct carry pattern, $base + C^j$, represents a distinct approximation. By applying several carry patterns to the same input/output pairs, one can increase the probability of a correct approximation, which in turn increases the bias.

3.3.3 Offsets

Another method of compensating for mis-predicted carries is through the use of *offsets*. Consider an integer *offset*$\in \{0, \cdots, 2^{w-1} - 1\}$ that is added to and subtracted from an approximated window value, *approx*. If every valid value of *offset* is tried and the number of observed correct guesses are plotted, the result is a bell-shaped curve with a peak at *offset* $= 0$, like the one shown in figure 3-2 [1]. The horizontal line shows the probability of guessing correctly at random (in this case, $\frac{1}{256}$).

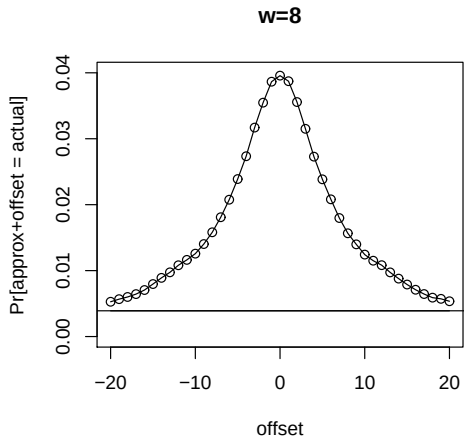


Figure 3-2: An example: observed probability of correctly guessing window of ARX target over several offsets. Plot created with window isolation (section 3.3.8), the base approximation presented in section 6.1.1, and $w = 8$.

Instead of considering an approximation correct only if it matches the target window, we consider it correct if it falls within the range *approx* $\pm$ *offset*. In figure 3-2, the approximation is correct with *offset* $= 1$ with probability $0.039571 + 0.038652 + 0.038726 = 0.116949$. This will increase the number of correct guesses even though carries were guessed incorrectly.

Using this technique improves our results with lower overhead than the application

[1] The results in figure 3-2 were obtained from approximating $x(0)$ in the first word of the 8^{th} round of Threefish-256 with the base approximation. Further details in section 6.

of more carry patterns. However, carry patterns are still important. In particular, we observe that the carry pattern used determines the height of the curve, so a carefully chosen pattern improves the bias for approximations when the target falls in the range $approx \pm offset$. This is because guessing the carry pattern correctly is not equivalent to guessing a correct number to add at the end after the base case. (This number could be, for example, the sum of all the carries that were ignored.) Rotation changes the impact of a single carry, and the value of the carry affects the value of several later additions. Thus guessing the carry correctly each time is superior to guessing the effective sum of all the carries.

Empirical examples with Threefish-256 are presented in section 6.

3.3.4 Computing Bias

The improvements outlined above increase the probability of the approximation being correct, but also increase the probability of guessing correctly at random. That is, a random guess with cp different carry patterns will be correct with probability $\frac{cp}{2^w}$ instead of $\frac{1}{2^w}$ since each pattern represents a different approximation. Thus to get the bias, compute $bias = times\ correct - \frac{\#\ patterns}{2^w} \times pairs$. Similarly, when offsets are used, the bias is computed as $times\ correct - \frac{((2 \times offset)+1)}{2^w} \times pairs$. If both are used, $bias = times\ correct - \frac{\#\ patterns \times ((2 \times offset)+1)}{2^w} \times pairs$.

3.3.5 Flexibility

The techniques described above lead to flexible approximations. Once a base approximation has been established, it can easily be modified through the use of offsets and carry patterns. It is also trivial to change the window size, since it is the start of the window that is important. The end of the window can be computed dynamically

based on the desired w.

3.3.6 Comparison to Linear Cryptanalysis

Though this attack is clearly inspired by linear cryptanalysis, there is a difference that should be noted. Currently, we cannot concatenate and simplify the effect of several approximations. This is because the two operations — exclusive-or and addition modulo 2^w — do not commute. Because of this, we cannot combine all key bits into a single function of the key. With linear cryptanalysis, on the other hand, the use of a single operation — exclusive-or — allows all key bits to be combined into a single function of key bits. Our approximation requires us to try all key bits used in the approximation. Even so, we are able to obtain attacks more efficient than brute force because our approximations enable us to reduce the number of key bits from those required by the cipher. We can approximate one operation using the other, but this only works reasonably well when w is small, which is precisely when pseudo-linear approximations provide the weakest assistance.

3.3.7 Approximation in Fixed Permutations

Several SHA-3 algorithm submissions do not use keyed permutations, but rather use fixed permutations. In functions of this type, their is no key to find. However, pseudo-linear approximation can still be used to perform attacks on hash functions, such as finding collisions and preimages.

For example, consider the scenario in figure 3-3, where the compression function comprises message mixing via exclusive-or and repeated applications of an ARX round function. Given a state S and state T, one can use pseudo-linear approximations to find a message block, M_1, such that a transition from S to T is found.

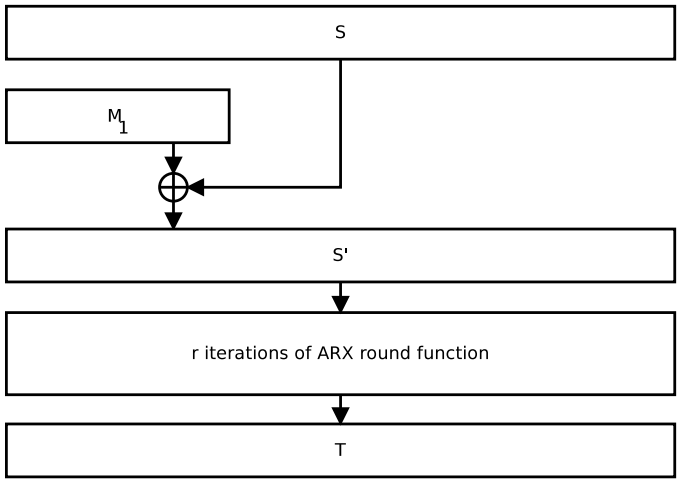


Figure 3-3: Transitions between states with simple ARX compression function

Finding paths such as these can be used to build attacks. Suppose that S is the initial state, and T some other internal state. Then one can find a preimage if there is a path from T to S in the backwards direction. Depending on the particular compression function, this may be accomplished with reduced effort using pseudo-linear analysis. If a path exists, it can be found with less effort than iterating over all possible M_1. If there is no such path between these states, then hopefully a contradiction in all possible solutions will be found before all possible values have been iterated. An example of such a contradiction would be if two different equations depend upon the same window, but they each require it to have a different value. If such a contradiction were reached, another (possibly random) message block should be selected, and the new state should be called T. In the worst case, all possibilities for M_1 will be tried.

Similarly, this approach may be used to find collisions. Pseudo-collisions are the simplest to derive, since they allow us to choose the chaining values. Again using the example in figure 3-3, a pseudo-collision may be achieved if from two different states for S, S_1 and S_2, there is a path $S_1 \rightarrow T_1$ and $S_2 \rightarrow T_2$, where T_1 and T_2 have

the same value for the bits which are not altered by message injection. Then choose message blocks M_2 and M_2' such that $T_1 \oplus M_2 = T' = T_2 \oplus M_2'$. At this point, we have a collision. This attack can use pseudo-linear analysis to find message blocks M_1 and M_1' that satisfy the paths $S_1 \rightarrow T_1$ and $S_2 \rightarrow T_2$, or determine that no such path exists.

In general, we may use pseudo-linear methods to find message blocks that satisfy particular paths. From these paths, we can build collision and preimage attacks. An example of using pseudo-linear analysis in a preimage attack is presented in section 7.5.2.

3.3.8 A Note on Implementation

When implementing pseudo-linear approximations over an ARX function, there are two methods that we have used.

1. Each window stored in its own word (window isolation)

2. All windows for a word stored in the same word (word isolation)

We find that the first is useful for developing approximations, but the latter is much better in practice. This is because the first case provides a way to isolate windows that simplifies the debugging process. However, when windows are adjacent or overlap, there is carry information that is not used due to the high level of isolation. It also becomes possible for a single bit to take on multiple values in multiple windows, which is clearly not correct and will reduce the bias. In the second method, we no longer have bits that take on multiple values, and take advantage of the extra carry information to obtain stronger biases.

The first approach also has another drawback. Because there is an addition performed for each addition window, the number of additions increases rapidly with each

round. The second method uses at most twice the number of additions (1 for word addition, 1 for carry addition).

It is also essential that window position is preserved. When an addition is performed on windows that wrap around a word, it is important that the carry out of position 63 is *not* propagated. Both implementation methods respect this. We stored windows in 64-bit words (regardless of isolation) in the following manner. The bits in a window were set to their correct value in the corresponding position, and all other bits set to 0. For example, $x(4)$ =0x2E would be represented as 00000000 00002E00. With word isolation and two 16-bit windows, $x(8) = 0x5678$ and $x(40) = 0x1234$, the word would be represented as 00123400 00567800.

When using word isolation, the inactive bits may all be set to zero after each addition. Alternatively, they may remain after all operations, and be used throughout the cipher. The latter may be difficult to implement correctly depending on the method of deciding whether or not to add 1 when a carry is guessed. We have found the first approach to be more beneficial overall, although there were drops in bias at certain w values.

Chapter 4 – Truncated Differentials

Recall that there are two prominent classes of cryptanalysis – linear and differential cryptanalysis. In linear cryptanalysis, linear approximations of the encryption function are formed using bits of the input, output and key. Differential cryptanalysis, on the other hand, uses properties of pairs of inputs. In particular, differential cryptanalysis uses the probability that a specific difference at the input of a function produces a certain difference at the output of the function.

A typical differential attack relies on *differential characteristics*, which are sequences of input and output differences over one or multiple rounds. A *differential trail* can be created by chaining characterstics to cover more rounds. To perform this attack, an adversary derives a trail over $r - 1$ of r rounds of the cipher. The adversary then chooses plaintext messages that conform to the input difference, and obtains the output. Next, the adversary guesses some portion of the key, decrypts one round using that key, and checks to see if the appropriate difference holds. The guess that yields the target difference with observed probability closest to the expected probability is most likely the correct guess.

Pseudo-linear approximations can be used to discover differential trails – particularly truncated differential trails. Unlike a traditional differential which uses the entire state, a truncated differential works with part of the state. A common practice in differential calculation is to linearize the function (replace addition/subtraction modulo 2^n with exclusive-or). We propose to eliminate the linearization step and instead follow the differentials in windows.

Let x and x' be two n-bit words, $x' \oplus x = \delta$, $x > x'$. w is the window size in bits. Then we have the following special case of corollary 3.

Corollary 4. $part(x, s, (s + w) \bmod n) \boxminus_w part(x', s, (s + w) \bmod n) = part(x -$

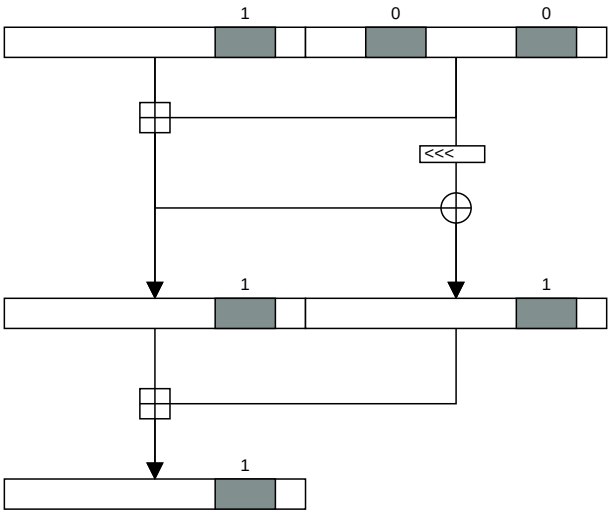


Figure 4-1: Simple view of tracing differences

$x', s, (s + w) \bmod n)$ *with probability* $> \frac{1}{2}$.

Proof. This follows directly from corollary 3. □

Figure 4-1 depicts a simple ARX function with a 1-bit difference in one of the words. The windows in the trace are highlighted in grey, with the difference labeled on top. In this example, the left word contains a 1-bit difference inside a window. There are no differences in the windows on the right. After one application of addition, rotation, and exclusive-or, there is a 1-bit difference in the left window and in the right window, with some probability. After another addition, there is still a 1-bit difference, with some probability.

The differences can be traced from word to word throughout the function. How a difference in the input affects each window in the trace depends on the location of the differences relative to the windows. That is, differences inside the windows affect the target difference, whereas differences outside the windows affect the target difference in varying ways. In particular, a difference occuring in less significant bits than a window has less impact as the distance between the start of the window and

the difference increases. This is is because the probability of the difference introduced by the carry function propagating to the window decreases as the distance between the difference and the window increases.

Consider the example in figure 4-1, but allow the possibility that there are differences outside the window which are possibly unknown. If the input difference of the left word is 1111_2, the difference of the addition will propagate differently than than if the input difference is 1000_2. If the window begins at the fourth least significant bit, then both result in a difference *inside* the window of 1, but the differences *before* the window produce different carry distributions, thereby affecting the difference propagation inside the window. If the window begins much farther away from other differences, then the impact will be less. For example, if the window began in position 32, and the difference outside the window only occured in the least significant bit, then the probabilty of that difference affecting the window after one addition is much less because the probability of the carry function reaching that far is much lower.

4.1 Difference in Addition

4.1.1 In one addend

Consider two n-bit words, a and a', $a' = a \oplus 1$, and a third word b. Let $c = a \boxplus b$ and $c' = a' \boxplus b$.

Since $a \oplus a'$ is odd, $c \oplus c'$ must be odd as well. Let a be even and a' be odd. Then $a \boxplus b$ will not cause a carry out of position 0 regardless of the value of b, causing c to be even if and only if b is even.

$a' \boxplus b$ may have a carry out of position 0. If the least significant bit of b (denote this as b_0) is 0, then $c'_0 = 1$. If $b_0 = 1$, then $c'_0 = 0$. Assuming that b is selected from

the uniform distribution at random, then $pr[c'_0 = 0] = \frac{1}{2}$. If $b_0 = 1$, then $carry_0 = 1$ and $c'_1 = \overline{c_1}$. Similarly, if $carry_1 = 1$ then $c'_2 = \overline{c_2}$. This continues until $carry_i = 0$. All of the other bits are the same in both x_0 and x'_0 for bits $i + 1, \ldots, n - 1$, and therefore produce the same values going forward.

This can be generalized for a difference starting in any bit position.

If the input difference has a run of multiple ones (e.g. 3, 7, F, but not 1), some of the ones may turn to zeros in the output difference. This occurs when one input contains the difference string in the appropriate position, and the other contains zeros in that position. For example, let $a = 1111_2$ and $a' = 1100_2$, giving $a \oplus a' = 11_2$. Let $b = 1101_2$.

$$\begin{array}{rrrrr} & 1 & 1 & 1 & 1 \\ \boxplus & 1 & 1 & 0 & 1 \\ \hline & 1 & 1 & 0 & 0 \end{array} \qquad \begin{array}{rrrrr} & 1 & 1 & 0 & 0 \\ \boxplus & 1 & 1 & 0 & 1 \\ \hline & 1 & 0 & 0 & 1 \end{array}$$

For $a \boxplus b$, carries result from positions 0, 1, and 2. For $a' \boxplus b$, carries result from position 2 only. The resulting difference characteristic here is $\Delta = 0011_2 \rightarrow \delta = 0101_2$, and the string of ones is broken.

4.1.2 In both addends

It may be the case that both addends have difference patterns. For example, let $a' = a \oplus \Delta$, and $b' = b \oplus \Gamma$. Addition is performed as $c = a \boxplus b$, and $c' = a' \boxplus b'$, and $c \oplus c' = \delta$ for some value δ.

While the structure of the output difference is going to vary greatly by input differences, we can easily say something about the least significant bit of the difference pattern.

Consider some difference pattern `000XXX000` in Δ and Γ where `X` marks a possibly nonzero difference. Since there is no difference coming into the bits marked by `X`, the carry into that region the same way. Therefore we eliminate the zero regions from consideration and only consider `XXX`. We can say something about the least significant bit of `XXX` based on the least significant bit of the input difference patterns. In particular, we can use the following rules of addition:

1. odd + odd = even
2. even + even = even
3. odd + even = odd

Lemma 3. *If both inputs have an even difference pattern, then the sum will have an even difference pattern.*

Proof. Let the function $lsb(x)$ return the least significant bit of the difference window. Since the difference in both input windows is even, $lsb(a) = lsb(a')$ and $lsb(b) = lsb(b')$. The possible combinations and resulting $lsb(c)$ and $lsb(c')$ are:

$lsb(a)$	$lsb(b)$	$lsb(c)$	$lsb(a')$	$lsb(b')$	$lsb(c')$	$lsb(c) \oplus lsb(c')$
0	0	0	0	0	0	0
0	1	1	0	1	1	0
1	0	1	1	0	1	0
1	1	0	1	1	0	0

All combinations produce an even difference in the difference window of $c \oplus c'$. □

Lemma 4. *If both inputs have an odd difference pattern, then the sum will have an even difference pattern.*

Proof. The difference in both input windows is odd, so we have $lsb(a) = \overline{lsb(a')}$ and $lsb(b) = \overline{lsb(b')}$. The possible combinations and resulting $lsb(c)$ and $lsb(c')$ are:

$lsb(a)$	$lsb(b)$	$lsb(c)$	$lsb(a')$	$lsb(b')$	$lsb(c')$	$lsb(c) \oplus lsb(c')$
0	0	0	1	1	0	0
0	1	1	1	0	1	0
1	0	1	0	1	1	0
1	1	0	0	0	0	0

All combinations produce an even difference in the difference window of $c \oplus c'$. □

Lemma 5. *If one input has an odd difference pattern and the other an even difference pattern, then the sum will have an odd difference pattern.*

Proof. Without loss of generality suppose $lsb(a) \oplus lsb(a') = 0$ and $lsb(b) \oplus lsb(b') = 1$. The difference in the first addend is even so we have $lsb(a) = lsb(a')$. The difference in the second addend is odd, so $lsb(b) = \overline{lsb(b')}$. The possible combinations and resulting $lsb(c)$ and $lsb(c')$ are:

$lsb(a)$	$lsb(b)$	$lsb(c)$	$lsb(a')$	$lsb(b')$	$lsb(c')$	$lsb(c) \oplus lsb(c')$
0	0	0	0	1	1	1
0	1	1	0	0	0	1
1	0	1	1	1	0	1
1	1	0	1	0	1	1

All combinations produce an odd difference in the difference window of $c \oplus c'$. □

4.2 Difference in Exclusive-Or

When a difference occurs in only one of the operands, that difference is preserved by exclusive-or. Consider n-bit words a, $a' = a \oplus \Delta$, and b, and let $c = a \oplus b$ and

$c' = a' \oplus b$. Then $a'_0 \oplus b_0 = a_0 \oplus \Delta_0 \oplus b_0 \rightarrow c'_0 \oplus c_0 = \Delta_0$. This can be shown for all i in c_i, c'_i.

If both operands have differences, then the output difference may preserve the value of one of the operands. Let Δ_a be the difference in operand a, and Δ_b be the difference in operand b. Without loss of generality, the value b is the output difference when $b \oplus \Delta_b = \Delta_a$. This is because $(a \oplus b) \oplus (a \oplus \Delta_a \oplus b \oplus \Delta_b)$ becomes $a \oplus a \oplus \Delta_a \oplus \Delta_a \oplus b$, which is b.

4.3 Low-Weight Differences

The simplest case to analyze is low Hamming weight differences. Consider $x' = x \oplus 2^i, i \in \{0, \ldots, n-1\}$.

The difference characteristics between rounds can be traced by following the effect of the carry. The input difference will affect the differential in varying ways depending on its position. For example, cryptanalysts have placed the difference in the most significant bit of a word, since this minimizes the effect of the carry. This is because during an addition operation, no combination of addends will produce a carry out of that bit due to modular addition. In other words, addition of the most significant bit of a word behaves like exclusive-or in addition modulo 2^n. However, this practice becomes ineffective after some time as the rotation will move the difference to a position where the carry will matter.

4.4 Windows and Truncated Differentials

A differential trail, truncated or otherwise, should cover $r - 1$ of r rounds. The adversary chooses inputs P and $P \oplus \Delta$, and receives their encryptions C and C'.

Next, the adversary guesses portions of the key that are needed to compute the target window (where the target difference occurs), and backs up one round to obtain T and T'. If $T \oplus T' = \delta$, where δ is the target difference, then increment a counter for those key bit values. At the end, the guessed key bits that produced $T \oplus T' = \delta$ at the expected rate are the most likely candidates.

Windowing will be used differently for differentials than for pseudo-linear approximations. This is because higher accuracy is required when working with differentials. Specifically, we need to form our target windows such that no subtraction is left to chance. We need to cover the word from the least significant bit to the highest position across all of our windows so that all borrows are known.

What we can do is start small with the windows, say $w = 4$. Once all the bits involved with that window size have been found, increase w. The window position does not change, but we slowly add more of the unknowns to the windows. This proceeds until all n bits of a key word are active and the entire word has been guessed, *or* the correct key is no longer distinguishable from incorrect keys. This can happen if all the new key bits guessed do not change the probability of resulting in the target difference. Note that the difference in a word is the difference over the entire n bit word, where inactive bits are zero.

4.5 Application

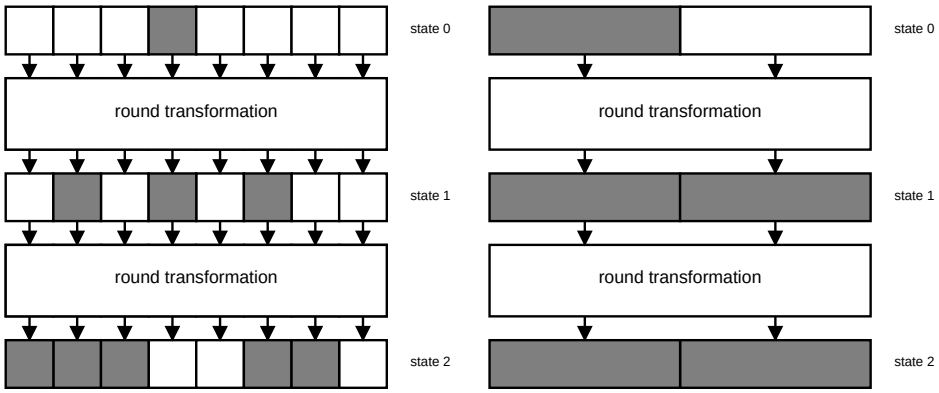
This technique can be applied to an ARX compression function or cipher by following the differences throughout execution. The probability of the characteristic $\Delta_P \rightarrow \Delta_C$ is obtained by taking the total probability of all trails beginning with Δ_P and ending with Δ_C. An application of this technique to Threefish-512 is presented in section 6.3.

Chapter 5 – Diffusion Metrics for ARX Functions

Diffusion is an important property in any iterative cryptographic function. Loosely speaking, diffusion measures the amount of dependence and/or influence a subset (e.g. bit or byte) of the input or output has. When diffusion is low, an adversary can effectively isolate portions of the input and output and perform attacks with reduced complexity. On the other hand, when diffusion is high, the adversary should not be able to isolate significantly smaller subsets of data and is far less likely to achieve any significant improvements in attack complexity. Therefore it is important in round function design that there is an effective way to reason about diffusion, and ensure that some minimum guarantees are met.

There is little work which describes how to construct a good round function, and how to determine how good an existing round function is. *Branch number* is a metric currently in use, and was successfully used in the design strategy for the Advanced Encryption Standard (AES) [20].

ARX functions that use large word sizes, such as 32 or 64 bits, may be difficult to accurately capture with this metric. Consider the scenario in figure 5-1, where two round transformations have approximately the same amount of diffusion, but one round function is a traditional SPN which operates on bytes (figure 5-1(a)) and the other is an ARX function using 32-bit words (figure 5-1(b)). In both round functions, the same bits are part of a linear or differential trail. Portions of the state (byte or word) that contain at least one bit of the trail are called active, and are colored grey in figure 5-1. When operations are performed in 32-bit words, each word covers four bytes. As a result, all of `state 1` appears active in 5-1(b), whereas only $\frac{3}{8}$ of `state 1` is active in 5-1(a). Similarly, $\frac{5}{8}$ of `state 2` is active in figure 5-1(a), while again being completely active in figure 5-1(b). In this example, it appears that the diffusion



(a) 8-bit operations (b) 32-bit operations

Figure 5-1: Trail in two transformations with approximately the same amount of diffusion

is far better in 5-1(b), when it really is not. A state composed of few large words can produce branch numbers that are misleading.

In this chapter, we provide two approaches to more accurately measure diffusion in this type of round function. The first approach uses branch number, but changes the way an ARX function is described in order to provide finer granularity and more meaningful results. Second, we propose an alternative metric, *effective branch number*, that is based on branch number. This metric builds on our word-partitioning methods in the pseudo-linear approximation technique. It allows for smaller partitions and therefore a more accurate assessment of diffusion.

5.1 Round Structure

The branch number metric is defined in terms of key-alternating block ciphers. Although branch number can be applied to any mapping, we describe how ARX designs align with this structure.

A key alternating block cipher provides the following two properties [27]:

Alternation the cipher is defined as the alternated application of key-independent

round transformations and the application of a subkey (round key). The first subkey is applied before the first round and the last subkey is applied after the last round.

Binary Key Addition the subkeys are injected into the state via exclusive-or.

5.1.1 $\gamma\lambda$ Structure

The key-independent portion of a round comprises of two substructures – a local non-linear transformation (γ), and a linear mixing step (λ) [27], where linearity refers to $\mathbb{Z}_2^n$. γ satisfies the criteria that output relies on a limited number of input bits, and neighboring output bits depend on neighboring input bits. λ provides a high level of diffusion. The round transformation is formed by the composition of γ and λ. Specifically, a round function ρ is computed by $\rho = \lambda \circ \gamma$.

5.1.2 ARX and the $\gamma\lambda$ Structure

Addition modulo 2^n is non-linear in $\mathbb{Z}_2^n$. It also relies on a limited number of inputs (the two addends). Consider two addends, a and b, where a_i represents bit i of word a. Let $c = a \boxplus b$. Then $c_i = a_i \oplus b_i \oplus carry_{i-1}$, where $carry_{i-1}$ is determined by a neighboring bit addition. That is, the neighboring output bits depend on neighboring input bits. Therefore, addition modulo 2^n satisfies γ.

Rotation is a linear mixing function which changes bit positions, therefore spreading the influence. Therefore, rotation satisfies λ.

Exclusive-or may play a role in γ, λ, or both, depending on the particular ARX function. The combination of addition and exclusive-or is non-linear in $\mathbb{Z}_2^n$, so the combination satisfies γ. If rotation is followed by exclusive-or, then the combined operations are a linear function that provides diffusion. This combination satisfies λ.

However, exclusive-or on its own satisfies neither γ nor λ.

Keyless ARX functions

In some iterated hash functions, the round function may not contain a key injection step. That is, the one-to-one mapping between input and output is always the same and not one selected by a key. For example, CubeHash performs the same transformation on the state for every round. In this scenario, the message mixing step may be considered as the key mixing step to fit the definition of a key alternating block cipher. Alternatively, the key mixing step could be thought of as exclusive-or with the zero string.

5.2 Branch Number

The *branch number* metric is a measure of diffusion that applies to any mapping. In the context of this work, the mapping (or transformation) is a round function. The branch number provides a lower bound for the amount of diffusion in a round function, allowing simpler analysis of the weight (or probability) of a trail over several rounds. In the context of the $\gamma\lambda$ round structure, diffusion is provided by λ [27]. The metric has been defined for any partition [27], as well as a specific type of partition [20]. In [20], it is defined as part of an m-bit partition of the state, consisting of *bundles*. Bundles are adjacent and do not overlap, such that the state is divided into n_b bundles and $n_b m$ is the size of the state in bits.

A bundle is active when it is part of a trail. That is, when a difference or selection pattern is non-zero for that bundle. The *bundle weight* of a state is defined as the number of active bundles in that state. The bundle weight of state a is written $w_b(a)$.

The branch number metric is defined differently for linear and differential trails

[20]. As the number of rounds increases, the branch number becomes more complex and difficult to reason about. However, it has been shown that properties of round functions can be derived by looking at two-round trails, and viewing $2r$ rounds as a sequence of r two-round trails[20]. Therefore, only two-round trails will be discussed here.

5.2.1 Differential Branch Number

For input vectors a and b, the differential branch number for round transformation ϕ is defined as [20]:

$$B_d(\phi) = \min_{a,b \neq a} \{w_b(a \oplus b) + w_b(\phi(a) \oplus \phi(b))\} \tag{5.1}$$

Generally speaking, this is the bundle weight of the input difference to the first round, added to that of the input difference to the second round. This is meaningful because it provides a lower bound on the propagation of the input difference pattern to the output of the first round, and thus illustrates a lower bound on the diffusion of a single bundle or set of bundles.

5.2.2 Linear Branch Number

For a boolean vector x and selection pattern α, $\alpha^T x$ is the dot product of x and α.

For a round 1 input selection pattern α, round 2 input selection pattern β, correlation function $C(.)$, and input x, the linear branch number of a round transformation ϕ is defined as [20]:

$$B_l(\phi) = \min_{\alpha,\beta,C(\alpha^T x, \beta^T \phi(x)) \neq 0} \{w_b(\alpha) + w_b(\beta)\} \tag{5.2}$$

Generally speaking, this is the minimum of the sum of bundle weights for the inputs to rounds one and two over all selection masks, where the active input and output to the ϕ function are correlated.

5.2.3 Application to ARX

In the following two sections, we propose two ways to evaluate ARX functions. The first changes the way we view an ARX function and uses the branch number metric. In particular, we show how to express an ARX function as an SPN.

The second uses windows from pseudo-linear analysis to consider only the bits that a target always depends on. That is, we use windows to represent the bundles. The carry into a window does not affect the branch number but affects the probability of a trail.

5.3 ARX as SPN

Consider the example ARX round in figure 2-3. We can represent addition modulo 2^n as a series of smaller data-dependent substitution boxes. This view is shown in figure 5-2. There are two kinds of S-boxes:

S_0**:** $a_i = a_i + b_i$

S_1**:** $a_i = a_i + b_i + 1$

Each bundle may be thought of as a window. Starting with the least significant window, apply S_0 to a_0. Note that the result is also dependent on the bundle b_0. Moving from the least significant bundle towards the most significant bundle, use $S_0(a_i)$ when there is not a carry out of the previous bundle addition. When there is a

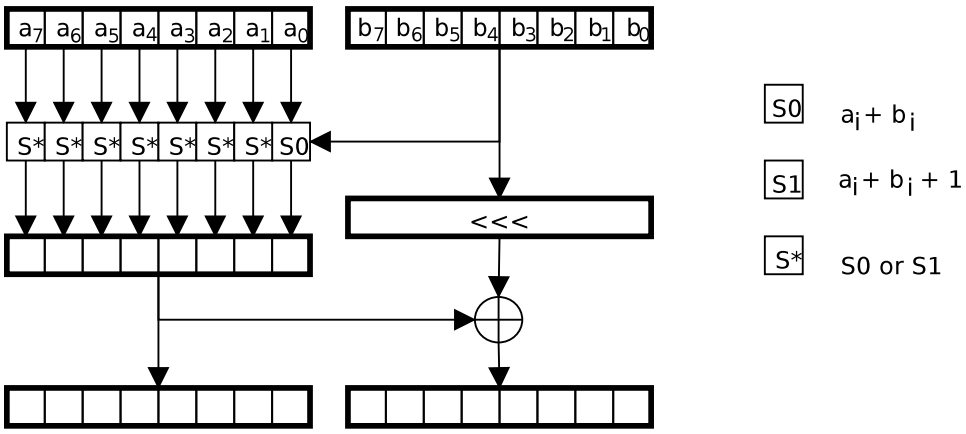


Figure 5-2: ARX as SPN

carry out of the previous bundle addition, $S_1(a_i)$ is used. This gives the same result as the original function.

Note that a bundle will not be considered active in a differential trail unless there is a non-zero difference in the input to the corresponding S-box. That is, if there is a zero difference in the output of an S-box, but the data-dependent S-boxes are different, the bundle will *not* be considered active. The rationale for this is that branch number provides a lower bound, so we only consider the minimum number of S-boxes affected.

However, this still leaves the matter of bundle size. A good choice of bundle width is a byte. We assume that $8|n$ because of software and hardware implementation. We do not assume any larger bundle sizes because it is typically the smallest size one will encounter in hardware. With this model, bundle weight and branch numbers are then calculated as described in equations 5.1 and 5.2.

The linear branch number calculation is easily extendable to pseudo-linear analysis, as one can find the dependencies of any target window in the same fashion as individual bits. However, this does introduce some variance based on the window size, w. Therefore, the pseudo-linear branch number, B_{pl}, should be computed with

respect to the window size. The equation for pseudo-linear branch number is shown in equation 5.3.

$$B_{pl}(\phi, w) = \min_{\alpha,\beta,C(\alpha^T x,\beta^T \phi(x))\neq 0} \{w_b(\alpha) + w_b(\beta)\} \tag{5.3}$$

5.4 Effective Branch Number

In this section, we examine the possibility of using the window dependencies in pseudo-linear analysis as bundles. That is, bits are active when they are in a window that belongs to a difference or selection pattern. The important issue we address is that of window overlap; this problem is not present in the original view of non-overlapping bundles. Here, two active w-bit windows may share up to $w-1$ bits and remain distinct windows. This can lead to very misleading results unless the overlap issue is addressed.

We now introduce a variation of the branch number metric, which we call *effective branch number*. This variant addresses the matter of overlapping bundles by explicitly discounting window overlap. It takes into account two aspects of the trail: (1) the number of windows, and (2) the amount of state spanned.

The very first step is to determine the proper window (bundle) size. In pseudo-linear cryptanalysis, the bounds on window size w are defined as $1 \leq w < n$. The minimum useful window size is that which allows a round function to be distinguished from a random permutation. This in turn depends on the number of additions required to compute a target window, and we denote it by $n_{\boxplus}$. If $w = n_{\boxplus}$, then it may not divide evenly into the state. It is desirable that if windows do not overlap, then the effective branch number is equivalent to the branch number calculations of section 5.3. This also makes the upper bound of effective branch number the same

as branch number. To satisfy these constraints, w should be a power of 2. Thus, let $w = 2^{\lfloor lg(n_{\boxplus}) \rfloor}$. Note that only the additions occurring in the key-independent portion of a round function should be used to derive the appropriate w.

Just as the original branch number metric used bundle weight, this variant uses *effective bundle weight*, denoted ew_b. Effective bundle weight is calculated by taking the ceiling of the total number of active bits divided by the bundle size.

Example 5.4.1. *Consider a round function and approximation with five active input windows, but all five windows start one bit off from another. That is, we have windows starting in positions $i, i+1, i+2, i+3$, and $i+4$. The only bits that are not contained in multiple windows are the first bit of window $x(i)$ and the last bit of $x(i+4)$. The bit range covered by these windows is $[i \ldots i+4+w)$. The number of state bits in this range is $4+w$.*

If $w = 2$, then the effective bundle weight is $\lceil \frac{6}{2} \rceil = 3$. If $w = 4$, then the effective bundle weight is $\lceil \frac{8}{4} \rceil = 2$. If $w = 8$, then the effective bundle weight is $\lceil \frac{12}{8} \rceil = 2$.

The definitions for linear and differential effective branch number are simple extensions of the branch number metrics from [20]. They only differ in that effective bundle weight is used in place of bundle weight. We denote the differential effective branch number as EB_d and pseudo-linear effective branch number as EB_{pl}.

Over two-round trails, the differential effective branch number of a transformation ϕ is derived by equation 5.4. Recall from equation 5.1 that a and b are input vectors to the first round, and $\phi(a) \oplus \phi(b)$ is the difference at the input to the second round.

$$EB_d(\phi) = \min_{a,b \neq a,w} \{ew_b(a \oplus b) + ew_b(\phi(a) \oplus \phi(b))\} \tag{5.4}$$

Similarly, the pseudo-linear effective branch number of a transformation ϕ is derived by equation 5.5. The selection mask of the input is α, the selection mask at the

input to the second round is β, and $C(.)$ is a correlation function over the masks and an input vector, x. Note that the linear effective branch number is the pseudo-linear effective branch number where $w = 1$.

$$EB_{pl}(\phi) = \min_{\alpha,\beta,C(\alpha^T x,\beta^T \phi(x))\neq 0,w} \{ew_b(\alpha) + ew_b(\beta)\} \tag{5.5}$$

Chapter 6 – Analysis of SHA-3 Candidate Round Functions: Threefish and Skein

Threefish is a tweakable block cipher that is part of the Skein hash function family[23][24]. There are three variants designed for 256, 512, and 1024-bit blocks and keys. Unlike traditional SPN or Feistel cipher designs that use substitution boxes (S-boxes) for non-linearity, Threefish relies on combining addition modulo 2^{64} and exclusive-or. In other words, it is an ARX cipher.

Threefish comes in three variants: Threefish-256, Threefish-512, and Threefish-1024. Threefish-x has an x-bit state and x-bit key. State is maintained throughout the cipher in 64-bit words. There are three components to the Threefish round function – mixing, permutation, and key injection. We describe these in a general sense here, and then describe the specifics of each variant later on.

Key injection is the addition of a round key (also called a subkey) to the state. Each 64-bit word of the subkey is combined with a 64-bit word of the state via addition modulo 2^{64}.

Mixing is achieved by parallel applications of the mix function. There are eight different mix functions, and the appropriate one is determined by the round number and where in the state it is applied. Each mix function takes two words as inputs, a and b, and outputs the words a' and b' calculated by equations 6.1 and 6.2. The rotation constant is specific to a particular mix function. These rotation constants facilitate diffusion in a pair of words.

$$a' = a \boxplus b \tag{6.1}$$

$$b' = (a \boxplus b) \oplus (b \lll constant) \tag{6.2}$$

Finally, words are swapped in the permutation step. This facilitates diffusion among all words of the state.

The generalized algorithm is as follows:

- initialize state to message block
- add $subkey_0$ to the state
- for r from 1 to the number of rounds, do:
  - apply the mix function
  - apply the permutation
  - if $4|r$, add $subkey_{\frac{r}{4}}$ to the state

6.1 Threefish-256

Threefish-256 is comprised of 72 rounds over a 256-bit block with a 256-bit key.

Each round of Threefish-256 contains two parallel mix functions followed by a permutation. Each mix function takes two words as inputs, a and b, and outputs the words $a \boxplus b$ and $(a \boxplus b) \oplus (b \lll constant)$, where the rotation constant is specific to a particular mix function. The permutation (1 3) is then applied. That is, if we label the first state word as word 0 and the last state word as word 3, words 1 and 3 are swapped in the permutation function.

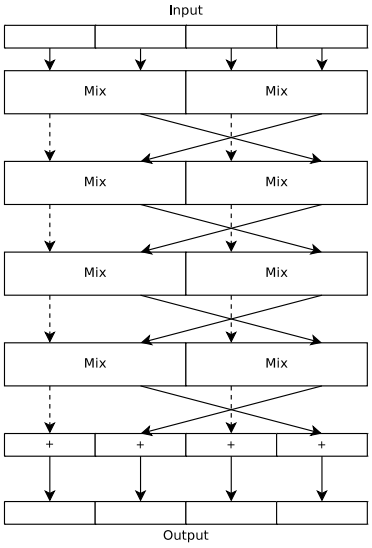


Figure 6-1: Four rounds of Threefish-256

If we label the four input words as $x0$, $x1$, $x2$, $x3$, and four output words as $y0$, $y1$, $y2$, $y3$, respectively, a round of mix and permutation functions can be expressed as

$$y0 = x0 \boxplus x1$$

$$y2 = x2 \boxplus x3$$

$$y1 = (x2 \boxplus x3) \oplus (x3 \lll r_{1,round-1})$$

$$y3 = (x0 \boxplus x1) \oplus (x1 \lll r_{0,round-1})$$

Every fourth round (4, 8, 12, etc.), a word of the round key is injected into each state word by addition modulo 2^{64}. Figure 6-1 depicts four rounds of Threefish-256.

Skein's UBI mode changes the key for each block, so a computationally intensive key schedule would have caused high performance overhead for the hash function.

Thus, Threefish has a very simple key schedule and relies on high diffusion over many rounds to provide protection from attacks. The key schedule is generated from a combination of key words and tweak words. There are four 64-bit key words (k_0 to k_3) and a fifth parity word (k_4) derived by equation 6.3. There are two 64-bit tweak words (t_0 and t_1), and a tweak parity word ($t_2 = t_0 \oplus t_1$). In each key injection, four of the key words are used in conjunction with two of the tweak words. In this work, we fix the tweak schedule to all zeros. The resulting key schedule is shown in equation 6.4.

$$k_4 = \lfloor \frac{2^{64}}{3} \rfloor \oplus \bigoplus_{0 \leq i \leq 3} k_i \tag{6.3}$$

$$roundKey_i = k_{i \bmod 5} || k_{(i+1) \bmod 5} || k_{(i+2) \bmod 5} || k_{(i+3) \bmod 5} + i \tag{6.4}$$

Because of the parity [1] relation in the key schedule, it is easy to calculate a portion of one key word given the corresponding bits in the other four key words.

Using techniques described in section 3, we can create approximations of windows throughout the cipher. Because the operations are performed on 64-bit words, $1 \leq w \leq 64$.

6.1.1 Approximation Over Modified 8 Rounds

We present the use of approximations on a modified version of Threefish-256. The target window is the least significant window of the first state word after round 8. The modified version has a fixed tweak schedule of zero, is reduced to 11 rounds, and does not include the injection of a whitening key. Encryption from rounds 1 to 4 is

[1] The parity constant was changed when Skein advanced to the final round. However, since pseudo-linear analysis does not take advantage of any structural properties of this constant, this change has no significant impact on the results presented here.

known because whitening has been removed, and decryption from rounds 11 to 8 is known because there are no key injections. Therefore this approximation covers the modified cipher from the key injection of round 4 to the key injection at round 8, inclusive.

Let x_i denote the i^{th} word of the plaintext and x_i^j the i^{th} word after round j. Let $x_i^j(m)$ denote the window of word x_i^j with its least significant bit in position m. That is, $x_i^j(m) = part(x_i^j, m, (m+w) \bmod 64)$.When a single word contains multiple active windows, $a_0 \ldots a_m$, let $x_i^j(a_0, \ldots, a_m)$ be the same as $x_i^j(a_0), \ldots, x_i^j(a_m)$. $K_i(m)$ denotes the window starting at m of the i^{th} word of the key schedule.

The approximation for $x_0^8(0)$ is presented in figure 6-2. The active addition windows are listed to the upper left of each $\boxplus$ symbol. $rotl(r)$ represents a left rotation by r bits of a 64-bit string.

Computations before the fourth round key addition can be performed by following the cipher operations, and are not included here. The number of key bits needed in this approximation depends on the window size. For example, $w = 3$ requires 58 key bits, $w = 4$ requires 75 key bits, and $w = 5$ requires 92 key bits.

We obtained empirical results for bias with varying window sizes, using 20,000,000 pseudorandomly generated (data, key) pairs and tweak schedule fixed to 0. We used the methodology in the original Skein submission to the SHA-3 competition [23], where differential biases were computed using 20,000,000 pseudorandom (data, key, tweak) tuples, as a model for our experiments.

As mentioned in section 3.3.8, there were two methods of implementation used – the less efficient and less accurate but easier to debug window isolation, and the more accurate and more efficient but more difficult to debug word isolation.

For simplicity, we represented carry pattern C^i as an array C^i_{jk}, where j is the round number and k is the word number. Each window in word k of round j was

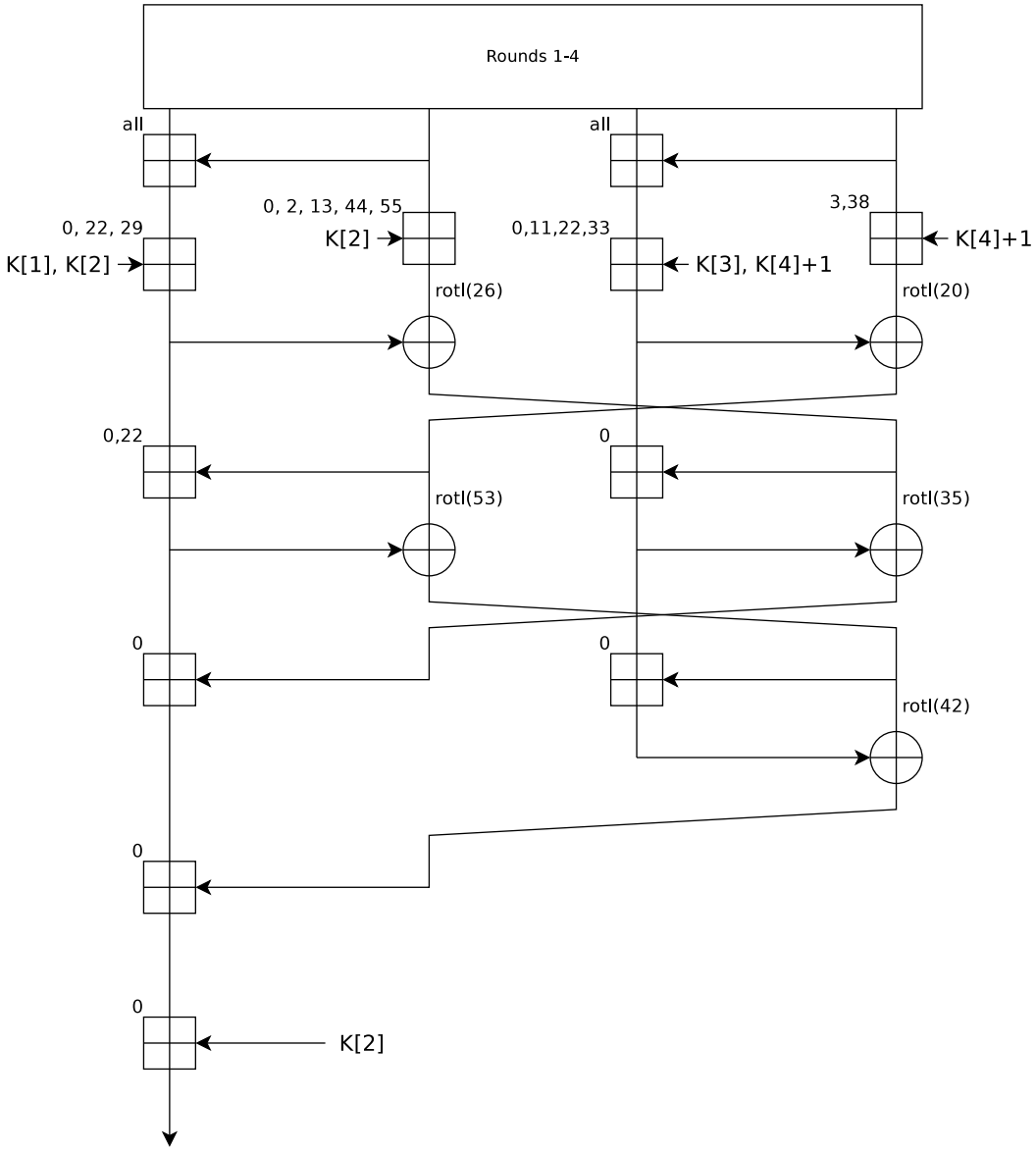


Figure 6-2: Approximation for $x_0^8(0)$, without whitening

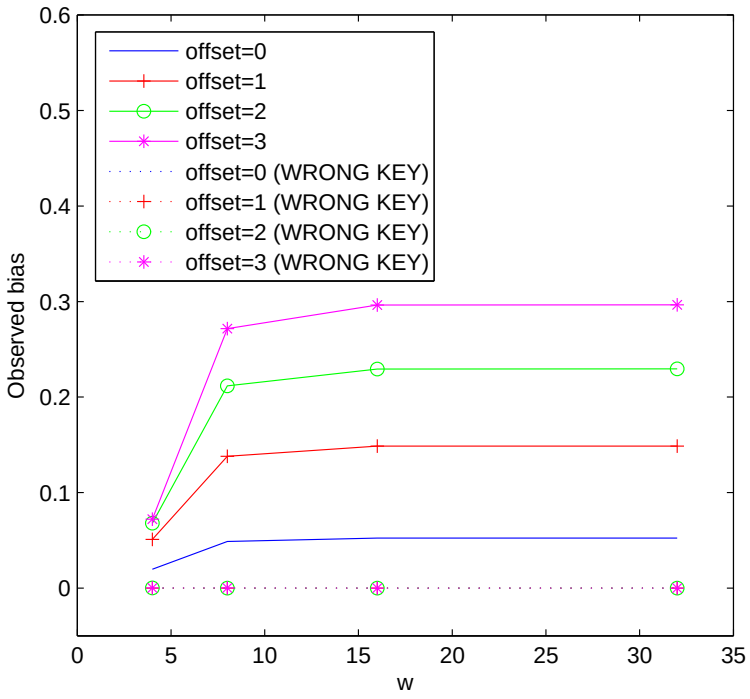


Figure 6-3: Base approximation with window isolation, $x_0^8(0)$

treated the same in these experiments.

Figure 6-3 demonstrates the effect of offsets on an approximation, with both the correct and incorrect key values. Offsets increase the probability of correctness for wrong values quite a bit when the range covers a large portion of the possible values, but the correct values still have a higher bias on average. The same is true when multiple carry patterns are used. Figure 6-3 also shows that there is a maximum bias that can be achieved with window isolation, where the maximum probability of correctness is strictly smaller than 1.

Figure 6-4 demonstrates the improvements that can be gained by using word isolation over window isolation, when compared with figure 6-3 for *offset*=0. This figure also demonstrates that the correctness probability grows to 1 as w increases using word isolation. This causes the bias to approach 1, since the probability of

correctness for a random guess drops closer to zero as w increases. Finally, it shows how different carry patterns can affect the approximation. The two carry patterns used here were:

$$C_j^0 k = 0,\ C_j^1 k = \begin{cases} 0 & \text{if } j \text{ is even} \\ 1 & \text{if } j \text{ is odd} \end{cases}$$

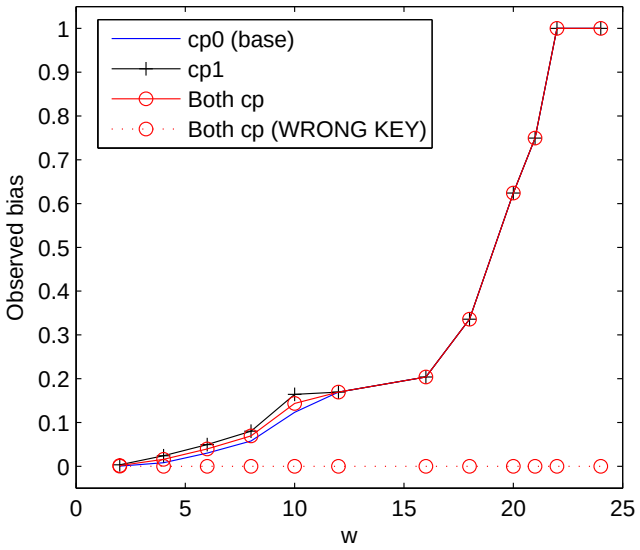


Figure 6-4: $x_0^8(0)$ approximation with word isolation and different carry patterns (cp)

When a large enough number of key bits is to be guessed, the bias values for different carry patterns converge (here, this happens at $w = 12$). If multiple carry patterns are used that have dissimilar biases, like the two used here, the bias of multiple patterns may be smaller than the bias obtained by using a single, stronger carry pattern.

6.1.2 Approximation Over Modified 12 Rounds

A 12-round approximation was derived over a variant with almost the same modifications as the variant used in our 8-round approximation. The one difference is that the cipher is reduced to 15 rounds instead of 11.

The 12-round approximation, shown in figure 6-5, uses a meet-in-the-middle approach. That is, it computes the value of a window from both the plaintext and the ciphertext, and is successful when they match. In the encryption direction, we approximate the least significant window of the first word after the 10^{th} round, $x_0^{10}(0)$. From the ciphertext, we then approximate the same word, denoted $d_0^{10}(0)$, and then check that $x_0^{10}(0) = d_0^{10}(0)$.

Figure 6-6 contains results from the 12 round approximation with carry pattern C^1 and different offsets using word isolation. It shows that the correct key bits are indistinguishable for $w < 5$. With $w = 5$, 232 bits of key need to be guessed. This approximation is computationally impractical for key recovery; however it may be possible to derive a better approximation that uses more of the same windows in rounds 4 and 12, thus reducing the number of key bits needed.

The empirical results presented in this section show that this method can be used to approximate a window with accuracy better than that obtained by random guesses. In the next section, we present a key recovery attack using the 8-round approximation.

6.1.3 Key Recovery

These approximations can be used for key recovery with known plaintext and ciphertext. Because there is no key injection in rounds 15 through 13, the output of round 15 can be decrypted to derive the output of round 12 correctly. The input up to the fourth round key injection can also be computed precisely because the modified

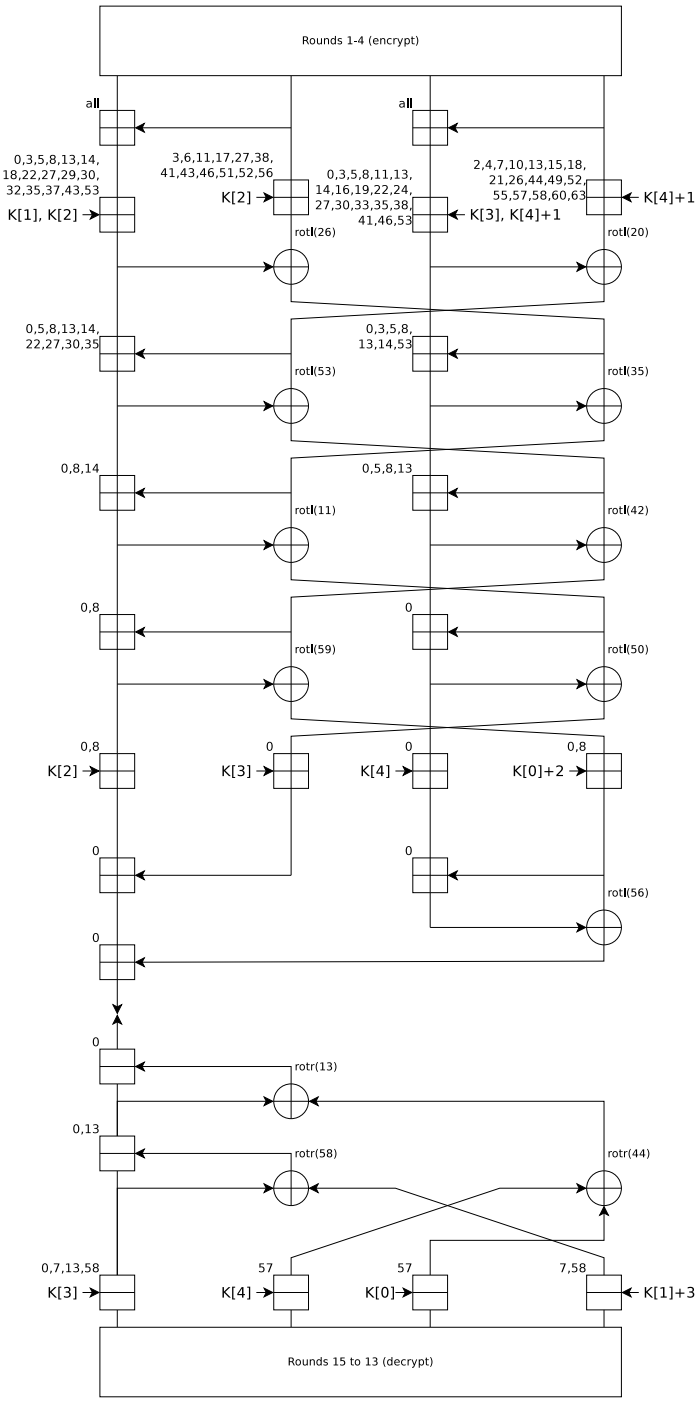


Figure 6-5: Approximation for 12 rounds, meeting at $x_0^{10}(0)$, without whitening

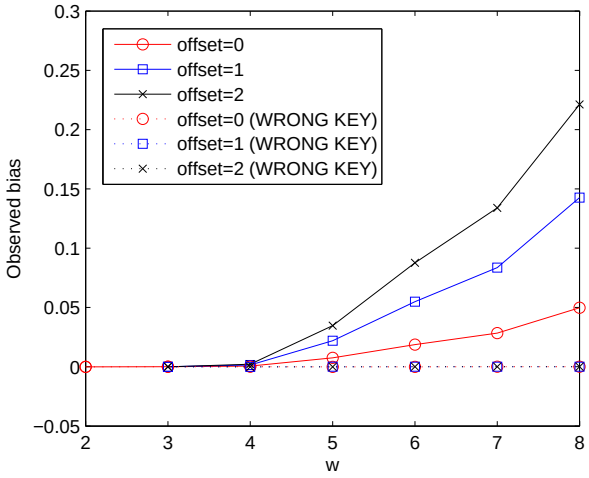


Figure 6-6: 12 round approximation

cipher does not include the whitening key injection. In this scenario, the 12 round approximation could be used to determine some of the key bits from 15 rounds.

To demonstrate key recovery using pseudo-linear approximations, a smaller attack was performed with the 8 round approximation for $x_0^8(0)$. Again, this gives an attack on 11 rounds of Threefish-256, since there is not another key injection until round 12.

If $w = 3$, then the attack on 11 rounds requires 58 bits of key (as compared to 256 bits for a brute force attack) to be guessed. Suppose that 40 bits needed in this attack were obtained by some other means, such as side channels. We show that the remaining 18 bits can be found using our approximations. In particular, we find the values of $k_1(0)$, $k_2(0)$, $k_3(0)$, $k_4(0)$, $k_2(3)$, and $k_4(11)$. This assumption that other bits are known prior to this attack was made due to limited time and resources. It is not a limitation of this technique.

Because the carry function is difficult to determine with incomplete knowledge of the operands, the correct value of the key does not always bubble to the top of the list of key candidates. However, it should still be near the top. In these experiments, the goal was to eliminate 90% of the possible key values, with the correct value

	offset=0	offset=1	offset=2	offset=3
correct	11%	12.2%	14.4%	7.2%
in top 10%	92.6%	97.4%	96.8%	92.2%

Table 6-1: 500 attacks, 10,000 pairs

in the remaining 10%. We ran 500 independent key recovery attacks, each with a pseudorandomly generated key and 10,000 pseudorandomly generated plaintexts. The results are summarized in table 6-1, where we note when the correct key value has the maximum bias, as well as when the correct value is in the top 10% of key estimates ranked by bias.

These results show that it is possible to eliminate many of the incorrect values while keeping the correct key bits with high probability, using only a portion of the key. While a larger value of the offset always helps in the task of approximating the output of the cipher, it does not always help in estimating key bits. This is because a larger offset is not necessarily more accurate, it simply encompasses a larger number of values. This will be true for incorrect key bits as well as correct key bits. For this reason, a larger window does not always help to effectively distinguish among correct and incorrect key bits. This can be seen in table 6-1 when the offset increases from 2 to 3. At *offset*=3, the number of values is so large that it becomes less effective in distinguishing the correct key bits.

6.2 Threefish-512

All Threefish variants use the same general mix permute structure. What changes is the number of parallel mix functions, the rotation constants, and the permutation. The main submission to the SHA-3 competition uses Threefish-512.

Threefish-512 has a 512-bit state, represented as eight 64-bit words. This means

Round mod 8	**Mix 0**	**Mix 1**	**Mix 2**	**Mix 3**
0	46	36	19	37
1	33	27	14	42
2	17	49	36	39
3	44	9	54	56
4	39	30	34	24
5	13	50	10	17
6	25	29	39	43
7	8	35	56	22

Table 6-2: Threefish-512 Rotation Constants (round 2)

x	0	1	2	3	4	5	6	7
$perm(x)$	6	1	0	7	2	5	4	3

Table 6-3: Threefish-512 Permutation

that a round function applies four parallel mix functions. The key schedule also differs from Threefish-256 in that the parity word is the exlusive-or of eight key words and a constant.

The rotation constants, as of round 2, are listed in table 6-2. Note that in this table, "Round" refers to the last round computed, not the current round.

Table 6-3 details the permutation step. Loosely speaking, even-indexed words rotate two words to the left, and words 3 and 7 swap.

In sections 6.1.1 and 6.1.2, we used a modified Threefish-256 where the whitening step was eliminated. In the following approximations, the only modification made is the number of rounds. That is, the first key injection considered is the key whitening.

6.2.1 8 Round Approximation

We have constructed an eight round approximation with a meet-in-the-middle approach. It meets at word 1 at the end of the fourth round. The addition (subtraction) windows for this distinguisher are presented in table 6-4. There are no subtraction

Round	Mix 0	Mix 1	Mix 2	Mix 3
1	0, 3, 14, 20, 31, 34, 47, 51	0, 3, 20, 22, 47	0, 20	0, 20
2	0, 3, 20, 47	0	0	0
3	0, 20	0	–	–
4	0	–	–	–
5	–	–	–	–
6	–	–	–	39
7	–	–	39, 56	52
8	21	14, 31, 39, 56	31, 52	13

Table 6-4: Addition masks for 8 round approximation, mix functions

windows in the fifth round because no subtractions are needed to obtain $x_1^4(0)$.

Empirical results for this distinguisher are presented in figure 6-7. These results were obtained using 20,000,000 randomly chosen $(plaintext, key)$ pairs with the carry pattern where carries are assumed every other round. It is clear from the plot that eight rounds of Threefish-512 are distinguishable for $w \geq 5$.

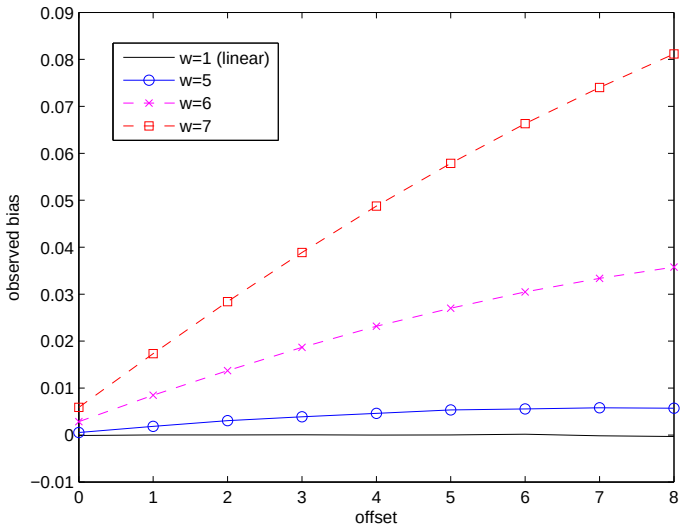


Figure 6-7: Observed bias for Threefish-512 8 round distinguisher

The number of key bits required by this distinguisher for window sizes $w = 5$, $w = 6$, and $w = 7$ are 310, 355, and 391, respectively. Recall that for a distinguisher

from Threefish-256, eight rounds with $w = 5$ (seen in the modified 12 round distinguisher, section 6.1.2) required 232 of the 256 bits, or about 90%. In this eight round distinguisher for Threefish-512, a window size of 5 requires only about 60% of the 512 key bits to be guessed. The principal reason for this is that with the increased state size, diffusion between words is slower. Specifically, it takes more applications of the permute function, and therefore more rounds, to mix all eight state words.

6.2.2 12 Round Approximation

A 12 round approximation can be obtained in a similar fashion to the 8 round one. We present one that meets in the middle at word 1 of the seventh round. The addition (subtraction) windows for this distinguisher are presented in table 6-5.

The key bits that need to be guessed for $w = 4$ and $w = 5$ are 509 and 511, respectively. Beyond that, all 512 bits need to be guessed for this distinguisher.

Empirical results are presented in figure 6-8 and 6-9 for two different plaintext selection criteria. In figure 6-8, 20,000,000 $(plaintext, key)$ pairs are chosen at random. The alternating carry pattern was used. In figure 6-9, key is still chosen at random, but the plaintexts are chosen such that they have a hamming weight of 1. For each randomly selected key, 512 plaintexts are used. Carries are assumed in rounds 6 and 8.

Figure 6-8 shows that one cannot effectively distinguish the correct key from incorrect keys when $w \leq 5$ and plaintexts are selected at random. When $w > 5$, all 512 bits need to be guessed.

However, when the model changes to allow the adversary to only choose low Hamming weight messages, the results improve. Figure 6-9 shows that when the message has a Hamming weight of 1, the correct key can be distinguished from a

Round	**Mix 0**	**Mix 1**	**Mix 2**	**Mix 3**
1	0, 1, 3, 7, 8, 10, 12, 14, 15, 18 ,20, 21, 23, 25, 26, 28, 31, 32, 34, 38, 39, 40, 43, 45, 46, 47, 51, 54, 55, 56, 59, 63	0, 3, 7, 8, 9, 10, 12, 15, 20, 22, 23, 25, 28, 30, 31, 32, 34, 37, 40, 45, 47, 51, 54, 55, 59	0, 1, 7, 8, 10, 12, 14, 16, 20, 24, 25, 26, 28, 31, 32, 37, 38, 40, 45, 50, 51, 55, 56, 60	0, 1, 4, 7, 8, 10, 12, 13, 16, 20, 24, 25, 28, 29, 31, 32, 37, 38, 40, 45, 49, 51, 53, 55, 56, 62
2	0, 3, 7, 8, 12, 15, 20, 25, 28, 31, 32, 34, 40, 45, 47, 51, 54, 55, 59	0, 1, 7, 12, 16, 20, 25, 31, 32, 40, 45, 51, 55, 56	0, 8, 10, 12, 25, 28, 38, 40, 51	0, 8, 10, 12, 15, 23, 25, 40, 51
3	0, 7, 12, 20, 25, 32, 45, 51	0, 8, 12, 25, 51	0, 10, 40, 51	0, 31, 40, 51, 55
4	0, 12, 25, 51	0, 40, 51	0	0
5	0, 51	0	0	0
6	0	0	–	–
7	0	–	–	–
8	–	–	–	0
9	–	–	0, 22	–
10	58	0, 19, 22, 41	–	–
11	0, 4, 19, 22, 27, 41, 46, 49	33, 55	36	27, 58
12	18, 21, 33, 40, 55	5, 8, 27, 36	2, 21, 24, 27, 33, 43, 58	0, 2, 4, 17, 19, 21, 22, 27, 36, 39, 41, 44, 46, 49, 58, 63

Table 6-5: Addition masks for 12 round approximation, mix functions

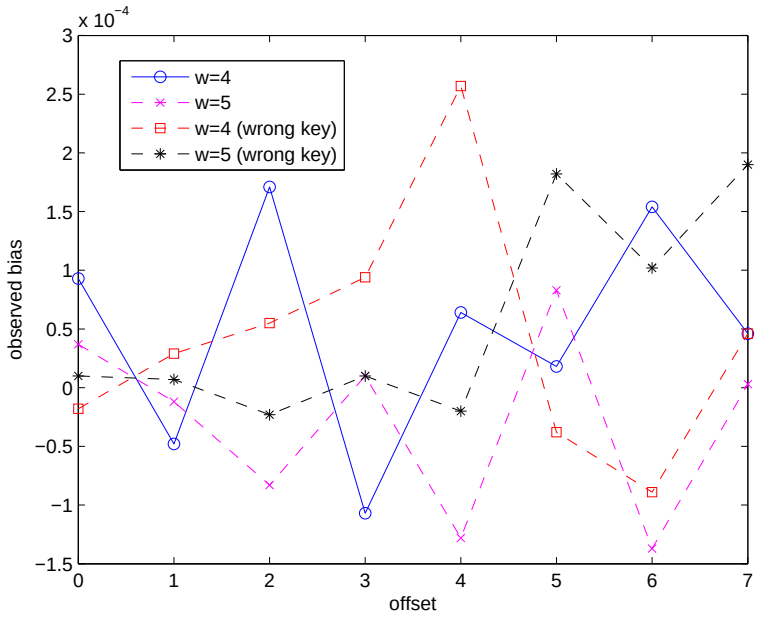


Figure 6-8: Observed bias for 12 round Threefish-512 distinguisher

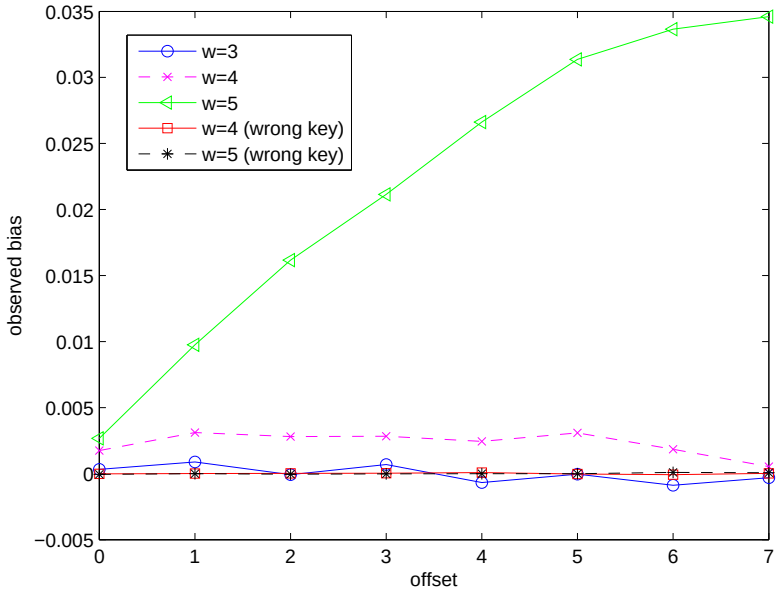


Figure 6-9: Observed bias for 12 round Threefish-512 distinguisher, low Hamming weight messages

random permutation with $w = 4$ and offset ≤ 6. However, it cannot be distinguished from a random permutation for $w \leq 3$, so 509 bits is the minimum that must be guessed for this distinguisher.

6.3 Truncated Differentials for Threefish-512

Here we demonstrate a truncated differential attack on four rounds of Threefish-512. This is a single-key chosen-plaintext attack with a pre-determined difference between pairs of plaintexts. Our focus is on differences in window bits. Differences in bits outside the window are ignored.

Let the input be $x'_0 = x_0 \oplus 1$, $x'_i = x_i$ for $i \in \{1 \ldots 7\}$, and the target window be $x_3^3(55)$. The size of the window will begin small, such as $w = 4$, and then increase as the adversary gains more information.

The target window location is not arbitrary. Recall from section 4.4 that we need to remove uncertainty introduced by the carry by working with the least significant window for all subtractions. Since there is a right rotation by 9 after the key subtraction, we have $(0 - 9) \mod 64 = 55$, so the target window for x_3^3 starts at position 55.

To see how the target window is affected by the input difference, it is helpful to observe how the difference propagates through the trail for x_3^3. The first three rounds of Threefish-512 are shown in figure 6-10. The trail is marked with thick arrows.

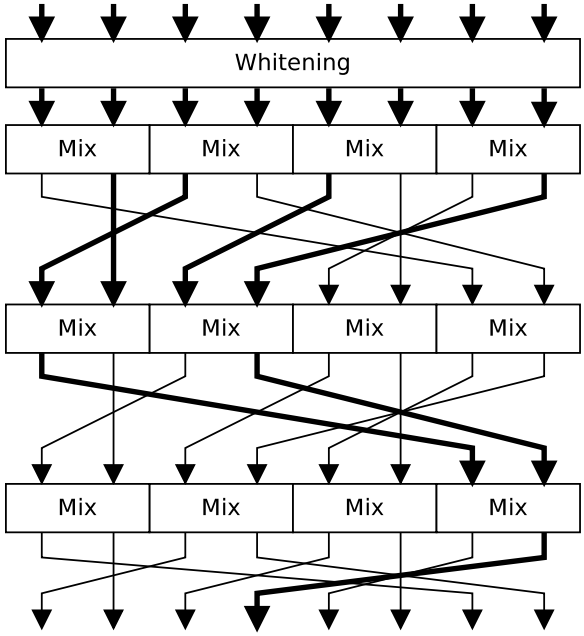


Figure 6-10: Three rounds of Threefish-512

6.3.1 First Round

Whitening

The first round begins with the key whitening. The same key will be added to both plaintexts, x and x', which are equal except in the least significant bit of word 0. Words 1 through 7 have no difference before and after the key injection.

In word 0, after key injection, the difference will consist of a string of $n - t$ zeros followed by t ones, $1 \leq t \leq n$. The reason for this is as follows (recall section 4.1.1). Without loss of generality, let x_0 be even and x'_0 be odd. Then $x_0 \boxplus k_0 = x^*_0$ will not cause a carry out of position 0 regardless of the value of k. If the least significant bit of k_0 (denote this as $k_0[0]$) is 0, then $x'^*_0[0] = 1$. If $k_0[0] = 1$, then $x'^*_0[0] = 0$. Assuming that k is selected from the uniform distribution at random, then $pr[x^*_0[0] = 0] = \frac{1}{2}$. If $k_0[0] = 1$, then $carry_0 = 1$ and $x'^*_0[1] = \overline{x^*_0[1]}$. This continues until $carry_{t-1} = 0$. Bits in position $t, \ldots, n-1$, are identical in x_0 and x'_0 and thus the run of ones ends at

after t bits.

A short list of possible difference values is provided in table 6-6. The approximated probabilities were obtained empirically over 20,000,000 randomly selected (x, key) pairs, where x' was derived from x.

Word	Input difference (Δ_{x_0})	Output difference (δ)	$Pr[\delta \mid \Delta_{x_0}]$ (approx)
x_0^*	1	1	2^{-1}
		3	2^{-2}
		7	2^{-3}
		F	2^{-4}
		1F	2^{-5}

Table 6-6: Key whitening difference probabilities

Mix and Permute

The first key injection is followed by the mix and permute steps. Recall that Threefish-512 computes four mix functions in parallel, where each mix function consists of an addition, word rotation, and exclusive-or. After mix function $i \in \{0, 1, 2, 3\}$ has been applied to words $2i$ and $2i + 1$, the words of the state are permuted to facilitate diffusion. There is no difference between words 1 in the two texts before the addition of the mix, but the words 0 differ by a string of ones. The rotation is performed on words 1, but there is no difference between them and thus there is still a zero difference after rotation. The final step of the mix is the exclusive-or of the rotated words 1 and result of the addition modulo 2^n. As a result, the words 0 still differ by a string of ones after the mix, and this string of ones is also the difference between the words 1 after the mix because these words differ from the corresponding words 0 only in that they are xored with the same string.

Permutation does not change the value of a difference within a word, only the word that it appears in. In particular, the string of ones forming the difference in word 0 after the mix step moves to word 6 after permutation, and the zero difference

in word 2 moves to word 0 (see table 6-3). Word 1 maps to itself during this step, so at the end of round 1, the only non-zero difference bits are in words 1 and 6.

Some of the higher probability characteristics are summed up in table 6-7. These probabilities are dependent on the input difference to the first round mix function. That is, they are conditional probabilities where the result depends on the difference after the whitening step.

Word	Input difference ($\Delta_{x_0^*}$)	Output difference (δ)	$Pr[\delta\|\Delta]$(approx)
x_6^1 and x_1^1	1	1	2^{-1}
		3	2^{-2}
		7	2^{-3}
		F	2^{-4}
		1F	2^{-5}
	3	1	2^{-1}
		3	2^{-2}
		7	2^{-3}
		F	2^{-4}
		1F	2^{-5}
	7	1	2^{-1}
		3	2^{-2}
		7	2^{-3}
		F	2^{-4}
		1F	2^{-5}
	F	1	2^{-1}
		3	2^{-2}
		7	2^{-3}
		F	2^{-4}
		1F	2^{-5}

Table 6-7: Round 1 difference probabilities

6.3.2 Second Round

The incoming differences are in x_6^1 and x_1^1. x_6^1 will affect output windows x_4^2 because word 6 will be permuted to word 4 after the addition, and x_3^2 because word 7 will be swapped with word 3. x_1^1 will affect x_6^2 as in the first round, because the zeroth word is permuted to the sixth position, and the first word is not affected by the permutation, and x_1^2. However, x_1^2, x_3^2, and x_4^2 are not in the dependency path for x_3^3, as can be seen

by following the words x_3^3 is derived from, and can be ignored for this differential. This leaves only x_6^2.

x_6^2 (derived from x_0^1 and x_1^1) is interesting because the format of the words is different from those we've seen before. In previous rounds, the differences in the left words have been strings of ones, but neither of the texts, at this part of the cipher, themselves contained the difference runs in the corresponding words. For example, a difference of 3 would be achieved by having the least significant bits 01_2 in one word and 10_2 in the other. In the input word x_1^1, one of the words contains the difference. That is, one word contains $\Delta_{x_1^1}$ in the least significant bits, while the other contains zeros. For a difference of 3, this would be 11_2 in one word and 00_2 in the other. Section 4.1.1 showed how the run of ones in the difference pattern can break in this scenario. Because of this, differences that are not runs of ones are possible, such as `0x05` and `0x0B`.

Examples of possible difference transitions are shown in table 6-8.

6.3.3 Third Round

Only one word in the trail has a difference into round 3: x_6^2. x_6^2 affects x_3^3.

Again, a difference in the right input creates more interesting transitions. The output difference probabilities for x_3^3 do not all contain perfect powers of 2 in the denominator. A subset of the possible transitions is shown in table 6-9.

6.3.4 A Truncated Differential

Subsections 6.3.1 to 6.3.3 showed the effect of the input difference throughout three rounds of a trail. We can use this information to construct a truncated differential attack on four rounds.

Word	Input difference ($\Delta_{x_1^1}$)	Output difference (δ)	$Pr[\delta\|\Delta]$ (approx)
x_6^2	1	1	2^{-1}
		3	2^{-2}
		7	2^{-3}
		F	2^{-4}
		1F	2^{-5}
	3	1	2^{-2}
		3	2^{-2}
		5	2^{-3}
		7	2^{-3}
		B	2^{-5}
		D	2^{-4}
		F	2^{-4}
		1D	2^{-5}
		1F	2^{-5}
	7	1	2^{-3}
		3	2^{-3}
		5	2^{-3}
		7	2^{-3}
		9	2^{-4}
		B	2^{-4}
		D	2^{-4}
		F	2^{-4}
		19	2^{-5}
		1B	2^{-5}
		1D	2^{-5}
		1F	2^{-5}

Table 6-8: Round 2 Difference Probabilities

Word	Input difference ($\Delta_{x_6^2}$)	Output difference (δ)	$Pr[\delta\|\Delta]$ (approx)
x_3^3	1	1	2^{-1}
		3	2^{-2}
		7	2^{-3}
		F	2^{-4}
		1F	2^{-5}
	3	1	$2^{-1.32}$
		3	2^{-2}
		5	$2^{-4.32}$
		7	2^{-3}
		D	$2^{-5.32}$
		F	2^{-4}
		1D	$2^{-6.32}$
		1F	2^{-5}
	5	3	$2^{-2.32}$
		5	2^{-2}
		7	2^{-3}
		D	2^{-3}
		F	2^{-4}
		1B	$2^{-6.32}$
		1D	2^{-4}
		1F	2^{-5}

Table 6-9: Round 3 Difference Probabilities

The trail covers the first three rounds – from the key whitening to the output of the third round. This is reflected in figure 6-10, where thicker lines indicate part of the trail. To perform the attack, the adversary guesses the round key of the target window. In this case the adversary guesses part of k_1 and k_8, subtracts it from the ciphertext pairs, reverses the fourth round permute and mix, and calculates the difference $x_3^3(55) \oplus x_3'^3(55)$. If the difference equals the expected difference at a rate close to expected, the guess is a candidate. If not, then it cannot be correct and is eliminated as a candidate.

The target difference will change as the window size changes. In particular, the target difference is zero when $w < 10$. As the window size increases, the target difference will change as more bits are included in the window.

We begin by taking a greedy approach and choose the highest probability output at each step, given the appropriate input differential. Tables 6-6 to 6-9 show that 1

w	Δ_3	Probability with right key	Key bits right	Probability with wrong key
4	0	1	0	0.562455
8	0	1	4	0.781984
12	1	0.375138	4	0.256083
12	1	0.375138	8	0.307487
16	1	0.333555	8	0.237504
16	1	0.333555	9	0.238292
16	1	0.333555	10	0.333471

Table 6-10: Probabilities for $\Delta_{x_3^3} = 1$

is the highest probability outcome at each step: that is, the most likely difference at each step is a difference only in the last bit of the string being considered. However, all paths that result in the target difference contribute to the actual probability. That is, one must calculate the total probability (equation 6.5) for each round to find the probability of output difference δ occurring given all possible input differences, Δ_i.

$$Pr[\delta] = \sum_i Pr[\delta|\Delta_i]Pr[\Delta_i] \tag{6.5}$$

This can be daunting because there can be up to 2^{64} difference patterns per word, per step. A fairly accurate approximation may also be obtained by considering the highest probability input differences at each step, such as those presented in tables 6-6 to 6-9. The more differences considered, the more accurate the approximation is.

The window size will start small, and gradually increase until all of both k_1 and k_8 have been guessed. Let us start with $w = 4$. Then 8 bits of key need to be guessed for the target window. Since the target window starts at position 55, and $55+4-1 < 64$, the target difference in the window is 0. Once the target window includes the least significant bit, the target difference will be 1.

Some empirical results are shown in table 6-10. Once the adversary knows 9 bits of k_1 and 9 bits of k_8, increasing w to include new bits will not help the adversary gain

w	Δ_3	Probability with right key	Key bits right	Probability with wrong key
4	0	1	0	0.562455
8	0	1	4	0.781897
12	7	0.250082	8	0.244532
16	1F	0.030764	12	0.026866
20	1F	0.030572	16	0.030526

Table 6-11: Probabilities for $\Delta_{x_3^3} = 1F$

new information. This happens because the newly guessed key bits are positioned such that they do not affect the differential, and therefore will not be distinguishable from incorrect key bits.

We could remedy this by using a longer difference pattern, such as `0x1F`. Although this difference pattern occurs with lower probability, it allows the adversary to find more of the key. In particular, the adversary can find up to 14 bits of each key word using this difference. Observed probabilities for this difference are given in table 6-11.

6.4 Diffusion in Threefish-512

The Threefish round function does not exactly fit the key-alternating function defined in section 5.1, but can easily be converted to one. We will allow that key injection is performed by addition modulo 2^n rather than exclusive-or. The main concern is that instead of applying one key-independent function and then one key injection in alternation, Threefish performs four key-independent functions and then one key injection.

Let *round* be the key-independent portion of a Threefish round function. Then write $Round = round \circ round \circ round \circ round$. Now alternation of *Round* and key injection follows the definition of a key-alternating block cipher.

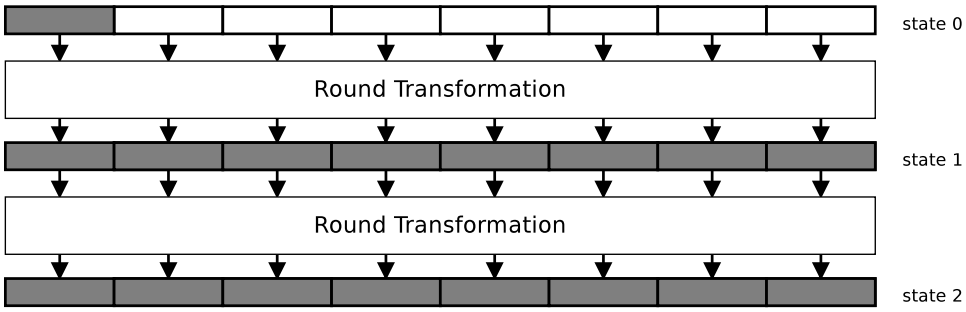


Figure 6-11: Differential trail for Threefish-512

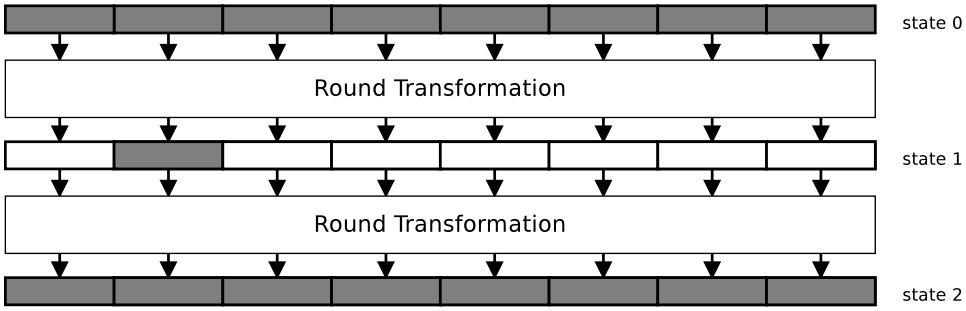


Figure 6-12: Linear trail for Threefish-512

6.4.1 Branch Number in Threefish-512: Word View

Consider a two-round trail that has a difference in least significant bit of the first word, depicted in figure 6-11. The input bundle weights (`state 0`) are $w_b(a \oplus b) = 1$, since only one word is active. At the input to the second *Round* (`state 1`), all eight words are active and therefore $w_b(Round(a) \oplus Round(b)) = 8$. It follows that $B_d(Round) \leq 9$. Note that the upper bound on the branch number in this case is 9.

Consider a linear approximation where one bit is active at the output of the first *Round* (`state 1`). Such an approximation is shown in figure 6-12. The 64-bit word encompassing that bit depends on all words at the input to the first *Round* (`state 0`). It follows that $B_l(Round) \leq 9$.

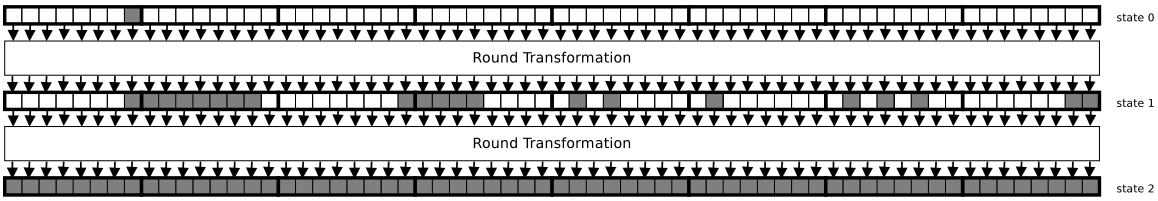


Figure 6-13: Differential trail for Threefish-512, SPN view

6.4.2 Branch Number in Threefish-512: SPN View

Using our SPN view (section 5.3) of Threefish-512, we can derive the linear and differential branch numbers.

Consider the differential trail in figure 6-11, where the difference is located in the least significant bit of the first word. The SPN view, where a byte is a bundle, is shown in figure 6-13. In the input to the first round, one bundle is active, so $w_b(a \oplus b) = 1$. At the input to the second *Round* (`state 1`), 21 bundles are active. Thus we have a bound $B_d(Round) \leq 22$.

In this view, with a byte as bundle, the upper bound for differential branch number is 65. Compare this to the differential branch number obtained by looking at 64-bit words in section 6.4.1. In section 6.4.1, all of `state 1` was active, whereas in the SPN view, the same difference pattern only activates much less of the state. Using the SPN model, it is apparent that the lower bound on diffusion is not as good as it was before. That is, a one-bit input difference in the SPN view produced differential branch number of at most 22, whereas the word view produced a differential branch number of at most 9, where 9 is the maximum possible for the word model. Calculating the differential branch number in the SPN view reveals more information about the propagation of an input difference by showing portions of the state where the difference does not propagate. Therefore, the SPN view provides a tighter upper bound over the same difference pattern.

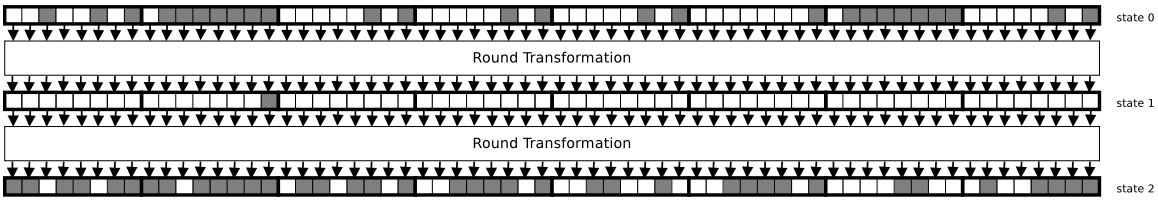


Figure 6-14: Linear trail for Threefish-512, SPN view

Figure 6-14 depicts a linear trail, with Threefish-512 as an SPN. At the input to the first *Round* (`state 0`), 26 bundles are active, giving $w_b(\alpha) = 26$. After one *Round* (`state 1`), there is only one bundle active, so $w_b(\beta) = 1$. This gives an upper bound on the linear branch number of $B_l(Round) \leq 27$.

The pseudo-linear branch number can easily be derived from the linear approximation by increasing w. In figure 6-7, it was shown that this approximation can be distinguished from a random permutation when $w \geq 5$. This will not alter the number of active bundles in β, the input to the second *Round*, but it will change the number of active bundles of the input. Because a bundle is a byte of the state and the active 5-bit window of β starts at the least significant bit, only one bundle is active. On the other hand, the weight of the input selection pattern, α, increases because some 5-bit windows do not align with bundles. That is, one 5-bit window can span, and therefore activate, two bundles. This results in an input bundle weight $w_b(\alpha) = 32$, giving $B_{pl}(Round, 5) \leq 33$.

6.4.3 Effective Branch Number in Threefish-512

Let x be the input to the mix function and y be the output. Write x_l for the left input and x_r for the right input, and similar for y. $y_r = (x_l \boxplus x_r) \lll constant$, but $(x_l \boxplus x_r)$ can be written as y_l. Therefore, we only consider each mix function to contain one addition.

Consider the 8 round trail of section 6.2.1, where a single window is computed in the forward and backward direction. There are 11 additions (excluding key additions) in the trail over the first *Round* (rounds 1-4). In *Round* 2 (rounds 5-8), there are 7 subtractions in the trail. Since the number of subtractions of the second *Round* is less than the number of additions in the first *Round* and branch number gives a lower bound, let $w = 7$. Overall, this yields $\lfloor lg(7) \rfloor = 2$ and $w = 4$.

Consider an input difference of 1 in the least significant bit of the first word, with the same differential trail as in figures 6-11 and 6-13. Then $ew_b(a \oplus b) = \lceil \frac{1}{4} \rceil = 1$. At the input to the second *Round*, 111 active bits are in the trail, giving an effective bundle weight $ew_b(Round(a) \oplus Round(b)) = \lceil \frac{111}{4} \rceil = 28$. This gives a bound on effective differential branch number, $EB_d(Round) \leq 29$.

Consider the approximation of section 6.2.1. At the input to the first *Round*, 113 bits are part of the trail. This gives effective bundle weight $ew_b(\alpha) = 29$. There are four active bits (one window) at the input to the second *Round*, giving $ew_b(\beta) = 1$. Therefore, $EB_{pl}(Round) \leq 30$.

Chapter 7 – Analysis of SHA-3 Candidate Round Functions: CubeHash

The compression function of CubeHash contains an ARX round function. It is not a keyed permutation, and thus cannot be used as a block cipher like Threefish, but its analysis is still relevant to the security of the overall hash function. In particular, distinguishers on the permutation are still relevant, as they show correlation between the input and output of the compression function. Any properties of the round function that could lead to attacks on the hash algorithm are especially important.

The round function has several *symmetric states* that can be used to mount attacks on the hash function with less work than specified by the SHA-3 algorithm requirements. A symmetric state occurs when a property of the input to the round function is preserved in the output.

7.1 CubeHash Overview

CubeHash is a parameterized hash algorithm that has a 1024-bit internal state, regardless of the output size. The state is represented as 32 32-bit words. The compression function injects a b-byte message block into the first b bytes of state via exclusive-or. An ARX round function is then applied to the state r times. A finalization process is performed after the final call to the compression function. This consists of adding 1 to the state by exclusive-or, followed by $10 \times r$ calls to the round function. Finally, the first h bits are returned as the hash value. These parameters are specified by CubeHash$r/b-h$.

The round two parameter recommendations for all h required by NIST are $b = 32$ and $r = 16$.

7.2 The CubeHash Round Function

CubeHash has a 1024-bit (128-byte) internal state size, represented as 32 32-bit words labeled x_{00000} to x_{11111}. Addition is performed modulo 2^{32}. Each round consists of 10 steps:

1. for each (j, k, l, m), $x_{1jklm} = x_{0jklm} \boxplus x_{1jklm}$
2. for each (j, k, l, m), $x_{0jklm} = x_{0jklm} \lll 7$
3. for each (k, l, m), swap x_{00klm} with x_{01klm}
4. for each (j, k, l, m), $x_{0jklm} = x_{0jklm} \oplus x_{1jklm}$
5. for each (j, k, m), swap x_{1jk0m} with x_{1jk1m}
6. for each (j, k, l, m), $x_{1jklm} = x_{1jklm} \boxplus x_{0jklm}$
7. for each (j, k, l, m), $x_{0jklm} = x_{0jklm} \lll 11$
8. for each (j, l, m), swap x_{0j0lm} with $x_0 j 1 l m$
9. for each (j, k, l, m), $x_{0jklm} = x_{0jklm} \oplus x_{1jklm}$
10. for each (j, k, l), swap x_{1jkl0} with x_{1jkl1}

Observe that the round function is a permutation over the set of states, and one can run it backwards as efficiently as one can run it in forward direction. Unlike Threefish, this is a fixed permutation. That is, there is no key to select a different permutation.

In the second round of the SHA-3 competition, the algorithm specified that 16 rounds are computed for each message block processed.

7.3 Symmetries

If the property is equalities between state words, there are 15 distinct symmetry classes with 67 subsets in CubeHash[2]. Once a state conforms to a symmetric state,

the state cannot get out of the symmetry until a nonzero block is processed. This is true regardless of the number of times the round function is applied.

For completeness, the symmetry classes are shown in table 7-1. Each letter represents a 32-bit word of the state.

C_1	AABBCCDD	EEFFGGHH	IIJJKKLL	MMNNOOPP
C_2	ABABCDCD	EFEFGHGH	IJIJKLKL	MNMNOPOP
C_3	ABBACDDC	EFFEGHHG	IJJIKLLK	MNNMOPPO
C_4	ABCDABCD	EFGHEFGH	IJKLIJKL	MNOPMNOP
C_5	ABCDBADC	EFGHFEHG	IJKLJILK	MNOPNMPO
C_6	ABCDCDAB	EFGHGHEF	IJKLKLIJ	MNOPOPMN
C_7	ABCDDCBA	EFGHHGFE	IJKLLKJI	MNOPPONM
C_8	ABCDEFGH	ABCDEFGH	IJKLMNOP	IJKLMNOP
C_9	ABCDEFGH	BADCFEHG	IJKLMNOP	JILKNMPO
C_{10}	ABCDEFGH	CDABGHEF	IJKLMNOP	KLIJOPMN
C_{11}	ABCDEFGH	DCBAHGFE	IJKLMNOP	LKJIPONM
C_{12}	ABCDEFGH	EFGHABCD	IJKLMNOP	MNOPIJKL
C_{13}	ABCDEFGH	FEHGBADC	IJKLMNOP	NMPOJILK
C_{14}	ABCDEFGH	GHEFCDAB	IJKLMNOP	OPMNKLIJ
C_{15}	ABCDEFGH	HGFEDCBA	IJKLMNOP	PONMLKJI

Table 7-1: Symmetry classes [2]

7.4 Symmetry Hierarchy

The 15 symmetry classes presented in [2] are very useful in analyzing CubeHash. Specifically, one can use this property to find collisions and preimages with lower complexity than the requirements demand. We add further structure to these symmetry classes by placing them into a hierarchy that describes how classes relate to each other. In particular, we provide further structure to describe the intersection of symmetry classes.

Let S be the state as an array of 32-bit words. Let $V = \{0,1\}^4$, the space of all 4-bit vectors, and D be a linear subspace of V. Then there is a symmetry in

CubeHash where a state that has $\forall d \in D$ and $\forall i \in (0,..,15), S[i] = S[i \oplus d]$ and $S[16+i] = S[16+(i \oplus d)]$.

For $D = V$, this yields a symmetry that has 2 free words, A and B, and 30 words fixed by the values of A and B. The corresponding symmetry class can only take on $2^{2\times32} = 2^{64}$ different values. Following the notation of [2], that symmetry class has the form shown in table 7-2.

$4d_1$	AAAAAAAA	AAAAAAAA	BBBBBBBB	BBBBBBBB

Table 7-2: 4-Dimensional symmetry class

Let D be a 3-dimensional linear subspace of V. There are 15 such subspaces, each representing a symmetry that has 4 free words, and the other 28 are defined by equality relations. The corresponding symmetry classes each contain $2^{4\times32} = 2^{128}$ distinct values. The 15 symmetry classes, which we will call 3-d symmetry classes, that result from these subspaces are listed in table 7-3.

$3d_1$:	AAAAAAAA	BBBBBBBB	CCCCCCCC	DDDDDDDD
$3d_2$:	AAAABBBB	AAAABBBB	CCCCDDDD	CCCCDDDD
$3d_3$:	AAAABBBB	BBBBAAAA	CCCCDDDD	DDDDCCCC
$3d_4$:	AABBBBAA	AABBBBAA	CCDDDDCC	CCDDDDCC
$3d_5$:	AABBBBAA	BBAAAABB	CCDDDDCC	DDCCCCDD
$3d_6$:	ABBAABBA	ABBAABBA	CDDCCDDC	CDDCCDDC
$3d_7$:	ABBAABBA	BAABBAAB	CDDCCDDC	DCCDDCCD
$3d_8$:	ABBABAAB	ABBABAAB	CDDCDCCD	CDDCDCCD
$3d_9$:	ABBABAAB	BAABABBA	CDDCDCCD	DCCDCDDC
$3d_{10}$:	AABBAABB	AABBAABB	CCDDCCDD	CCDDCCDD
$3d_{11}$:	AABBAABB	BBAABBAA	CCDDCCDD	DDCCDDCC
$3d_{12}$:	ABABABAB	ABABABAB	CDCDCDCD	CDCDCDCD
$3d_{13}$:	ABABABAB	BABABABA	CDCDCDCD	DCDCDCDC
$3d_{14}$:	ABABBABA	ABABBABA	CDCDDCDC	CDCDDCDC
$3d_{15}$:	ABABBABA	BABAABAB	CDCDDCDC	DCDCCDCD

Table 7-3: 3-dimensional symmetry classes

Let D be a 2-dimensional linear subspace of V. There are 35 distinct 2-dimensional

subspaces, yielding symmetries with 8 free words and 24 words defined by relations. The complete list of 2-dimentsional symmetry classes appears in table 7-4. These classes each have $2^{8\times 32} = 2^{256}$ possible values.

The 15 symmetry classes from [2] (shown in table 7-1) correspond to the nontrivial 1-dimensional subspaces. Only 16 words are free and the other 16 are completely determined by relations with the free words. This leads to $2^{16\times 32} = 2^{512}$ states belonging to each of these classes.

Finally, there is a symmetry class where all words are the same. In total, this yields 67 distinct symmetry classes, corresponding to the 67 subsets calculated in [2]. The intersection between two symmetry classes can be represented as the linear span of the union of their subspaces.

Figure 7-1 depicts the symmetry hierarchy. Let us start by considering it from the subspace perspective. Each 1-dimensional symmetry is part of 7 2-dimensional symmetries, and each 2-dimensional symmetry contains 3 1-dimensional symmetries. Each 2-dimensional symmetry is also part of 3 3-dimensional symmetries and each 3-dimensional symmetry contains 7 2-dimensional symmetries. All 15 3-dimensional symmetries are part of the single 4-dimensional symmetry. To view the hierarchy from the state perspective, simply switch the relations (i.e. a 1-d symmetry class contains 3 2-d symmetry classes).

If one selects one symmetry from each level in the graph such that the higher-dimensional symmetries always contain the lower dimensional one, then one can find a total of $15 \times 7 \times 3 = 315$ different symmetry hierarchies.

$2d_1$:	AAAABBBB	CCCCDDDD	EEEEFFFF	GGGGHHHH
$2d_2$:	AABBAABB	CCDDCCDD	EEFFEEFF	GGHHGGHH
$2d_3$:	AABBBBAA	CCDDDDCC	EEFFFFEE	GGHHHHGG
$2d_4$:	ABBAABBA	CDDCCDDC	EFFEEFFE	GHHGGHHG
$2d_5$:	ABBABAAB	CDDCDCCD	EFFEFEEF	GHHGHGGH
$2d_6$:	ABABABAB	CDCDCDCD	EFEFEFEF	GHGHGHGH
$2d_7$:	ABABBABA	CDCDDCDC	EFEFFEFE	GHGHHGHG
$2d_8$:	AABBCCDD	AABBCCDD	EEFFGGHH	EEFFGGHH
$2d_9$:	AABBCCDD	BBAADDCC	EEFFGGHH	FFEEHHGG
$2d_{10}$:	AABBCCDD	CCDDAABB	EEFFGGHH	GGHHEEFF
$2d_{11}$:	AABBCCDD	DDCCBBAA	EEFFGGHH	HHGGFFEE
$2d_{12}$:	ABABCDCD	ABABCDCD	EFEFGHGH	EFEFGHGH
$2d_{13}$:	ABABCDCD	BABADCDC	EFEFGHGH	FEFEHGHG
$2d_{14}$:	ABABCDCD	CDCDABAB	EFEFGHGH	GHGHEFEF
$2d_{15}$:	ABABCDCD	DCDCBABA	EFEFGHGH	HGHGFEFE
$2d_{16}$:	ABBACDDC	ABBACDDC	EFFEGHHG	EFFEGHHG
$2d_{17}$:	ABBACDDC	BAABDCCD	EFFEGHHG	FEEFHGGH
$2d_{18}$:	ABBACDDC	CDDCABBA	EFFEGHHG	GHHGEFFE
$2d_{19}$:	ABBACDDC	DCCDBAAB	EFFEGHHG	HGGHFEEF
$2d_{20}$:	ABCDABCD	ABCDABCD	EFGHEFGH	EFGHEFGH
$2d_{21}$:	ABCDABCD	BADCBADC	EFGHEFGH	FEHGFEHG
$2d_{22}$:	ABCDABCD	CDABCDAB	EFGHEFGH	GHEFGHEF
$2d_{23}$:	ABCDABCD	DCBADCBA	EFGHEFGH	HGFEHGFE
$2d_{24}$:	ABCDBADC	ABCDBADC	EFGHFEHG	EFGHFEHG
$2d_{25}$:	ABCDBADC	BADCABCD	EFGHFEHG	FEHGEFGH
$2d_{26}$:	ABCDBADC	CDABDCBA	EFGHFEHG	GHEFHGFE
$2d_{27}$:	ABCDBADC	DCBACDAB	EFGHFEHG	HGFEGHEF
$2d_{28}$:	ABCDCDAB	ABCDCDAB	EFGHGHEF	EFGHGHEF
$2d_{29}$:	ABCDCDAB	BADCDCBA	EFGHGHEF	FEHGHGFE
$2d_{30}$:	ABCDCDAB	CDABABCD	EFGHGHEF	GHEFEFGH
$2d_{31}$:	ABCDCDAB	DCBABADC	EFGHGHEF	HGFEFEHG
$2d_{32}$:	ABCDDCBA	ABCDDCBA	EFGHHGFE	EFGHHGFE
$2d_{33}$:	ABCDDCBA	BADCCDAB	EFGHHGFE	FEHGGHEF
$2d_{34}$:	ABCDDCBA	CDABBADC	EFGHHGFE	GHEFFEHG
$2d_{35}$:	ABCDDCBA	DCBAABCD	EFGHHGFE	HGFEEFGH

Table 7-4: 2-dimensional symmetry classes

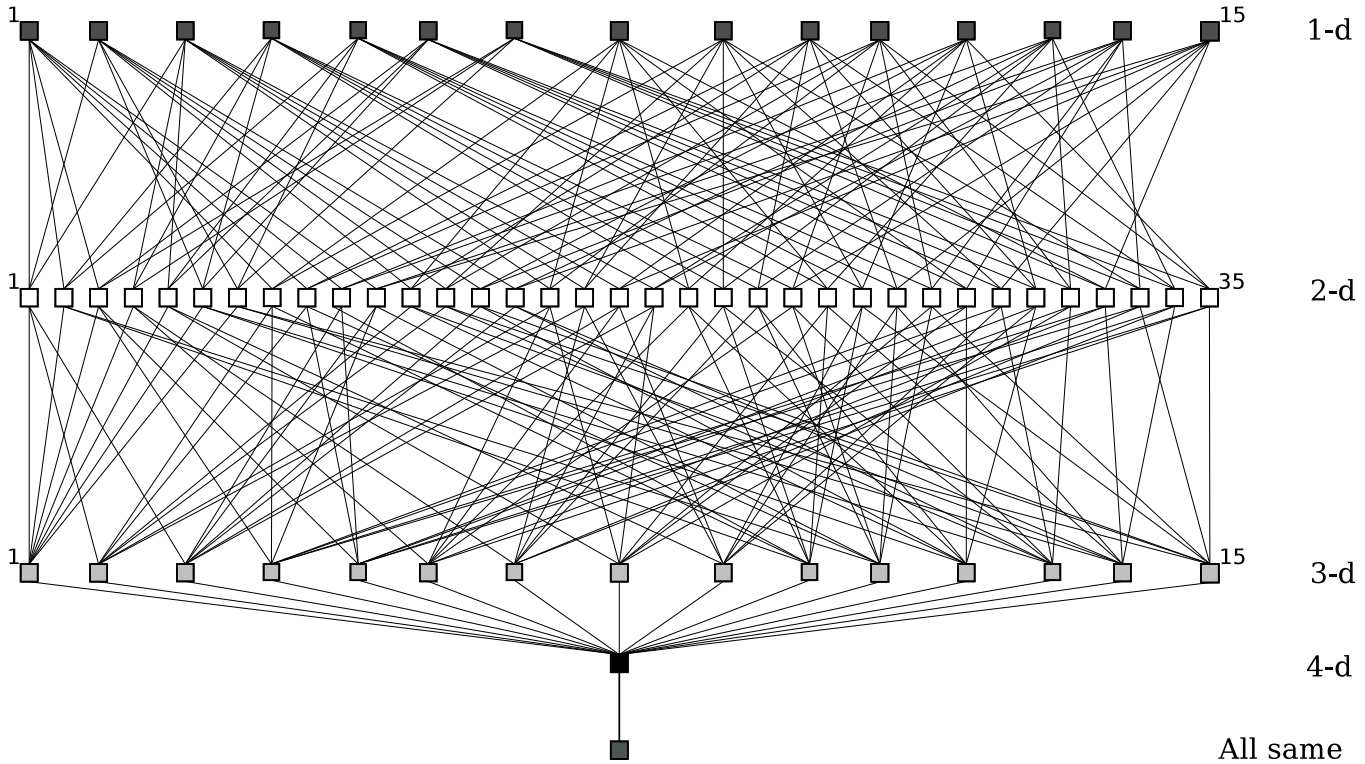


Figure 7-1: Symmetry Hierarchy

7.4.1 Traversing the Hierarchy

Now consider the symmetry class of each symmetry, where a x-dimensional symmetry class corresponds to an x-dimensional symmetry. Let an x-dimensional state refer to a state that has the same number of free words, but is not necessarily symmetric. It is easy to see that the intersection of 1-dimensional symmetric states leads to 2-dimensional symmetric states, but since there is no way to leave a symmetric state, it seems difficult to traverse the hierarchy from higher dimensions to lower ones in such a way that the higher dimension state is not part of a symmetry class in the higher dimension. For example, it is not clear how to go from a 2-dimensional state to a 1-dimensional state such that the 1-dimensional state is symmetric, but the 2-dimensional state is not in a 2-dimensional symmetry class. We show that there is a way using portions of the state. In particular, if the state is divided into halves or quarters, where each half or quarter belongs to a higher dimension symmetry class, a lower dimension symmetry may be achieved if the combination of higher-level symmetry class patterns allows it. Thus, the hierarchy can be traversed from 1 dimension to 4 dimensions through intersection, and from 3 dimensions to 1 dimension in this manner.

A 2-dimensional symmetric state can be reached from 3-dimensional halves that combined do not belong to a 3-dimensional symmetry class in the following way:

1. Set the left half of the state such that it conforms to the left half of a 3-dimensional symmetry class, $3d_i$, $i \in 1 \ldots 15$.

2. Set the right half of the state such that it conforms to the right half of a 3-dimensional symmetry class, $3d_j$, $j \in 1 \ldots 15$, $i \neq j$.

All 35 2-dimensional symmetry classes can be reached in this manner. If a particular 2-dimensional state is needed, there are further restrictions on the classes the

halves belong to. In order to reach a class $2d_i$, the 3-dimensional halves must belong to $3d_j$ and $3d_k$ such that $2d_i$ contains $3d_j$ and $3d_k$.

1-dimensional symmetric states can be generated in a similar way using halves from 2-dimensional symmetric classes. However, not all combinations yield a symmetric state. The left and right halves of the state should both belong to classes that a 1-dimensional class is a part of in the hierarchy. For example, C_1 contains $2d_1, 2d_2, 2d_3, 2d_8, 2d_9, 2d_{10}$, and $2d_{11}$. If each half is set to a pattern found in these symmetric classes, the state will belong to C_1.

It is convenient to divide the state into left and right halves, but not necessary. It may also be divided into 32-byte quarters, where each quarter is from a symmetric state. We call these quarter-symmetries. There are many duplicate state forms when only quarters are considered. For example, $3d_2$ and $3d_3$ both have the same pattern when only 8 words are viewed. Then we say $3d_2$ to mean either class, since they are equivalent in this context. Thus we only consider one class for these duplicates, as a representative.

Tables 7-5 and 7-6 show the quarter-symmetry transitions to lower-dimension symmetry classes. When traversing the hierarchy from 3-dimensional quarter-symmetries to 1-dimensional symmetry classes, the 2-dimensional quarter-symmetries must contain the 3-dimensional quarter-symmetries and be part of the 1-dimensional symmetry class. Note that only states where each quarter is self-contained can be reached via quarter-symmetries. That is, states that have equalities that cross the quarter boundary cannot be reached this way – that traversal must occur with half-symmetries.

Symmetry class	2d quarter-symmetry classes
C_1	1, 2, 3, 8(to 11)
C_2	1, 6, 7, 12(to 15)
C_3	1, 4, 5, 16(to 19)
C_4	2, 4, 6, 20(to 23)
C_5	2, 5, 7, 24(to 27)
C_6	3, 5, 6, 28(to 31)
C_7	3, 4, 7, 32(to 35)

Table 7-5: 2d quarter-symmetries to symmetry classes (equivalent quarters in parentheses)

Symmetry class	3d quarter-symmetry classes
C_1	1, 2(3), 4(5), 10(11)
C_2	1, 2(3), 12(13), 14(15)
C_3	1, 2(3), 6(7), 8(9)
C_4	1, 6(7), 10(11), 12(13)
C_5	1, 8(9), 10(11), 14(15)
C_6	1, 4(5), 8(9), 12(13)
C_7	1, 4(5), 6(7), 14(15)
$2d_1$	1, 2(3)
$2d_2$	1, 10(11)
$2d_3$	1, 4(5)
$2d_4$	1, 6(7)
$2d_5$	1, 8(9)
$2d_6$	1, 12(13)
$2d_7$	1, 14(15)

Table 7-6: 3d quarter-symmetries to symmetry classes (equivalent quarters in parentheses)

7.5 Improvements over Previous Attacks

7.5.1 A Flawed Preimage Attack

The attacks presented in [3] provided an interesting analysis of what can be done once in a symmetric state. However, one of the preimage attacks was flawed. In this attack, the adversary arrives at a symmetric state, S, in class C_i from the forward direction. Next, they arrive at state T, belonging to symmetry class C_j, from the backwards direction, where i and j do not have to be equal. Null message can than be processed to bridge between the two states.

There are two issues in this description.

1. It is impossible to bridge a state belonging to C_i with a state belonging to C_j when $i \neq j$. This is a direct result of the symmetry property. Once the state conforms to a symmetry class, it cannot deviate from that class until a nonzero message is inserted. The only way to arrive in C_j from C_i is to reach a state in $C_i \cap C_j$. However, this means that both states belong to C_i, and thus reinforce that this attack may only succeed when $i = j$.

2. Even when $i = j$, there is no guarantee that there is a path from S to T. This is due to fact that using null message produces a permutation of the state, and this permutation has disjoint cycles. This is easy to see when we consider fixed points, where the input state is equivalent to the output state. These show that the permutation is not a single large cycle, and thus disjoint cycles exist. Therefore if S and T lie in different cycles, one can never reach T from S.

7.5.2 An Improved Preimage Attack

The second attack for finding preimages presented in [3] does work, but can be improved. The original attack is performed as follows:

1. reach a symmetric state in target class C_k, S, in the forward direction
2. reach a state, T, in the backward direction such that T belongs to the same symmetry class as S
3. insert nonzero message blocks that keeps the state in the symmetry class until there is a path from S to T

The complexity of this attack is driven by the difficulty of arriving at a symmetric state. For $b = 32$, this complexity is 2^{384} calls to the compression function. Our improved attack is as follows:

Let us call the target hash value Z. Then:

1. Extend Z by $(1024 - h)$ bit to a full 1024-bit state and run the finalization backwards to get a state H_4.
2. Search for a message prefix M_1 and a $H_1 = H(H_0, M_1)$ in a symmetry class C_i, and a message prefix M_1' and a $H_1' = H(H_0, M_1')$ in a symmetry class C_j, $j \neq i$.
3. Search for a postfix M_4 and H_3 in the same C_i *or* C_j with $H(H_3, M_4) = H_4$.
4. Apply a meet-in-the-middle approach to search two nonzero message parts M_2 and M_3 and a state $H_2 \in C_i$ with $H(H_1, M_2) = H_2$ and $H(H_2, M_3) = H_3$.

In order for this attack to work, we must have $i, j \in \{1 \dots 7\}$, but any of these seven classes will do. This attack does not fix the class in steps 2 and 3 before the attack, but rather accepts whatever is reached first. These are the symmetry classes

where two equal words occur in the first 32 bytes – this is what allows us to insert nonzero message into the state while maintaining symmetry. For example, if the state arrived at in step 2 were in class C_1, the message would have the form `XX000000`. `X` could be anything – what is important is that the two words will remain equal. If the state were in class C_2, the message would have the form `X0X00000`.

The improvements are gained from the following points:

1. The cost of reaching a satisfactory symmetric state is now $2^{384}/7$ instead of 2^{384}. Since we need two that are in different classes, the cost for steps 2 and 3 is $2^{384}/7 + 2^{384}/6 < 2^{384}/3 \approx 2^{382.4}$ calls to the compression function.

2. The cost of reaching a satisfactory symmetric state in step 4 is now 2^{383}.

These improvements yield a total complexity of $\approx 2^{382.4} + 2^{383} \approx 2^{383.7}$, making this the fastest preimage attack on CubeHash16/32-h so far.

Now consider what would happen if CubeHash were altered slightly by setting $b = 33$, making the compression insert just 1 byte more of the message. Then any class will do, and the cost of steps 2 and 3 becomes 2^{253} (instead of 2^{256}). The cost of of step 4 is 2^{256}. Thus the attack has an overall cost of $3 \times 2^{256} + 2^{253} \approx 3 \times 2^{256}$

Improvement in the Fourth Step

We can possibly use pseudo-linear approximation to reduce the work of the fourth step using the method discussed in section 3.3.7.

Instead of trying out all 2^{8b} candidates for M_2 and all 2^{8b} candidates for M_3, use pseudo-linear analysis to find M_1 and M_2 such that $H(H_1, M_2) = H_2$ and $H(H_2, M_3) = H_3$. This is pictured in figure 7-2, with additional interim states T_1 and T_2 immediately following the injection of M_2 and M_3, respectively.

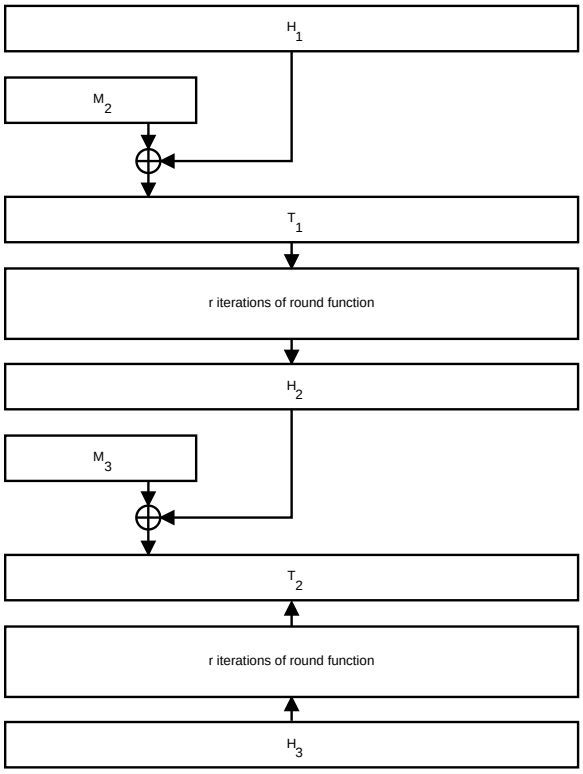


Figure 7-2: Step 5 - finding a bridge between states H_1 and H_3

target	**dependencies after 1 round**	M_2 **dependencies**
$H_2[16](0)$	1(0),3(0),9(25),17(0),19(0)	0(0),1(0,18),2(0),3(0),5(7,21),7(21)
$H_2[17](0)$	0(0),2(0),8(25),16(0),18(0)	0(0,18),1(0),2(0),3(0),4(7,21),6(21)
$H_2[20](0)$	5(0),7(0),13(25),21(0),23(0)	1(7,21),3(21),4(0),5(0,18),6(0),7(0)
$H_2[21](0)$	4(0),6(0),12(25),20(0),22(0)	0(7,21),2(21),4(0,18),5(0),6(0),7(0)

Table 7-7: Two-round dependencies

In the worst case, all 2×2^{8b} candidates would need to be tried as before. Depending on the value of r, we may be able to reduce this work significantly. This will only work, however, until full diffusion is achieved. The modified step 4 proceeds as follows:

1. Compute $T_2 = round^{-r}(H_3)$, set last $128-b$ bytes of H_2 to last $128-b$ bytes of T_2

2. Solve for words 16, 17, 20, and 21 of H_2 to obtain M_2, using pseudo-linear approximations

3. Find appropriate M_3 such that $H_2 \oplus M_3 = T_2$

Step 1 is trivial and gives us $128-b$ bytes of H_2.

In the second step, we ignore the portion of the state that M_3 is combined with. Instead, we use words that only depend on M_2 and known values to reduce complexity. Further, the diffusion for words x_{10000} to x_{11111} is slower, so words have fewer dependencies. The complexity of this step is highly dependent on r.

As an example, consider $r = 2$ and $b = 32$. The dependencies on M_2 are found in table 7-7. If we let $w = 8$, we can find 168 bits of M_2 in time about $2^{64}+2^{32}+2^{41}+2^{21}$, by solving for the least significant window of each of the words. The complexity for each word decreases because of overlapping windows in M_2 – there is no need to try values that have already been determined to be incorrect. The remaining 88 bits may be found by brute force, or by further approximation.

For $r = 16$, we expect the complexity of the fourth step to remain 2×2^{8b}.

Once we have a message block M_2 that gives the correct value in the last $128 - b$ bytes of H_2, finding M_3 is simple. We know all of H_3, and therefore all of T_2. Our M_2 gives us $H_2 = H(H_1, M_2)$ – the full value of state H_2. The only operation between H_2 and T_2 is the message addition, so we can easily find that message block by calculating $M_3 = H_2 \oplus T_2$. Therefore this step is also trivial.

Note that this technique can be performed for any value for b and h. The only parameter that prevents this from working effectively is r large enough such that any meaningful approximation requires all 2^{8b} bits of M_2 to be guessed.

7.6 Multicollisions

Symmetric states also allow us to mount a Joux-style multicollision attack [28]: finding a k-collision takes the time sequentially computing $\lceil \log_2(k) \rceil$ collisions, each in time 2^{256}.

Reaching a symmetric state costs $2^{381.2}$. Once in a symmetric state, the complexity to find a k-collision is $\lceil \log_2(k) \rceil \times 2^{256}$. This gives a total cost of

$$2^{381.2} + \lceil \log_2(k) \rceil \times 2^{256},$$

which is dominated by the cost for getting into a symmetric state, regardless of k.

Again, let us consider increasing the block size by 1 to $b = 33$. This significantly reduces the cost of finding k-collisions. It can be done in time

$$2^{253} + \lceil \log_2(k) \rceil \times 2^{256} \;\approx\; \lceil \log_2(k) \rceil \times 2^{256}.$$

In this case, the complexity is dominated by the cost of the multicollision.

Chapter 8 – Conclusions

This work has shown that pseudo-linear approximations may be applicable to round functions that are composed of addition modulo 2^n, exclusive-or, rotation, and permutations. When permutation is performed on the word level, approximations may be constructed that apply to windows with size less than or equal to the word size. In particular, pseudo-linear approximations trace dependencies of w-bit target windows throughout a function, using addition modulo 2^w to approximate addition modulo 2^n. Reducing n-bit word addition to w-bit word addition allows the adversary to reduce the number of key bits that need to be guessed, and thus reduces overall attack complexity. The carry function will either add one to a window or not, depending on the sum of preceding bits. The carry function is independent of the window size. By increasing the size of the window, the expected number of times an approximation holds at random decreases, while the number of times the approximation holds for the cipher remains unchanged. Hence, the bias increases. It is important to note that, as with traditional linear cryptanalysis, high diffusion and a large enough number of rounds can be used to cause approximations over an entire cipher to become infeasible. In this situation, too many key bits would typically need to be guessed, and the search space would swiftly approach that of a brute force search.

This work has also shown that the approximations may be used to find truncated differentials, and to analyze fixed-permutation ARX-based hash functions. In addition, the branch number metric has been revisited to provide a more focused idea of diffusion in functions that use large words.

It would be interesting to understand the generality of such attacks, and also the types of ciphers that are more and less resilient to these attacks. The following are among open questions that come from this work. First, a question arises that is

similar to that in linear cryptanalysis, where one computes the parity of some key bits from related input and output bits. Is there a simple function of the internal key bits in the pseudo-linear approximations of several ARX rounds? If such a function were found, and adversary need only to compute this function, rather than find the key bits themselves. Second, how can one improve carry pattern selection for different input distributions? Third, are there other properties of linearity that can be leveraged to improve bias? Fourth, are larger words more or less secure than smaller words? Fifth, can the symmetry hierarchy of CubeHash be used to mount an attack? Sixth, it would also be interesting to perform further investigation on the pseudo-linear branch number for ARX functions and determine whether it can effectively be used to create a strong ARX round function. Finally, although the symmetries in CubeHash provide a basis for attack, the unlikelihood of arriving at a symmetric state at random prevents the attacks from being feasible. It may be possible to use pseudo-linear analysis to find a way into these states, when a path exists with one message block. In particular, it would be interesting to see if there is a path from the initialization state into a symmetric state that could be easily found.

Bibliography

[1] Jean-Philippe Aumasson. Collision for CubeHash2/120-512. NIST mailing list (local link), 2008.

[2] Jean-Philippe Aumasson, Eric Brier, Willi Meier, María Naya-Plasencia, and Thomas Peyrin. Inside the hypercube. In Colin Boyd and Juan Manuel González Nieto, editors, *ACISP*, volume 5594 of *LNCS*, pages 202–213. Springer, 2009.

[3] Jean-Philippe Aumasson, Eric Brier, Willi Meier, Maria Naya-Plasencia, and Thomas Peyrin. Inside the hypercube. In *ACISP*, volume 5594 of *LNCS*, pages 202–213. Springer, 2009.

[4] Jean-Philippe Aumasson, Cagdas Calik, Willi Meier, Onur Ozen, Raphael C.-W. Phan, and Kerem Varici. Improved cryptanalysis of Skein. Cryptology ePrint Archive, Report 2009/438, 2009. Available at `http://eprint.iacr.org/`.

[5] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. SHA-3 proposal BLAKE. Submission to NIST, 2008.

[6] Thomas Baignères, Jacques Stern, and Serge Vaudenay. Linear cryptanalysis of non binary ciphers. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *Selected Areas in Cryptography*, volume 4876 of *LNCS*, pages 184–211. Springer, 2007.

[7] Daniel J. Bernstein. CubeHash specification (2.B.1). Submission to NIST (Round 2), 2009. Available at `http://cubehash.cr.yp.to/submission2/spec.pdf`.

[8] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. In *CRYPTO*, volume 537 of *LNCS*, pages 2–21. Springer, 1990.

[9] Alex Biryukov and David Wagner. Slide attacks. In Lars R. Knudsen, editor, *FSE*, volume 1636 of *LNCS*, pages 245–259. Springer, 1999.

[10] Alex Biryukov and David Wagner. Advanced slide attacks. In *Advances in Cryptology EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 589–606. Springer, 2000.

[11] Benjamin Bloom and Alan Kaminsky. Single block attacks and statistical tests on CubeHash. Cryptology ePrint Archive, Report 2009/407, 2009. Available at `http://eprint.iacr.org/`.

[12] Eric Brier, Shahram Khazaei, Willi Meier, and Thomas Peyrin. Attack for CubeHash-2/2 and collision for CubeHash-3/64. NIST mailing list, 2009.

[13] Eric Brier, Shahram Khazaei, Willi Meier, and Thomas Peyrin. Linearization framework for collision attacks: Application to CubeHash and MD6. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *LNCS*, pages 560–577. Springer, 2009.

[14] Eric Brier, Shahram Khazaei, Willi Meier, and Thomas Peyrin. Real collisions for CubeHash-4/48. NIST mailing list (local link), 2009.

[15] Eric Brier, Shahram Khazaei, Willi Meier, and Thomas Peyrin. Real collisions for CubeHash-4/64. NIST mailing list (local link), 2009.

[16] Eric Brier and Thomas Peyrin. Cryptanalysis of CubeHash. In Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud, editors, *ACNS*, volume 5536 of *LNCS*, pages 354–368, 2009.

[17] Florent Chabaud and Serge Vaudenay. Links between differential and linear cryptoanalysis. In *EUROCRYPT*, pages 356–365, 1994.

[18] Jiazhe Chen and Keting Jia. Improved related-key boomerang attacks on round-reduced Threefish-512. Cryptology ePrint Archive, Report 2009/526, 2009. Available at `http://eprint.iacr.org/`.

[19] Joo Yeon Cho and Josef Pieprzyk. Multiple modular additions and crossword puzzle attack on NLSv2. In Juan A. Garay, Arjen K. Lenstra, Masahiro Mambo, and René Peralta, editors, *ISC*, volume 4779 of *LNCS*, pages 230–248. Springer, 2007.

[20] Joan Daemen and Vincent Rijmen. *The Design of Rijndael*, chapter 9, pages 123–133. Springer, 2002.

[21] Wei Dai. Collisions for CubeHash1/45 and cubehash2/89. NIST mailing list, 2008.

[22] Niels Ferguson, Stefan Lucks, and Kerry A. McKay. Symmetric states and their structure: Improved analysis of CubeHash. In *Second SHA-3 Candidate Conference*, August 2010.

[23] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The Skein hash function family (version 1.0), October 2008. Available at `http://www.skein-hash.info/sites/default/files/skein.pdf`.

[24] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The Skein hash function family (version 1.1), November 2008. Available at `http://www.skein-hash.info/sites/default/files/skein1.1.pdf`.

[25] Niels Ferguson and Bruce Schneier. Cryptanalysis of Akelarre. In *Workshop on Selected Areas in Cryptography (SAC '97) Workshop Record*, pages 201–212. 1997.

[26] Danilo Gligoroski, Vlastimil Klima, Svein Johan Knapskog, Mohamed El-Hadedy, Jorn Amundsen, and Stig Frode Mjolsnes. Cryptographic hash function BLUE MIDNIGHT WISH. Submission to NIST (Round 2), 2009.

[27] Bahram Honary, editor. *The Wide Trail Design Strategy*, volume 2260 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2001.

[28] Antoine Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In Matthew K. Franklin, editor, *CRYPTO*, volume 3152 of *LNCS*, pages 306–316. Springer, 2004.

[29] Burton S. Kaliski Jr. and Matthew J. B. Robshaw. Linear cryptanalysis using multiple approximations. In *CRYPTO*, volume 839 of *LNCS*, pages 26–39. Springer, 1994.

[30] John Kelsey, Bruce Schneier, and David Wagner. Mod n cryptanalysis, with applications against RC5p and m6. In Lars R. Knudsen, editor, *FSE*, volume 1636 of *LNCS*, pages 139–155. Springer, 1999.

[31] Dmitry Khovratovich, Ivica Nikoli, and Christian Rechberger. Rotational rebound attacks on reduced skein. Cryptology ePrint Archive, Report 2010/538, 2010.

[32] Dmitry Khovratovich and Ivica Nikolic. Rotational cryptanalysis of ARX. In *FSE 2010*, LNCS. Springer, 2010.

[33] Dmitry Khovratovich, Ivica Nikolic, and Ralf-Philipp Weinmann. Preimage attack on CubeHash512-r/4 and CubeHash512-r/8. NIST mailing list, 2008.

[34] Lars R. Knudsen. Truncated and higher order differentials. In *FSE*, volume 1008 of *LNCS*, pages 196–211. Springer, 1994.

[35] Lars R. Knudsen and David Wagner. Integral cryptanalysis. In Joan Daemen and Vincent Rijmen, editors, *FSE*, volume 2365 of *Lecture Notes in Computer Science*, pages 112–127. Springer, 2002.

[36] X. Lai and R. E. Blahut. Higher order derivatives and differential cryptanalysis. In *Proc. Symp. Communication, Coding and Cryptography*, pages 227 – 233. Kluwer, 1994.

[37] Susan K. Langford and Martin E. Hellman. Differential-linear cryptanalysis. In *CRYPTO*, volume 839 of *LNCS*, pages 17–25. Springer, 1994.

[38] Helger Lipmaa and Shiho Moriai. Efficient algorithms for computing differential properties of addition. In Mitsuru Matsui, editor, *FSE*, volume 2355 of *LNCS*, pages 336–350. Springer, 2001.

[39] Helger Lipmaa, Johan Wallén, and Philippe Dumas. On the additive differential probability of exclusive-or. In Bimal K. Roy and Willi Meier, editors, *FSE*, volume 3017 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2004.

[40] Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In Tor Helleseth, editor, *EUROCRYPT*, volume 765 of *LNCS*, pages 386–397. Springer, 1993.

[41] Kerry A. McKay and Poorvi L. Vora. Pseudo-linear approximations for ARX ciphers with application to Threefish. In *Second SHA-3 Candidate Conference*, August 2010.

[42] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*, chapter Chapter 9: Hash Functions and Data Integrity, pages 321–383. CRC Press, 1996.

[43] NIST. FIPS pub 46-2: Data encryption standard (DES), December 1993. Available at `http://www.itl.nist.gov/fipspubs/fip46-2.htm`.

[44] NIST. FIPS 46-3: Data encryption standard (DES), October 1999. Available at `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`.

[45] NIST. FIPS 197: Announcing the advanced encryption standard (AES), November 2001. Available at `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[46] NIST. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family, 2007. Available at `http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf`.

[47] Kaisa Nyberg. Linear approximation of block ciphers. In *EUROCRYPT*, pages 439–444, 1994.

[48] Kaisa Nyberg. Correlation theorems in cryptanalysis. *Discrete Applied Mathematics*, 111(1-2):177–188, 2001.

[49] Kaisa Nyberg and Johan Wallén. Improved linear distinguishers for snow 2.0. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *LNCS*, pages 144–162. Springer, 2006.

[50] Souradyuti Paul and Bart Preneel. Solving systems of differential equations of addition. In Colin Boyd and Juan Manuel González Nieto, editors, *ACISP*, volume 3574 of *Lecture Notes in Computer Science*, pages 75–88. Springer, 2005.

[51] Ronald L. Rivest. The RC5 encryption algorithm. In *FSE*, LNCS, pages 86–96. Springer, 1994.

[52] C. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal, Vol 28*, pages 656 – 715, October 1949.

[53] Othmar Staffelbach and Willi Meier. Cryptographic significance of the carry for ciphers based on integer addition. In Alfred Menezes and Scott A. Vanstone, editors, *CRYPTO*, volume 537 of *LNCS*, pages 601–614. Springer, 1990.

[54] Johan Wallén. Linear approximations of addition modulo 2^n. In *Fast Software Encryption 2003*, volume 2887 of *LNCS*, pages 261–273. Springer-Verlag, 2003.

[55] Johan Wallén. Linear approximations of addition modulo 2^{n}. In Thomas Johansson, editor, *FSE*, volume 2887 of *LNCS*, pages 261–273. Springer, 2003.

[56] Johan Wallén. On the differential and linear properties of addition. Research Report A84, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 2003.

[57] David J. Wheeler and Roger M. Needham. TEA, a tiny encryption algorithm. In *FSE*, LNCS, pages 363–366. Springer, 1994.

[58] Hongjun Wu and Bart Preneel. Cryptanalysis of the stream cipher ABC v2. In Eli Biham and Amr M. Youssef, editors, *Selected Areas in Cryptography*, volume 4356 of *LNCS*, pages 56–66. Springer, 2006.