Linearization Framework for Collisions in Hash Functions

Willi Meier

University of Applied Sciences Northwestern Switzerland
School of Engineering

Contents

- Hash functions; Overview
- Linear differential cryptanalysis
- How to find conforming message pairs efficiently
- Application: Collisions in reduced-round CubeHash (and MD6)
- Joint work with E. Brier, S. Khazaei and Thomas Peyrin

Hash functions: Overview

A few Notions

A hash function maps a variable-length input to a fixed-size output or fingerprint, of length n bits.

Requirements:

- Collision resistance: Can't find two inputs with a same fingerprint.
- Pre-image resistance: Given a fingerprint, it is infeasible to find a corresponding input.
- Second pre-image resistance: Given an input, can't find a second input with same fingerprint.

Requirements (quantified)

Can find collisions by birthday paradox in $2^{n/2}$ time and memory for any hash function if fingerprint size is n bits.

- Collision resistance: Can't find two inputs with a same fingerprint in (much) less than $2^{n/2}$.
- Pre-image resistance: Given a fingerprint, it is infeasible to find a corresponding input in less than 2^n.
- Second pre-image resistance: Given an input, can't find a second input with same fingerprint in less than 2^n.

Applications of hash functions

- Electronic (digital) signatures
- Secure storage of passwords
- Generation of pseudo-random numbers
- Payment schemes
- Commitments

Hash function constructions

With or without a key.

Keyed constructions:

- Message Authentication Codes (MAC)
- Universal hash functions

Keyless construction:

- Compression function
- Extension method:
  - Tree
  - Merkle-Damgard
  - Sponge

Compression function

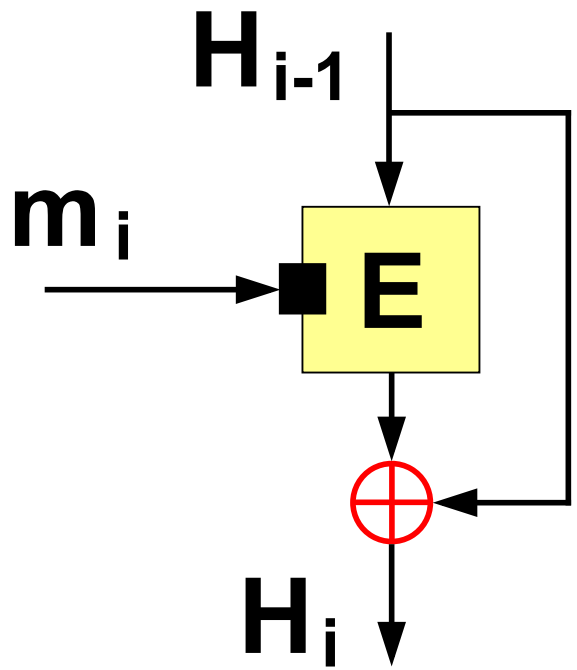
- Input of fixed size
- Processes one message block at a time
- Outputs are inputs for next iteration

Compression function constructions

No efficient provably secure compression function in sight.

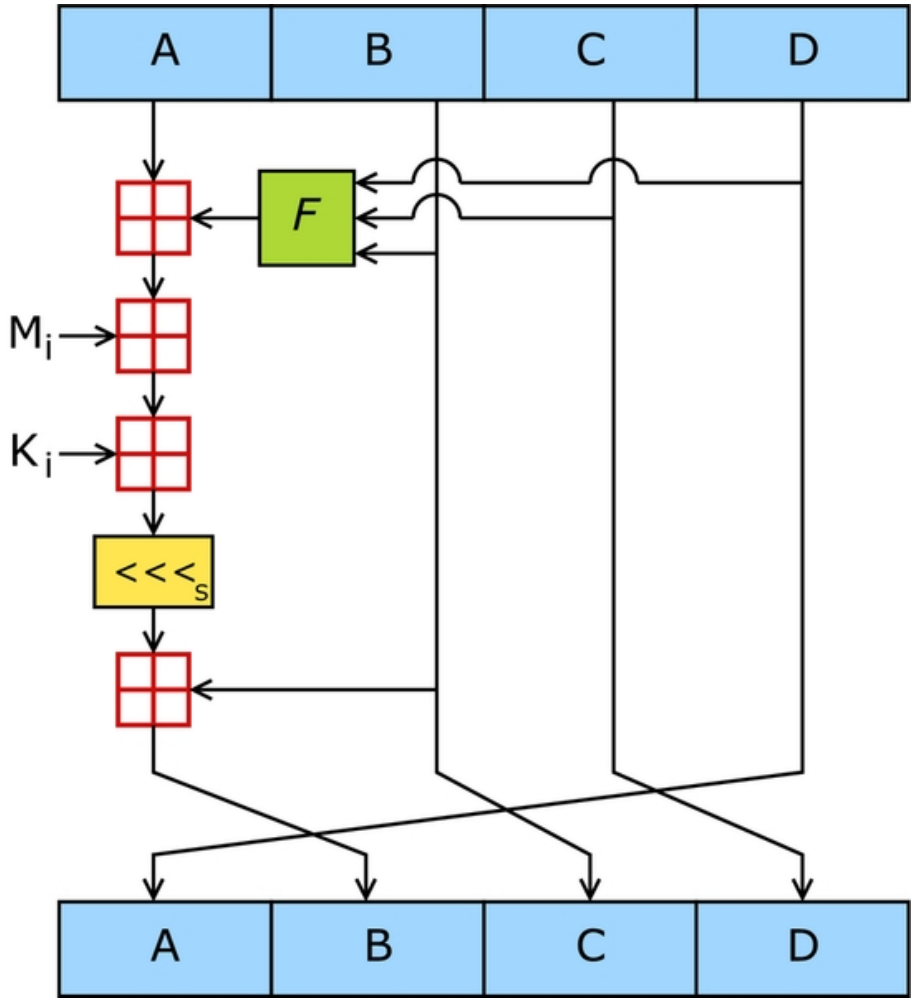
- Based on block cipher, e.g., Davies-Meyer
- Dedicated designs: MD5, SHA-1.
  - In worldwide use
  - Both broken in 2005 by X. Wang et. al.

Davies-Meyer mode



Pictures: Wikipedia

MD5 step function



MD5 compression function

MD5 has 4 rounds of 16 steps.

Chaining variable of 4 32-bit words.

Message input: 512 bits divided into 16 32-bit words.

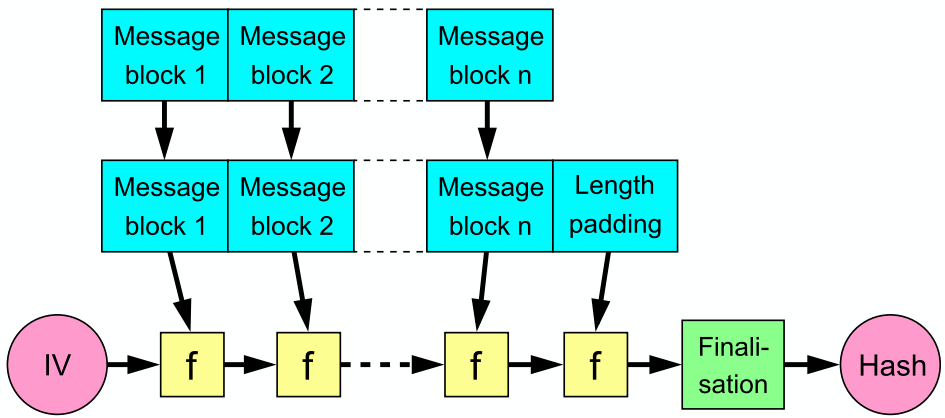
Each step processes one message word in prescribed order.

Uses as basic operations:

- xor
- integer addition of 32-bit words
- For each round a Boolean function with 3 inputs

Always acting on 32 bits in parallel. Extension to arbitrary length messages with Merkle-Damgard.

Merkle-Damgard construction



The SHA-3 competition

SHA-3

- October 2008: NIST received 64 submissions worldwide.
- December 2008: 51 submissions accepted to be in first round.
- July 2009: 14 candidates to be in round 2.
- Motivated a number of cryptanalytic attempts and breaks of several candidates.

Linear differentials

The principle

- Linearize compression function to find low weight differential characteristics.
- Find conforming message pair whose differential trail follows the linear one.
- Initiated by Chabaud-Joux for SHA-0 hash function.
- Main focus: compression functions composed only of integer addition and linear transforms, AXR (addition-xor-rotation), e.g. SHA-3 candidates BLAKE, BMW, CubeHash, Skein.

Compress and $\text{Compress}_{\text{lin}}$

Compression function $H = \text{Compress}(M, V)$.

Works with n-bit words, and maps m-bit message M and v-bit initial value V into h-bit output H.

For Compress look for two messages with a difference Δ that results in collision with some probability for random V.

Compress AXR-based: Replace all additions by XOR. Get a linear function $\text{Compress}_{\text{lin}}$. Then

$$\text{Compress}_{\text{lin}}(M, V) \oplus \text{Compress}_{\text{lin}}(M \oplus \Delta, V) = \text{Compress}_{\text{lin}}(\Delta, 0)$$

is independent of the value of V.

Notation: $\text{Compress}_{\text{lin}}(M) = \text{Compress}_{\text{lin}}(M, 0)$.

Let Δ be an element of the kernel of $\text{Compress}_{\text{lin}}$, *i.e.* $\text{Compress}_{\text{lin}}(\Delta) = 0$.

Are interested in the probability

$$\Pr\{\text{Compress}(M, V) \oplus \text{Compress}(M \oplus \Delta, V) = 0\}$$

for a random M and V.

Present algorithm which computes this probability, called the *raw* (or *bulk*) *probability*.

Notations

n_{add}: number of additions in Compress.

$A^i = A^i(M, V)$, $B^i = B^i(M, V)$, $1 \leq i \leq n_{add}$: addends of i-th addition.

In Compress$_{lin}(\Delta)$: $\alpha^i(\Delta)$, $\beta^i(\Delta)$: two inputs of i-th (linearized) addition.

Define four more functions $\mathbf{A}(M, V)$, $\mathbf{B}(M, V)$, $\boldsymbol{\alpha}(\Delta)$ and $\boldsymbol{\beta}(\Delta)$ with $(n-1)n_{\mathrm{add}}$-bit outputs.

Are the concatenation of all n_{add} relevant words excluding their MSBs,

e.g., $\mathbf{A}(M, V)$ and $\boldsymbol{\alpha}(\Delta)$ are respectively the concatenation of the n_{add} words $\left(A^1(M, V), \ldots, A^{n_{\mathrm{add}}}(M, V)\right)$ and $\left(\alpha^1(\Delta), \ldots, \alpha^{n_{\mathrm{add}}}(\Delta)\right)$ excluding the MSBs.

Computing raw probability

Assume n-bit integers A and B. Then

$$\Pr\{\big((A\oplus\alpha)+(B\oplus\beta)\big)\oplus(A+B)=\alpha\oplus\beta\}=2^{-\mathrm{wt}\left((\alpha\vee\beta)\wedge(2^{n-1}-1)\right)}.$$

This gives probability that modular addition behaves like XOR operation.

$\mathrm{Compress}_{\mathrm{lin}}$ approximates Compress by replacing modular addition with XOR.

Devise simple algorithm to compute (estimate) the raw probability

$$\Pr\{\mathrm{Compress}(M,V)\oplus\mathrm{Compress}(M\oplus\Delta,V)=\mathrm{Compress}_{\mathrm{lin}}(\Delta)\}.$$

Raw Probability

For any message difference Δ and for random values M and V,

$$p_\Delta = 2^{-\mathrm{wt}\left(\alpha(\Delta) \vee \beta(\Delta)\right)}$$

is an upper bound for

$$\Pr\{\mathrm{Compress}(M, V) \oplus \mathrm{Compress}(M \oplus \Delta, V) = \mathrm{Compress}_{\mathrm{lin}}(\Delta)\}.$$

A message M (for a given V) conforms to (or follows) the trail of Δ iff

$$\left((A^i \oplus \alpha^i) + (B^i \oplus \beta^i)\right) \oplus (A^i + B^i) = \alpha^i \oplus \beta^i, \text{ for } 1 \leq i \leq n_{\mathrm{add}}.$$

Finding a conforming message pair efficiently

Methods

- Message modification
- Neutral bits
- Boomerang attacks
- ....

Reformulate finding of conforming message pairs as finding preimages of zero of a function, called condition function.

Analyze condition function to see how freedom degrees can be used in efficient preimage construction.

Measure amount of influence of each input bit to each output bit of condition function.

Methods contd.

Dependency table: to distinguish influental bits from those with marginal influence.

In case the condition function doesn't mix its inputs well, exploit neutral bits and even probabilistic neutral bits.

Backtracking search algorithm based on dependency table.

Methods contd.

Assume that message difference Δ satisfies $\text{Compress}_{lin}(\Delta) = 0$.

Need to try around $1/p_\Delta$ random message pairs to find collision conforming to trail of Δ.

Method for finding conforming message pair sooner?

Condition function

Assume we found a differential path for message difference Δ, with prob. $p_\Delta = 2^{-y}$, where $y = \mathrm{wt}\big(\alpha(\Delta) \vee \beta(\Delta)\big)$.

Show that for given initial value V, problem of finding conforming message pair such that

$$\text{Compress}(M, V) \oplus \text{Compress}(M \oplus \Delta, V) = 0$$

can be translated into finding a message M such that

$$\text{Condition}_\Delta(M, V) = 0.$$

Here $Y = \text{Condition}_\Delta(M, V)$ is a function which maps m-bit message M and v-bit initial value V into y-bit output Y.

In other words, problem is reduced to finding a preimage of zero under the $\text{Condition}_{\Delta}$ function.

Depending on difference Δ, it may happen that not every output bit of the Condition function depends on all the message input bits.

By taking good strategy, this property enables to find the preimages of zero under this function more efficiently than random search.

In order to derive the function Condition from Compress in AXR-case, need a basic property of integer addition.

Nonlinearity is introduced by the carry word C when two integer words A and B are added:

$$C = (A + B) \oplus A \oplus B.$$

Let A and B be two n-bit words and C represent their carry word.

Let $\delta = 2^i$ for $0 \leq i \leq n-2$. Then,

$$\big((A \oplus \delta) + (B \oplus \delta)\big) = (A + B) \Leftrightarrow A_i \oplus B_i \oplus 1 = 0 \ ,$$

$$\big(A + (B \oplus \delta)\big) = (A + B) \oplus \delta \Leftrightarrow A_i \oplus C_i = 0 \ ,$$

and similarly

$$\big((A \oplus \delta) + B\big) = (A + B) \oplus \delta \Leftrightarrow B_i \oplus C_i = 0 \ .$$

Define the function $Y = \text{Condition}_{\Delta}(M, V)$ as:

$$Y_j = \begin{cases} \mathbf{A}_{i_j} \oplus \mathbf{B}_{i_j} \oplus 1 & \text{if} \quad (\alpha_{i_j}, \beta_{i_j}) = (1,1), \\ \mathbf{A}_{i_j} \oplus \mathbf{C}_{i_j} & \text{if} \quad (\alpha_{i_j}, \beta_{i_j}) = (0,1), \\ \mathbf{B}_{i_j} \oplus \mathbf{C}_{i_j} & \text{if} \quad (\alpha_{i_j}, \beta_{i_j}) = (1,0), \end{cases}$$

Then for a given V and Δ, a message M conforms to the trail of Δ iff $\text{Condition}_{\Delta}(M, V) = 0$.

Notation:

Dependency table

Notation: $F(M, V) = \text{Condition}_{\Delta}(M, V)$.

Given a general function $Y = F(M, V)$, which maps m message bits and v initial value bits into y output bits.

Goal: Reconstruct preimages of given output, e.g., zero vector, efficiently, i.e., find V and M such that $F(M, V) = 0$.

If diffusion of highly nonlinear function F is good, expect to try about 2^y random inputs.

In special cases: Not every input bit of F affects every output bit.

Ideal situation: Message bits and output bits can be divided into ℓ and$\ell + 1$ disjoint subsets, resp., as

$$\bigcup_{i=1}^{\ell} \mathcal{M}_i \text{ and } \bigcup_{i=0}^{\ell} \mathcal{Y}_i$$

such that the output bits $\mathcal{Y}_j$ $(0 \leq j \leq \ell)$ only depend on the input bits $\bigcup_{i=1}^{j} \mathcal{M}_i$ and the initial value V.

Once initial value V is known, can determine the output part $\mathcal{Y}_0$.

If initial value V and the input portion $\mathcal{M}_1$ is known, the output part $\mathcal{Y}_1$ is also known and so on.

This property suggests backtracking search algorithm.

Implemented recursively with tree-based search to avoid memory requirements: Algorithm 1.

Algorithm 1 : Preimage finding

Require: $q_0, q_1, \ldots, q_\ell$

Ensure: some preimage of zero under F

0: Choose 2^{q_0} initial values at random and keep those $2^{q'_1}$ candidates which make $\mathcal{Y}_0$ part null.

1: For each candidate, choose $2^{q_1 - q'_1}$ values for $\mathcal{M}_1$ and keep those $2^{q'_2}$ ones making $\mathcal{Y}_1$ null.

2: For each candidate, choose $2^{q_2 - q'_2}$ values for $\mathcal{M}_2$ and keep those $2^{q'_3}$ ones making $\mathcal{Y}_2$ null.

$\vdots$

i: For each candidate, choose $2^{q_i - q'_i}$ values for $\mathcal{M}_i$ and keep those $2^{q'_{i+1}}$ ones making $\mathcal{Y}_i$ null.

$\vdots$

ℓ: For each candidate, choose $2^{q_\ell - q'_\ell}$ values for $\mathcal{M}_\ell$ and keep those $2^{q'_{\ell+1}}$ final candidates making $\mathcal{Y}_\ell$ null.

Values $q_0, .., q_\ell$: To be determined in optimal way.

Complexity of algorithm (discussion):

Let $|\mathcal{M}_i|$ and $|\mathcal{Y}_i|$ denote the cardinality of $\mathcal{M}_i$ and $\mathcal{Y}_i$ resp., where $|\mathcal{Y}_0| \geq 0$ and $|\mathcal{Y}_i| \geq 1$ for $1 \leq i \leq \ell$.

Consider an ideal behavior of F for which each output part depends in a complex way on all the variables that it depends on.

Thus, the output segment changes independently and uniformly at random if we change any part of the relevant input bits.

Analysis of Algo: need to determine optimal values for $q_0, ..., q_\ell$.

Time complexity of Algo: $\sum_{i=0}^{\ell} 2^{q_i}$, as at each step 2^{q_i} values are examined.

Algo successful if at least one candidate left at the end, *i.e.* $q'_{\ell+1} \geq 0$.

Have $q'_{i+1} \approx q_i - |\mathcal{Y}_i|$, coming from the fact that at the i-th step 2^{q_i} values are examined each of which makes the portion $\mathcal{Y}_i$ of the output null with probability $2^{-|\mathcal{Y}_i|}$.

We have the restrictions $q_i - q'_i \leq |\mathcal{M}_i|$ and $0 \leq q'_i$ since we have $|\mathcal{M}_i|$ bits of freedom degree at the i-th step and we require at least one surviving candidate after each step.

Consequence: Optimal values for q_i's can be recursively computed as $q_{i-1} = |\mathcal{Y}_{i-1}| + \max(0, q_i - |\mathcal{M}_i|)$ for $i = \ell, \ell - 1, \ldots, 1$ with $q_\ell = |\mathcal{Y}_\ell|$.

How to determine the partitions $\mathcal{M}_i$ and $\mathcal{Y}_i$ for a given function F?

Heuristic method for determining the message and output partitions in practice:

First construct a $y \times m$ binary valued table T called dependency table.

The entry $T_{i,j}$, $0 \leq i \leq m-1$ and $0 \leq j \leq y-1$, is set to one iff the j-th output bit is highly affected by the i-th message bit.

To this end: empirically measure the probability that changing the i-th message bit changes the j-th output bit.

Probability computed over random initial values and messages.

Then set $T_{i,j}$ to one iff this probability is greater than a threshold $0 \leq th < 0.5$, for example $th = 0.3$.

Input/Output partitioning of a Condition function

Dependency table ...

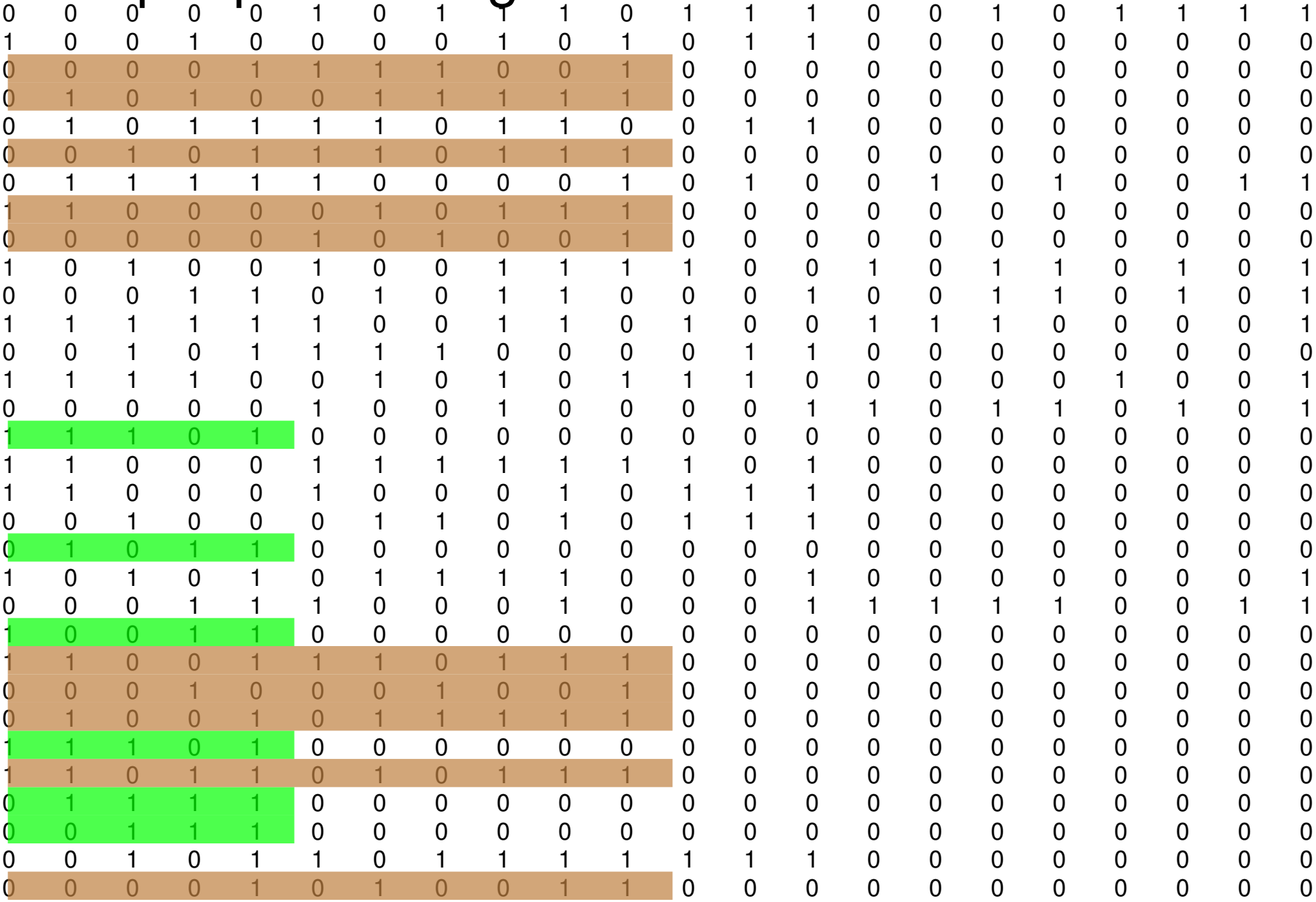
Input/Output partitioning of a Condition function

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

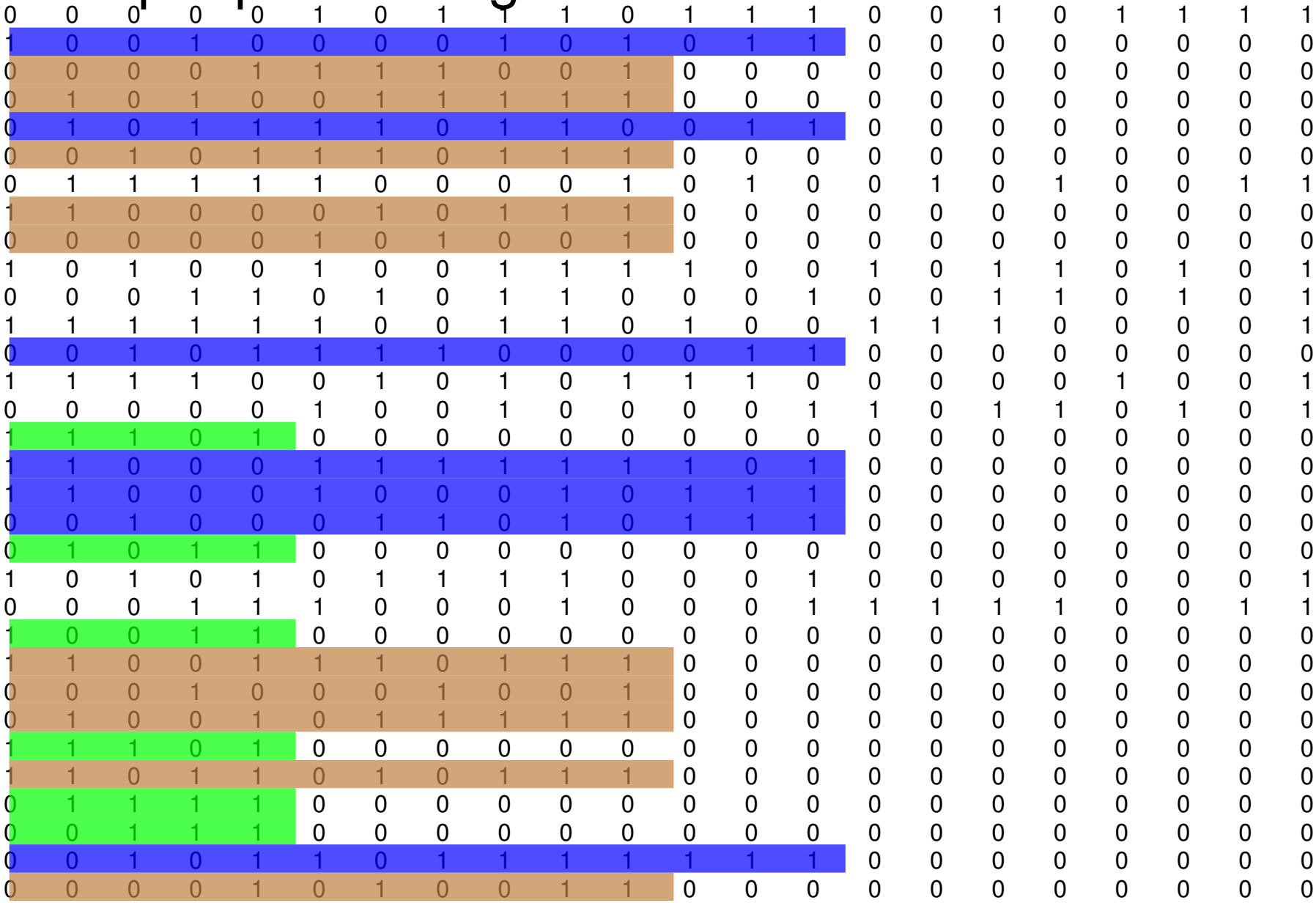
Input/Output partitioning of a Condition function

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

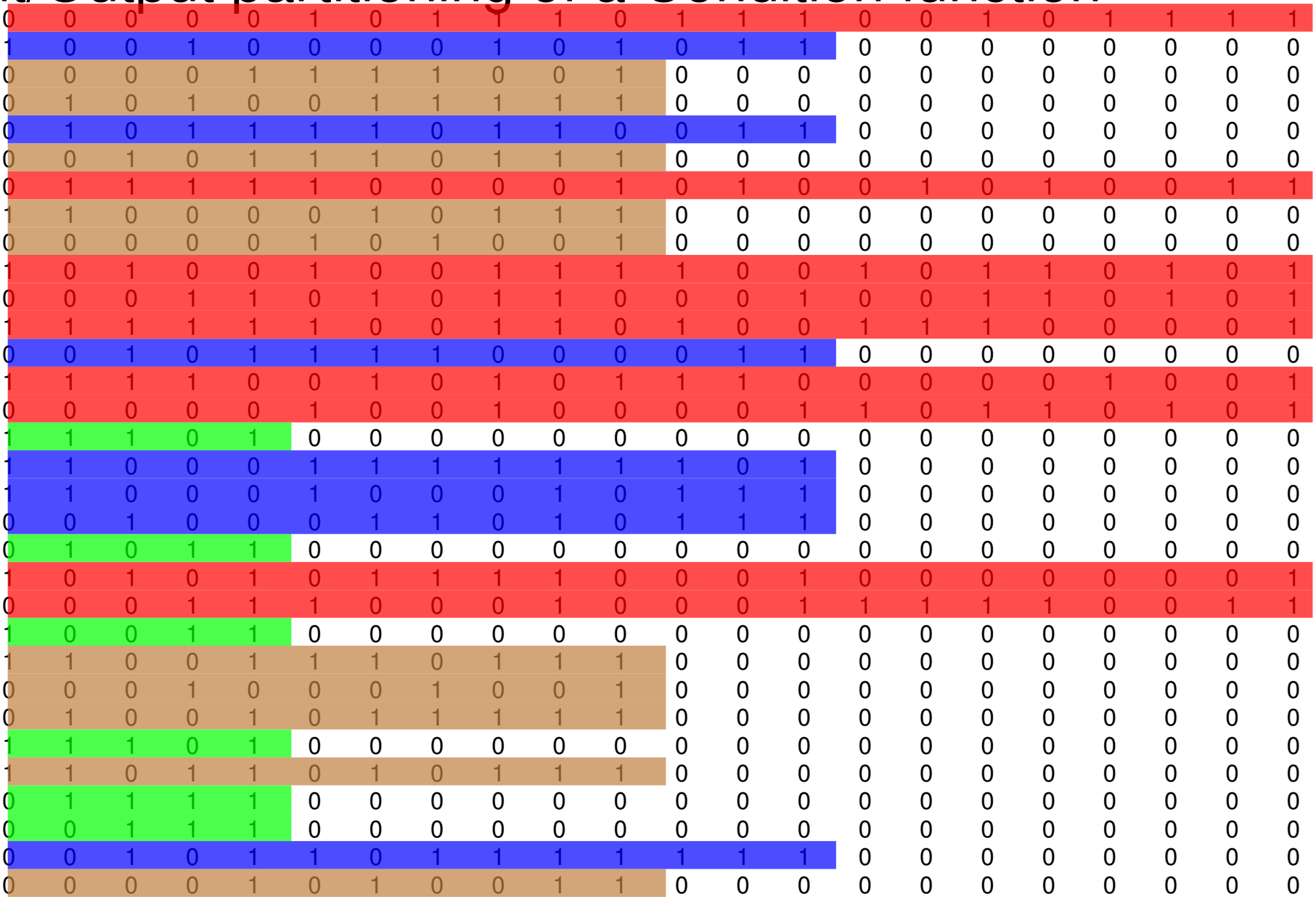
Input/Output partitioning of a Condition function



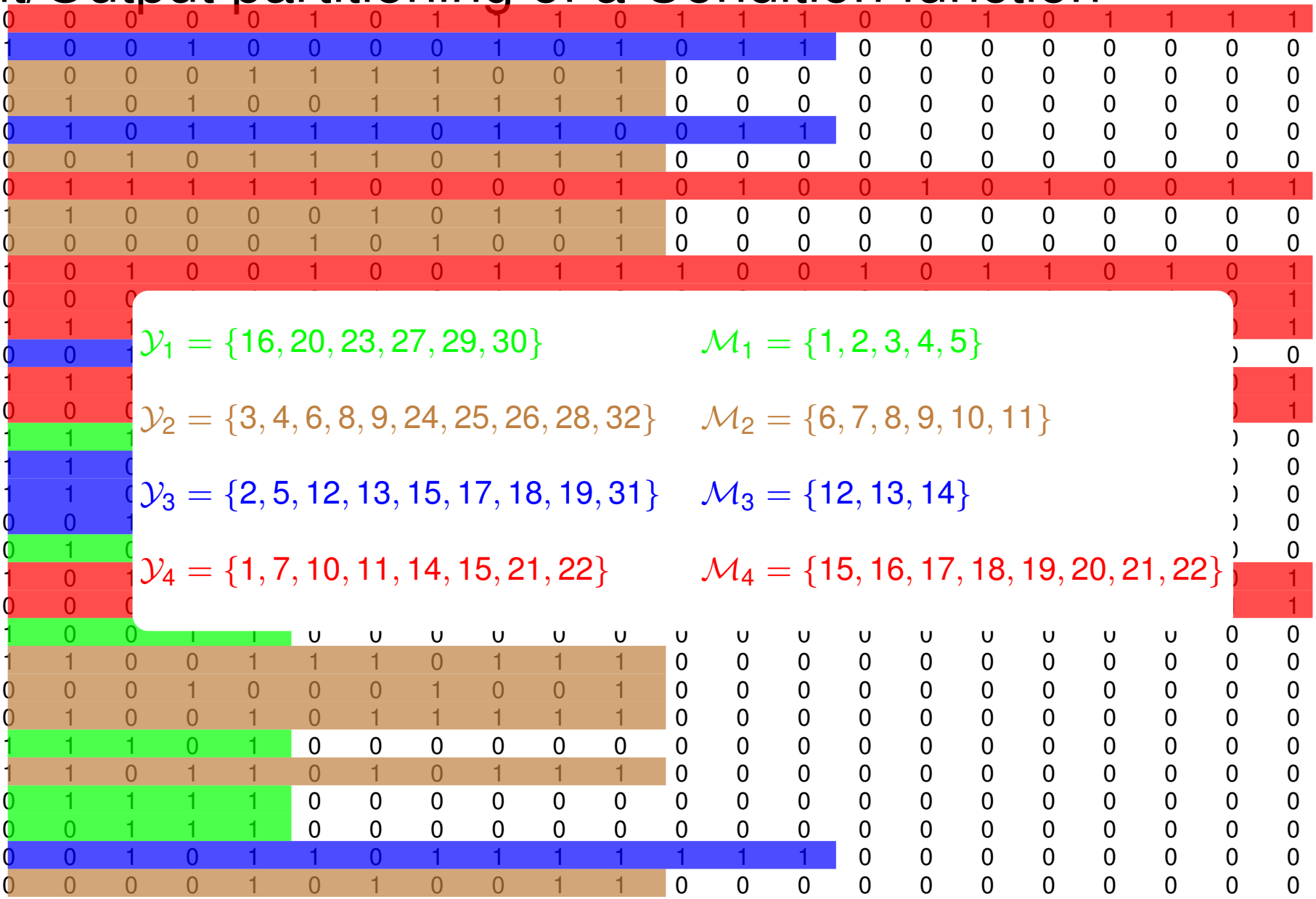
Input/Output partitioning of a Condition function



Input/Output partitioning of a Condition function



Input/Output partitioning of a Condition function



Call Algorithm 2, whose output is dependency table T., i.e., a number ℓ, message partitions $\mathcal{M}_1, \ldots, \mathcal{M}_\ell$ and output partitions $\mathcal{Y}_0, \ldots, \mathcal{Y}_\ell$.

Preparation: Put all the output bits j in $\mathcal{Y}_0$ for which the row j of T is all-zero, and delete all the all-zero rows from T.

Algorithm 2 : Message and output partitioning

1: $\ell := 0$;
2: **while** T is not empty **do**
3: $\quad \ell := \ell + 1$;
4: $\quad$ **repeat**
5: $\quad\quad$ Determine the column i in T which has the highest number of 1's and delete it from T.
6: $\quad\quad$ Put the message bit which corresponds to the deleted column i into the set $\mathcal{M}_\ell$.
7: $\quad$ **until** There is at least one all-zero row in T OR T becomes empty
8: $\quad$ If T is empty set $\mathcal{Y}_\ell$ to those output bits which are not in $\bigcup_{i=0}^{\ell-1} \mathcal{Y}_i$ and stop.
9: $\quad$ Put all the output bits j in $\mathcal{Y}_\ell$ for which the corresponding row of T is all-zero.
10: $\quad$ Delete all the all-zero rows from T.
11: **end while**

In practice, deviation from assumed ideal behaviour:

1. The message segments $\mathcal{M}_1, \ldots, \mathcal{M}_i$ do not have full influence on $\mathcal{Y}_i$,
2. The message segments $\mathcal{M}_{i+1}, \ldots, \mathcal{M}_\ell$ have influence on $\mathcal{Y}_0, \ldots, \mathcal{Y}_i$.

First issue:

Would like that all the message segments $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_i$ as well as the initial value V have full influence on the output part $\mathcal{Y}_i$.

However, effect of the last few message segments $\mathcal{M}_{i-d_i}, \ldots, \mathcal{M}_i$ (for some small integer d_i) is more important, though.

Some tweaks on the tree-based (backtracking) search algorithm allow to overcome this effect in practice, e.g.:

If message segment $\mathcal{M}_{i-1}$ does have only small influence on the output segment $\mathcal{Y}_i$, we may decide to backtrack two steps at depth i, instead of one (the default value).

Reason is as follows: Imagine that we are at depth i of the tree and are trying to adjust the i-th message segment $\mathcal{M}_i$, to make the output segment $\mathcal{Y}_i$ null.

If after trying about $2^{\min(|\mathcal{M}_i|,|\mathcal{Y}_i|)}$ choices for the i-th message block, we don't find an appropriate one, go one step backward and choose another choice for the $(i-1)$-st message segment $\mathcal{M}_{i-1}$;

Then go one step forward once we have successfully adjusted the $(i-1)$-st message segment.

If $\mathcal{M}_{i-1}$ has no effect on $\mathcal{Y}_i$, this would be useless and increase our search cost at this node.

Hence appropriate to backtrack two steps at this depth.

In general: Tweak tree-based search by setting the number of steps which we want to backtrack at each depth.

Second issue:

Ideally, we would like that the message segments $\mathcal{M}_i, \ldots, \mathcal{M}_\ell$ have no influence on the output segments $\mathcal{Y}_0, \ldots, \mathcal{Y}_{i-1}$.

The smaller the threshold value *th* is chosen, the less the influence would be.

Let 2^{-p_i}, $1 \leq i \leq \ell$, denote the probability that changing the message segment $\mathcal{M}_i$ does not change any bit from the output segments $\mathcal{Y}_0, \ldots, \mathcal{Y}_{i-1}$.

The probability is computed over random initial values and messages, and a random non-zero difference in the message segment $\mathcal{M}_i$.

Algorithm 1 must be reanalyzed in order to recompute the optimal values for $q_0, \ldots, q_\ell$.

Algorithm 1 also needs to be slightly changed by reassuring that at step i, all the output segments $\mathcal{Y}_0, \ldots, \mathcal{Y}_{i-1}$ remain null.

The time complexity of the algorithm is still $\sum_{i=0}^{\ell} 2^{q_i}$ and it is successful if at least one surviving candidate is left at the end, *i.e.* $q_{\ell+1} \geq 0$.

However, here we set $q'_{i+1} \approx q_i - |\mathcal{Y}_i| - p_i$.

This comes from the fact that:

1. at the i-th step 2^{q_i} values are examined each of which makes the portion $\mathcal{Y}_i$ of the output null with probability $2^{-|\mathcal{Y}_i|}$
2. keeping the previously set output segments $\mathcal{Y}_0, \ldots, \mathcal{Y}_{i-1}$ null with probability 2^{-p_i} (we assume these two events are independent).

Here, our restrictions are again $0 \leq q'_i$ and $q_i - q'_i \leq |\mathcal{M}_i|$.

Hence, the optimal values for q_i's can be recursively computed as $q_{i-1} = p_{i-1} + |\mathcal{Y}_{i-1}| + \max(0, q_i - |\mathcal{M}_i|)$ for $i = \ell, \ell - 1, \ldots, 1$ with $q_\ell = |\mathcal{Y}_\ell|$.

Remark

When working with functions with a huge number of input bits, it might be appropriate to consider the m-bit message M as a string of u-bit units instead of bits.

For example one can take $u = 8$ and work with bytes. We then use the notation $M = (M[0], \ldots, M[m/u - 1])$ (assuming u divides m) where $M[i] = (M_{iu}, \ldots, M_{iu+u-1})$.

In this case the dependency table must be constructed according to the probability that changing every message unit changes each output bit.

Application: Collisions in reduced-round CubeHash

Cubehash: Description

CubeHash: Proposed by Dan Bernstein.

One of the 14 2nd-round SHA-3 candidates.

CubeHash variants: Denoted by CubeHash-r/b. Parametrized by r and b.

At each iteration, b message bytes are processed in r rounds.

CubeHash-8/1 was the original official submission.

Later the designer proposed the tweak CubeHash-16/32, which is almost 16 times faster than the initial proposal.

Nevertheless, the author has encouraged cryptanalysis of CubeHash-r/b variants for smaller r's and bigger b's.

CubeHash works with 32-bit words.

Uses only XOR, bitwise rotations of words, and addition mod 2^{32}.

Internal state $S = (S_0, S_1, \ldots, S_{31})$ of 32 words.

Parameters in CubeHash-r/b: $r \in \{1, 2, \ldots\}$ and $b \in \{1, 2, \ldots, 128\}$.

Internal state S is set to a specified value which depends on the digest length (limited to 512 bits) and parameters r and b.

Message to be hashed is appropriately padded and divided into b-byte message blocks.

At each iteration one message block is processed as follows.

The 32-word internal state S is considered as a 128-byte value.

The message block is XORed into the first b bytes of the internal state:

- 1st message byte is XORed into the least significant byte of S_0,
- 2nd byte into the second least significant byte of S_0,
- 3d byte into the third least significant byte of S_0,
- 4th byte into the most significant byte of S_0,
- 5th byte into the least significant byte of S_1, and so forth until all b message bytes have been exhausted.

Then, the following fixed permutation is applied r times to the internal state to prepare it for the next iteration.

Permutation of CubeHash

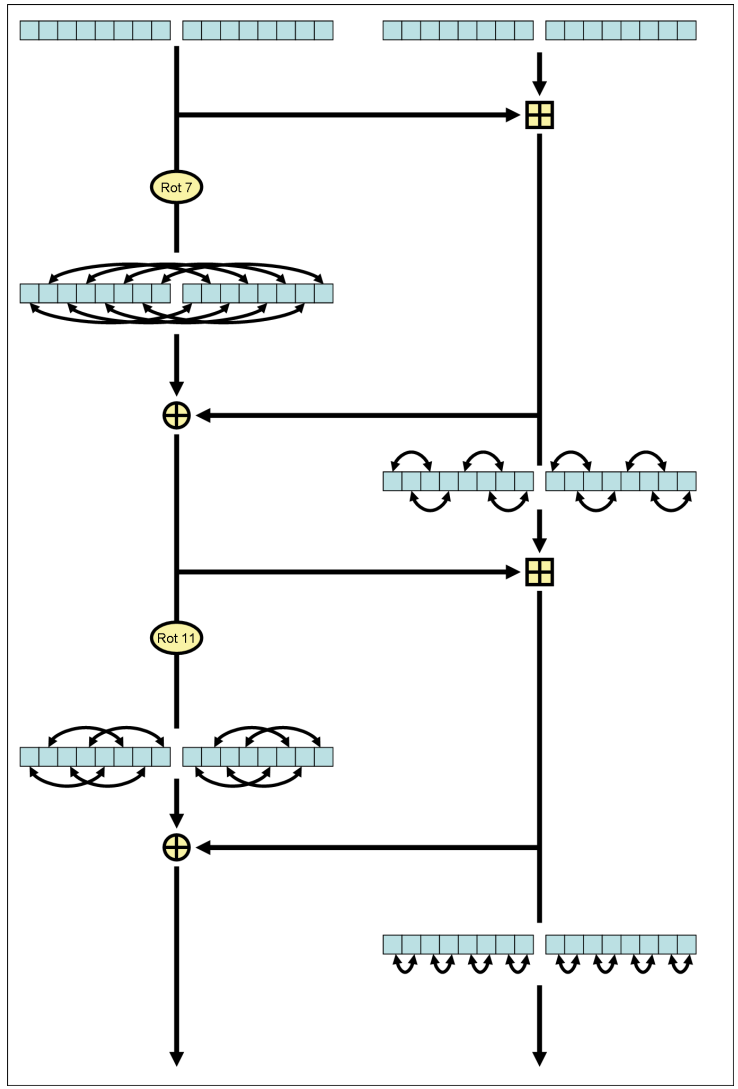
1. Add S_i into $S_{i \oplus 16}$, for $0 \leq i \leq 15$.
2. Rotate S_i to the left by seven bits, for $0 \leq i \leq 15$.
3. Swap S_i and $S_{i \oplus 8}$, for $0 \leq i \leq 7$.
4. XOR $S_{i \oplus 16}$ into S_i, for $0 \leq i \leq 15$.
5. Swap S_i and $S_{i \oplus 2}$, for $i \in \{16, 17, 20, 21, 24, 25, 28, 29\}$.
6. Add S_i into $S_{i \oplus 16}$, for $0 \leq i \leq 15$.
7. Rotate S_i to the left by eleven bits, for $0 \leq i \leq 15$.
8. Swap S_i and $S_{i \oplus 4}$, for $i \in \{0, 1, 2, 3, 8, 9, 10, 11\}$.
9. XOR $S_{i \oplus 16}$ into S_i, for $0 \leq i \leq 15$.
10. Swap S_i and $S_{i \oplus 1}$, for $i \in \{16, 18, 20, 22, 24, 26, 28, 30\}$.

Having processed all message blocks, a fixed transformation is applied to the final internal state to extract the hash value as follows.

First, the last state word S_{31} is ORed with integer 1 and then the above permutation is applied $10 \times r$ times to the resulting internal state.

Finally, the internal state is truncated to produce the message digest of desired hash length.

Permutation of CubeHash



Compression function of CubeHash

To conform with our general method, need to deal with fixed-size input compression function textcolormyblueCompress.

Consider t ($t \geq 1$) consecutive iterations of CubeHash.

Define the function $H = \text{Compress}(M, V)$ with an $8bt$-bit message $M = M^0 || \ldots || M^{t-1}$, a 1024-bit initial value V and a $(1024 - 8b)$-bit output H.

The initial value V is used to initialize the 32-word internal state of CubeHash.

Each M^i is a b-byte message block.

Start from the initialized internal state and update it in t iterations, i.e.,

in t iterations the t message blocks $M^0, \ldots, M^{t-1}$ are sequentially processed in order to transform the internal state into a final value.

The output H is then the last $128 - b$ bytes of the final internal state value which is ready to absorb the $(t+1)$-st message block (the 32-word internal state is interpreted as a 128-byte vector).

Goal: Find collisions for this Compress function.

Collision construction

Plan to construct collision pairs (M', M'') for CubeHash-r/b which are of the form

$$M' = M^{\mathrm{pre}} || M || M^t || M^{\mathrm{suf}}$$

and

$$M'' = M^{\mathrm{pre}} || M \oplus \Delta || M^t \oplus \Delta^t || M^{\mathrm{suf}}.$$

Here,

- M^{pre} is the common prefix of the colliding pairs whose length in bytes is a multiple of b,
- M^t is one message block of b bytes, and
- M^{suf} is the common suffix of the colliding pairs whose length is arbitrary.

The message prefix M^{pre} is chosen for randomizing the initial value V.

More precisely, V is the contents of the internal state after processing the message prefix M^{pre}.

For this value of V, $(M, M \oplus \Delta)$ is a collision pair for the compression function, *i.e.*

$$\mathrm{Compress}(M, V) = \mathrm{Compress}(M \oplus \Delta, V).$$

Recall that a collision for the Compress indicates collision over the last $128 - b$ bytes of the internal state.

The message blocks M^t and $M^t \oplus \Delta^t$ are used to get rid of the difference in the first b bytes of the internal state.

The difference Δ^t is called the *erasing block difference* and is computed as follows.

When Compress is evaluated with inputs (M, V) and $(M \oplus \Delta, V)$:

Δ^t is the difference in the first b bytes of the final internal state values.

Once we find message prefix M^{pre}, message M and difference Δ, any message pairs (M', M'') of the above-mentioned form is a collision for CubeHash for *any* message block M^t and *any* message suffix M^{suf}.

Find the difference Δ using the previous linearization method to be applied to CubeHash.

Then, M^{pre} and M are found by determining a preimage of zero under the Condition function.

Linear differentials for CubeHash

As previously seen, the linear transformation $\text{Compress}_{\text{lin}}$ can be described by a matrix $\mathcal{H}_{h \times m}$.

Are interested in Δ's such that $\mathcal{H}\Delta = 0$ and such that the differential trails have high probability.

For CubeHash-r/b with t iterations:

$$\Delta = \Delta^0 || \ldots || \Delta^{t-1}$$

and $\mathcal{H}$ has size $(1024 - 8b) \times 8bt$.

Fact: This matrix suffers from having low rank.

Enables to find low weight vectors of the kernel.

Hope that low weight vectors are also good candidates for providing highly probable trails.

Assume that matrix $\mathcal{H}_{h\times m}$ has rank $(8bt - \tau)$, $\tau \geq 0$.

Implies existence of $2^{\tau} - 1$ nonzero solutions to $\mathcal{H}\Delta = 0$.

To find a low weight nonzero Δ, we use the following method.

The rank of $\mathcal{H}$ being $(8bt - \tau)$:

Shows that the solutions can be expressed by identifying τ variables as free and expressing the rest in terms of them.

Any choice for the free variables uniquely determines the remaining $8bt - \tau$ variables, hence providing a unique member of the kernel.

Choose a set of τ free variables at random.

Thereafter, assign bit value 1 to one, two, or three of the τ free variables, and the other $\tau - 1$, or $\tau - 2$ or $\tau - 3$ variables to bit value

Hope to get a Δ providing a high-probability differential path.

Exhaustive search over all $\tau + \binom{\tau}{2} + \binom{\tau}{3}$ possible choices for all $b \in \{1, 2, 3, 4, 8, 16, 32, 48, 64\}$ and $r \in \{1, 2, 3, 4, 5, 6, 7, 8\}$ in order to find the best characteristics.

Next Table includes the ordered pair (t, y), *i.e.* the corresponding number of iterations and the $-\log_2$ probability (number of bit conditions) of the best raw probability path we found.

Table: Values of (t, y) for the differential path with the best found raw probability: Part I

$r \setminus b$	1	2	3	4	8
1	$(14, 1225)$	$(8, 221)^\star$	$(4, 46)$	$(4, 32)$	$(4, 32)$
2	$(7, 1225)$	$(4, 221)^\star$	$(2, 46)$	$(2, 32)$	$(2, 32)$
3	$(16, 4238)^\star$	$(6, 1881)$	$(4, 798)$	$(4, 478)^\star$	$(4, 478)^\star$
4	$(8, 2614)$	$(3, 964)$	$(2, 195)$	$(2, 189)$	$(2, 189)$
5	$(18, 10221)^\star$	$(8, 4579)$	$(4, 2433)$	$(4, 1517)$	$(4, 1517)$
6	$(10, 4238)$	$(3, 1881)$	$(2, 798)$	$(2, 478)$	$(2, 478)$
7	$(14, 13365)$	$(8, 5820)$	$(4, 3028)$	$(4, 2124)$	$(4, 2124)$
8	$(4, 2614)$	$(4, 2614)$	$(2, 1022)$	$(2, 1009)$	$(2, 1009)$

Table: Values of (t, y) for the differential path with the best found raw probability: Part II

$r \setminus b$	12	16	32	48	64
1	–	–	–	–	–
2	–	–	–	–	–
3	$(4, 400)^\star$	$(4, 400)^\star$	$(4, 400)^\star$	$(3, 364)^\star$	$(2, 65)$
4	$(2, 156)$	$(2, 156)$	$(2, 156)$	$(2, 130)$	$(2, 130)$
5	$(4, 1244)$	$(4, 1244)$	$(4, 1244)$	$(4, 1244)^\star$	$(2, 205)$
6	$(2, 400)$	$(2, 400)$	$(2, 400)$	$(2, 351)$	$(2, 351)$
7	$(4, 1748)$	$(4, 1748)$	$(4, 1748)$	$(4, 1748)^\star$	$(2, 447)$
8	$(2, 830)$	$(2, 830)$	$(2, 830)$	$(2, 637)$	$(2, 637)$

Comments:

In most cases, best characteristic belongs to the minimum value of t for which $\tau > 0$.

There are a few exceptions though, which are starred in previous Table, e.g.,

in the CubeHash3/4 case, while for $t = 2$ we have $\tau = 4$ and $y = 675$, by increasing the number of iterations to $t = 4$, we get $\tau = 40$ and a better characteristic with $y = 478$.

This may hold for other cases as well since we only increased t until our program terminated in a reasonable time.

Emphasize that since we are using linear differentials, the erasing block difference Δ^t only depends on the difference Δ.

Second preimage attacks on CubeHash

Any differential path with raw probability greater than 2^{-512} can be considered as a (theoretical) second preimage attack on CubeHash with 512-bit digest size.

In previous Table, the entries which correspond to such cases have been highlighted.

For example, the differential path found for CubeHash-6/4 with raw probability 2^{-478} indicates that by only one hash evaluation we can produce a second preimage with probability 2^{-478}.

Alternatively, it can be stated that for a fraction of 2^{-478} messages we can easily provide a second preimage.

Collision attacks on CubeHash variants

Aim: Find collisions using dependency table and condition function regarding a good differential.

Analyze condition function at byte level.

Reveals degrees of freedom that can be exploited in accelerated search of preimage of 0 of condition function.

Results in (much) improved collision search.

Best found differential path regarding raw probability does not always conform to best differential, when freedom degrees are used (optimized differential).

Next Table shows reduced time complexities of collision attack using these methods and optimized differentials regarding freedom degrees.

Successful instances, i.e., those with probability higher than 2^{-256}, are highlighted.

Table: Theoretical $\log_2$ complexities of improved collision attacks with freedom degrees use for optimized differential paths, Part I .

$r \setminus b$	1	2	3	4	8
1	1121.0	135.1	24.0	15.0	7.6
2	1177.0	179.1	27.0	17.0	7.9
3	4214.0	1793.0	720.0	380.1	292.6
4	2598.0	924.0	163.0	138.4	105.3
5	10085.0	4460.0	2345.0	1397.0	1286.0
6	4230.0	1841.0	760.6	422.1	374.4
7	13261.0	5709.0	2940.0	2004.0	1892.0
8	2606.0	2590.0	982.0	953.0	889.0

Table: Theoretical $\log_2$ complexities of improved collision attacks with freedom degrees use for optimized differential paths, Part II .

$r \setminus b$	12	16	32	48	64
1	–	–	–	–	–
2	–	–	–	–	–
3	153.5	102.0	55.6	53.3	9.4
4	67.5	60.7	54.7	30.7	28.8
5	946.0	868.0	588.2	425.0	71.7
6	260.4	222.6	182.1	147.7	144.0
7	1423.0	1323.0	978.0	706.0	203.0
8	699.0	662.0	524.3	313.0	304.4

Complexities given correspond to optimally chosen threshold value for computing the dependency table.

Real collisions

Use a 2-iteration message difference for CubeHash-3/64 and CubeHash-4/48.

For CubeHash-3/64 use relation (1) and for CubeHash-4/48 relation (2).

Include the erasing block difference Δ^2, required for collision construction.
Collisions with 512-bit digest size.

Differential path for CubeHash-3/64, $y = 65$:

$$
\begin{aligned}
\Delta^0 = \quad & \texttt{4000000000000000000000000000000000000000000000} \\
& \texttt{0000000000000000000080000000000000800000000} \\
& \texttt{0000000000000000000000000000000} \\
\Delta^1 = \quad & \texttt{0000000000410000000000000100000000000000000} \\
& \texttt{8000208000000080080002080800000000000000000} \\
& \texttt{4100001000000000140000000000000} \\
\Delta^2 = \quad & \texttt{0000000000000000000000000000000000000002000} \\
& \texttt{0000000000002000000000000000000000000000000} \\
& \texttt{0000000000000000000000000010}
\end{aligned}
\tag{1}
$$

Differential path for CubeHash-4/48, $y = 134$:

$$\begin{aligned}
\Delta^0 = \quad & 00000008000000000000008000000000000000000000 \\
& 00000000000000004000000 \\
\Delta^1 = \quad & 88008000000000088008000000000000000000000000 \\
& 00000000000000000440040 \\
\Delta^2 = \quad & 08000000000000008000000000000000000000000000 \\
& 00000000000000000040000
\end{aligned} \tag{2}$$

Collision for CubeHash-3/64:

$M^{\mathrm{pre}} =$ `9B91E97363511AC3AF950F54DBCFD5DF91BC26BDD759104D`
`F15B37847A4F7015E15A8844ABA3075A3816AE13E583F276`
`4019331772446464 9F9BE819EB582ECC`

$M^0 =$ `B22A98139CC0C8606525818EE6DD7775CF25B34196DC51F4`
`641E56ACB918296BBD082AD01D7481EECC950B6C176C45B6`
`23CFE1E2638B16255F61E806F34DE91C`

$M^1 =$ `4D9E9CD62ED12CBDBA1E0B631856DCFE5BD996571CFF6E94`
`A52242382E154FA6AEB44AC0A247CB298550C7B82BDCA924`
`E81D5E51E997CA67FBDD86FF15D04A0D`

Includes value of M^{pre} and $M = M^0||M^1$ for collision.

Collision for CubeHash-4/48:

$$M^{\mathrm{pre}} = \texttt{741B87597F94FF1CC01761CA0D80B07CC2E6E760C95DF9A5}$$
$$\texttt{08FFCBABDA11474E2CCEA7AC62A7C822BE29EDCBA99D476C}$$
$$M^0 = \texttt{1D30F8022F4AE8DBD477FA1F7DE37C1AF2516BC6FA4657F9}$$
$$\texttt{E51539C10EC114DA3B8264DD9361FE07C3D56E88E8512201}$$
$$M^1 = \texttt{014A11BFE2FF346FC306D1E430EE80268785A9F841562C9A}$$
$$\texttt{88A6BF5858E95362F541ACF41C2FDCC1C49470DF1DFAEFDC}$$

Includes value of M^{pre} and $M = M^0||M^1$ for collision.

Here $\text{Condition}_{\Delta}(M, V) = 0$ does hold for corresponding condition functions.

V is the content of the internal state after processing the message prefix M^{pre}.

The pair M' and M'' where

$$M' = M^{\text{pre}} || M^0 || M^1 || M^2 || M^{\text{suf}},$$

$$M'' = M^{\text{pre}} || (M^0 \oplus \Delta^0) || (M^1 \oplus \Delta^1) || (M^2 \oplus \Delta^2) || M^{\text{suf}},$$

collides for *any* message block M^2 and *any* message suffix M^{suf} of arbitrary length.

Note: A collision pair for a given r and b can be easily transformed to a collision pair for the same r and bigger b's by appending enough zeros to each message block.

Attack method generalizes to other hash functions, e.g., to practical collisions of 16-round MD6 (out of 80 rounds).

References

J.-P. Aumasson, S. Fischer, S. Khazaei, W. Meier, Ch. Rechberger: New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba. FSE 2008, pp. 470-488.

E. Brier, S. Khazaei, W. Meier, Th. Peyrin: Linearization Framework for Collision Attacks: Application to CubeHash and MD6, ASIACRYPT 2009. Also on Cryptology ePrint Archive, Report 2009/382.

F. Chabaud, A. Joux: Differential Colisions in SHA-0, CRYPTO 1998, pp. 56-71.