

# Cryptography and Cryptographic Hash Functions in Blockchain

**Presented BY: Dr Abdurashid Turgunov**

**Course name: Blockchain Technology**

# What are hash functions?

- Just a method of compressing strings
  - E.g.,  $H : \{0,1\}^* \rightarrow \{0,1\}^{160}$
  - Input is called “message”, output is “digest”
- Why would you want to do this?
  - Short, fixed-size better than long, variable-size
    - True also for non-crypto hash functions
  - Digest can be added for redundancy
  - Digest hides possible structure in message

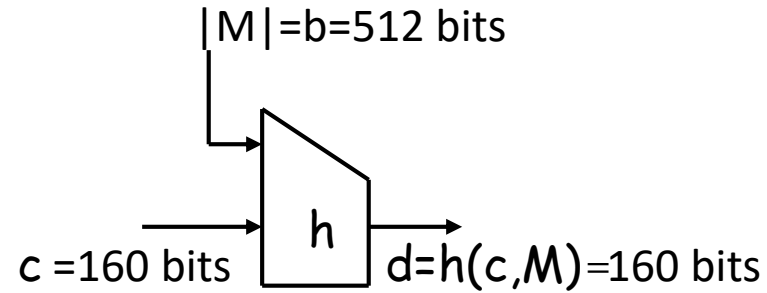
But not always...

# ... they built?

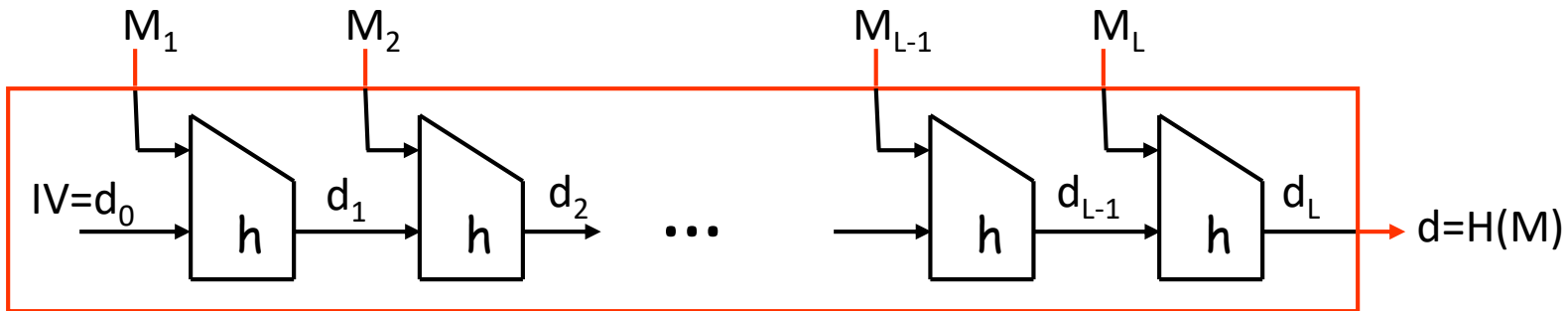
Typically using Merkle-Damgård iteration:

1. Start from a “compression function”

- $h: \{0,1\}^{b+n} \rightarrow \{0,1\}^n$



2. Iterate it



What are they good for?

**signature algorithm**

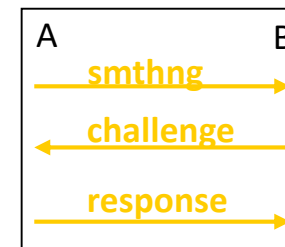
**digital**

**message authentication codes, pseudo  
random number generators key derivation functions.**

“Request for Candidate Algorithm Nominations”,  
-- NIST, November 2007

# Some examples

- Signatures:  $\text{sign}(M) = \text{RSA}^{-1}( H(M) )$
- Message-authentication:  $\text{tag} = H(\text{key}, M)$
- Commitment:  $\text{commit}(M) = H(M, \dots)$
- Key derivation:  $\text{AES-key} = H(\text{DH-value})$
- Removing interaction [Fiat-Shamir, 1987]
  - Take interactive identification protocol
  - Replace one side by a hash function  
 $\text{Challenge} = H(\text{smthng}, \text{context})$
  - Get non-interactive signature scheme



$\text{smthng, response}$  →

# Part I: Random functions vs. hash functions

# Random functions

- What we really want is  $H$  that behaves “just like a random function”:

Digest  $d=H(M)$  chosen uniformly for each  $M$

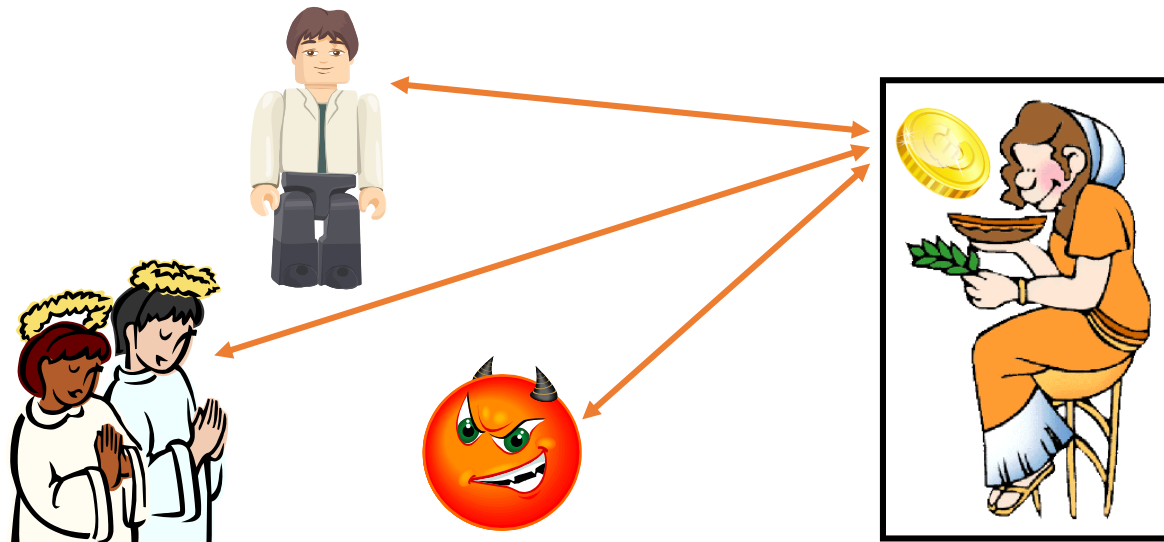
- Digest  $d=H(M)$  has no correlation with  $M$
- For distinct  $M_1, M_2, \dots$ , digests  $d_i=H(M_i)$  are completely uncorrelated to each other
- Cannot find collisions, or even near-collisions
- Cannot find  $M$  to “hit” a specific  $d$
- Cannot find fixed-points ( $d = H(d)$ )
- etc.



# The “Random-Oracle paradigm”

[Bellare-Rogaway, 1993]

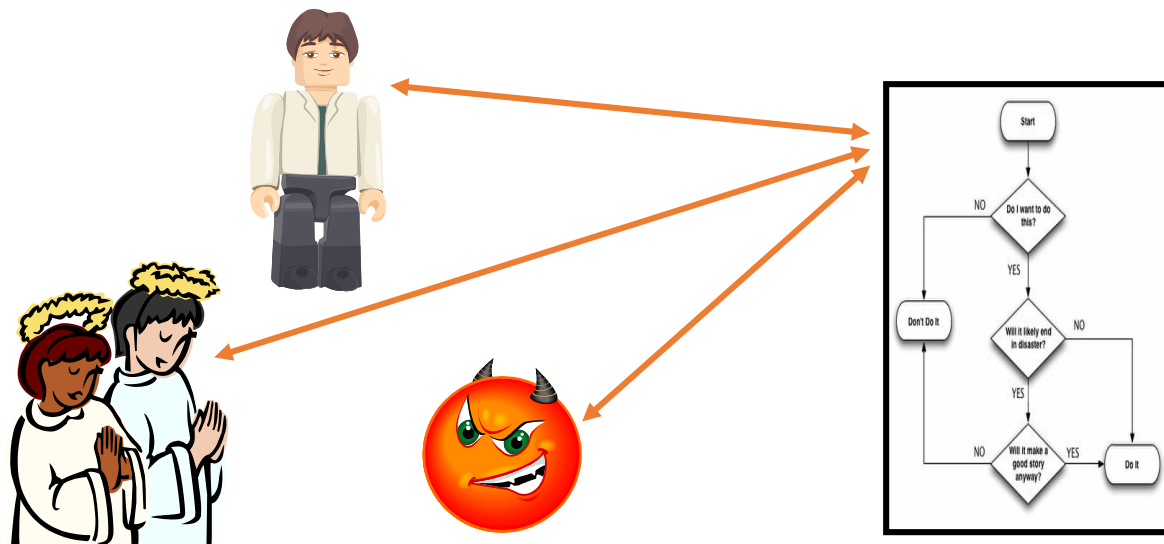
1. Pretend hash function is really this good
2. Design a secure cryptosystem using it
  - Prove security relative to a “random oracle”



# The “Random-Oracle paradigm”

[Bellare-Rogaway, 1993]

1. Pretend hash function is really this good
2. Design a secure cryptosystem using it
  - Prove security relative to a “random oracle”
3. Replace oracle with a hash function
  - Hope that it remains secure



# The “Random-Oracle paradigm”

[Bellare-Rogaway, 1993]

1. Pretend hash function is really this good
2. Design a secure cryptosystem using it
  - Prove security relative to a “random oracle”
3. Replace oracle with a hash function
  - Hope that it remains secure
- Very successful paradigm, many schemes
  - E.g., OAEP encryption, FDH, PSS signatures
    - Also all the examples from before...
  - Schemes seem to “withstand test of time”

# Random oracles: rationale

- $\mathcal{S}$  is some crypto scheme (e.g., signatures), that uses a hash function  $H$
- $\mathcal{S}$  proven secure when  $H$  is random function
- Any attack on real-world  $\mathcal{S}$  must use some “nonrandom property” of  $H$
- We should have chosen a better  $H$ 
  - without that “nonrandom property”
- Caveat: how do we know what “nonrandom properties” are important?

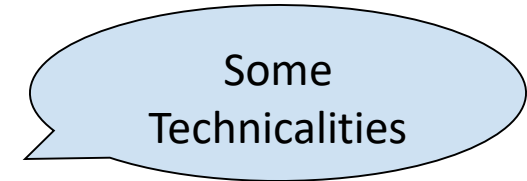
# This rationale isn't sound

[Canetti-Goldreich-H 1997]

- Exist signature schemes that are:
  1. Provably secure wrt a random function
  2. Easily broken for EVERY hash function
- Idea: hash functions are computable
  - This is a “nonrandom property” by itself
- Exhibit a scheme which is secure only for “non-computable H's”
  - Scheme is (very) “contrived”

# Contrived example

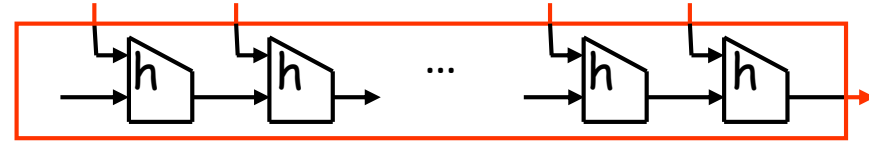
- Start from any secure signature scheme
  - Denote signature algorithm by  $SIG1^H(key, msg)$
- Change  $SIG1$  to  $SIG2$  as follows:  
 $SIG2^H(key, msg)$ : interpret  $msg$  as code  $\Pi$ 
  - If  $\Pi(i)=H(i)$  for  $i=1,2,3,\dots, |msg|$ , then output  $key$
  - Else output the same as  $SIG1^H(key, msg)$
- If  $H$  is random, always the “Else” case
- If  $H$  is a hash function, attempting to sign the code of  $H$  outputs the secret key




# Cautionary note

- ROM proofs may not mean what you think...
  - Still they give valuable assurance, rule out “almost all realistic attacks”
- What “nonrandom properties” are important for OAEP / FDH / PSS / ...?
- How would these scheme be affected by a weakness in the hash function in use?
- ROM may lead to careless implementation

# Merkle-Damgård vs. random functions



- Recall: we often construct our hash functions from compression functions
  - Even if compression is random, hash is not
    - E.g.,  $H(\text{key} | M)$  subject to extension attack
      - $H(\text{key} | M | M') = h(H(\text{key} | M), M')$
    - Minor changes to MD fix this
      - But they come with a price (e.g. prefix-free encoding)
- Compression also built from low-level blocks
  - E.g., Davies-Meyer construction,  $h(c, M) = E_M(c)$    $c$
  - Provide yet more structure, can lead to attacks on provable ROM schemes [H-Krawczyk 2007]

# Part II: Using hash functions in applications

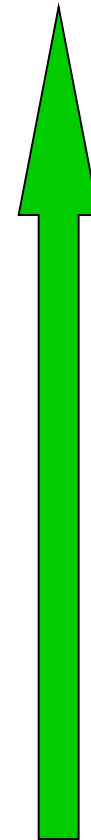
# Using “imperfect” hash functions

- Applications should rely only on “specific security properties” of hash functions
  - Try to make these properties as “standard” and as weak as possible
- Increases the odds of long-term security
  - When weaknesses are found in hash function, application more likely to survive
  - E.g., MD5 is badly broken, but HMAC-MD5 is barely scratched

# Security requirements

- Deterministic hashing
  - Attacker chooses  $M$ ,  $d=H(M)$
- Hashing with a random salt
  - Attacker chooses  $M$ , then good guy chooses public salt,  $d=H(\text{salt},M)$
- Hashing random messages
  - $M$  random,  $d=H(M)$
- Hashing with a secret key
  - Attacker chooses  $M$ ,  $d=H(\text{key},M)$

Stronger



Weaker

# Deterministic hashing

- Collision Resistance
  - Attacker cannot find  $M, M'$  such that  $H(M)=H(M')$
- Also many other properties
  - Hard to find fixed-points, near-collisions,  $M$  s.t.  $H(M)$  has low Hamming weight, etc.

# Hashing with public salt

- Target-Collision-Resistance (TCR)
  - Attacker chooses  $M$ , then given random  $salt$ , cannot find  $M'$  such that  $H(salt, M) = H(salt, M')$
- enhanced TRC (eTCR)
  - Attacker chooses  $M$ , then given random  $salt$ , cannot find  $M', salt'$  s.t.  $H(salt, M) = H(salt', M')$

# Hashing random messages

- Second Preimage Resistance
  - Given random  $M$ , attacker cannot find  $M'$  such that  $H(M)=H(M')$
- One-wayness
  - Given  $d=H(M)$  for random  $M$ , attacker cannot find  $M'$  such that  $H(M')=d$
- Extraction\*
  - For random  $salt$ , high-entropy  $M$ , the digest  $d=H(salt,M)$  is close to being uniform

\* Combinatorial, not cryptographic

# Hashing with a secret key

- Pseudo-Random Functions
  - The mapping  $M \rightsquigarrow H(\text{key}, M)$  for secret *key* looks random to an attacker
- Universal hashing\*
  - For all  $M \neq M'$ ,  $\Pr_{\text{key}}[H(\text{key}, M) = H(\text{key}, M')] < \epsilon$

\* Combinatorial, not cryptographic

# Application 1:

## Digital signatures


- Hash-then-sign paradigm
    - First shorten the message,  $d = H(M)$
    - Then sign the digest,  $s = \text{SIGN}(d)$
  - Relies on collision resistance
    - If  $H(M) = H(M')$  then  $s$  is a signature on both
- Attacks on MD5, SHA-1 threaten current signatures
- MD5 attacks can be used to get bad CA cert
- [Stevens et al. 2009]

# Collision resistance is hard

- Attacker works off-line (find  $M, M'$ )
  - Can use state-of-the-art cryptanalysis, as much computation power as it can gather, without being detected !!
- Helped by birthday attack (e.g.,  $2^{80}$  vs  $2^{160}$ )
- Well worth the effort
  - One collision → forgery for any signer

# Signatures without CRHF

[Naor-Yung 1989, Bellare-Rogaway 1997]

- Use randomized hashing
  - To sign  $M$ , first choose fresh random  $salt$
  - Set  $d = H(salt, M)$ ,  $s = SIGN(salt || d)$
- Attack scenario (collision game):
  - Attacker chooses  $M, M'$
  - Signer chooses random salt 
  - Attacker must find  $M'$  s.t.  $H(salt, M) = H(salt, M')$
- Attack is inherently on-line
  - Only rely on target collision resistance



same salt (since salt is explicitly signed)

# TCR hashing for signatures

- Not every randomization works
  - $H(M | salt)$  may be subject to collision attacks
    - when  $H$  is Merkle-Damgård
    - Yet this is what PSS does (and it's provable in the ROM)
- Many constructions “in principle”
  - From any one-way function
- Some engineering challenges
  - Most constructions use long/variable-size randomness, don't preserve Merkle-Damgård
- Also, signing salt means changing the underlying signature schemes

# Signatures with enhanced TCR

[H-Krawczyk 2006]

- Use “stronger randomized hashing”, eTCR
  - To sign  $M$ , first choose fresh random  $salt$
  - Set  $d = H(salt, M)$ ,  $s = SIGN(d)$
- Attack scenario (collision game):
  - Attacker chooses  $M$
  - Signer chooses random  $salt$
  - Attacker needs  $M', salt'$  s.t.  $H(salt, M) = H(salt', M')$
- Attack is still inherently on-line

attacker can use  
different salt'

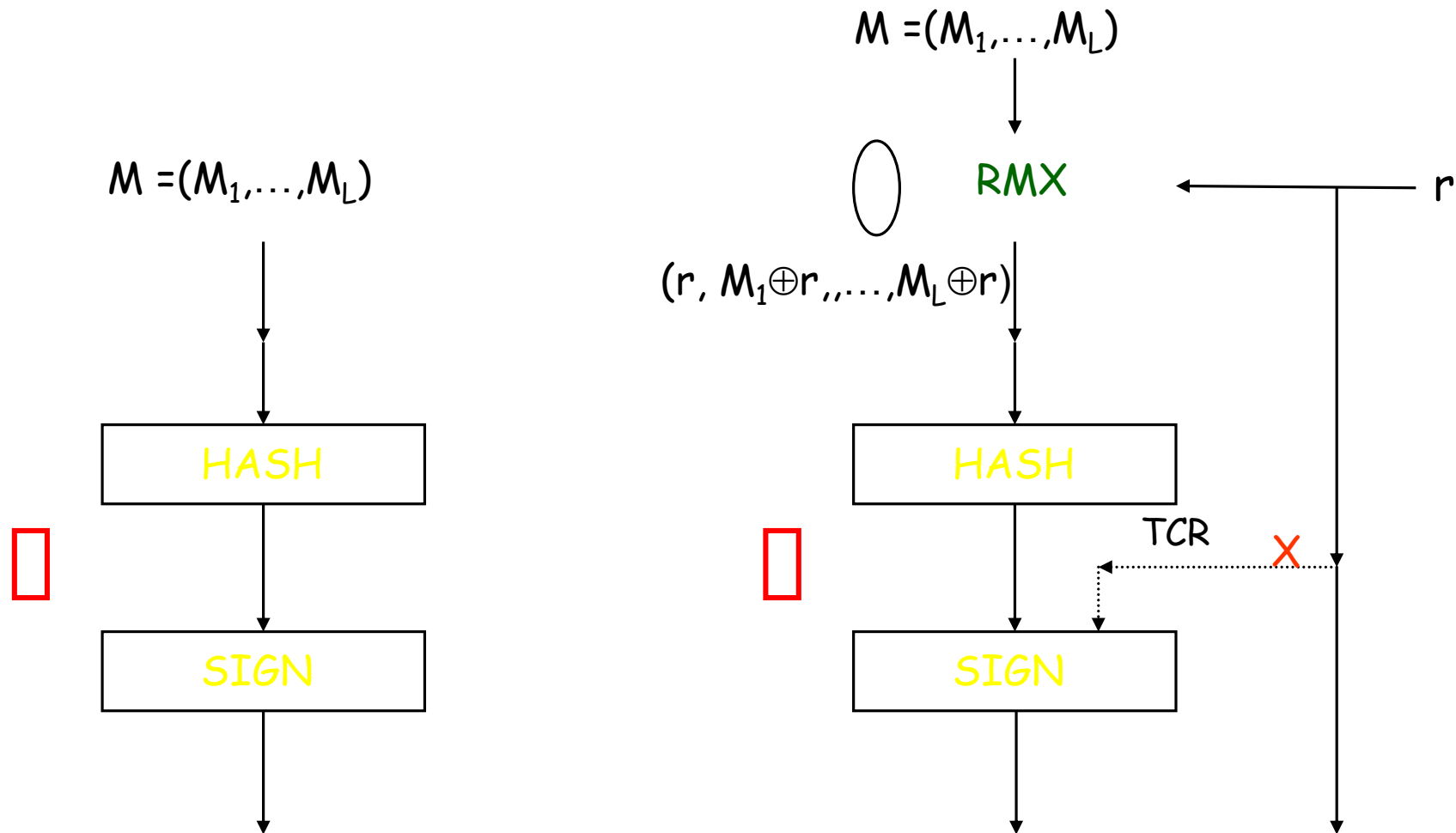


# Randomized hashing with RMX

[H-Krawczyk 2006]

- Use simple message-randomization
  - RMX:  $M=(M_1, M_2, \dots, M_L), r \stackrel{\approx}{\sim}$   
 $(r, M_1 \oplus r, M_2 \oplus r, \dots, M_L \oplus r)$
- Hash( RMX( $r, M$ ) ) is eTCR when:
  - Hash is Merkle-Damgård, and
  - Compression function is  $\sim 2^{\text{nd}}$ -preimage-resistant
- Signature: [  $r, \text{SIGN}( \text{Hash}( \text{RMX}(r, M) ) )$  ]
  - $r$  fresh per signature, one block (e.g. 512 bits)
  - No change in Hash, no signing of  $r$

# Preserving hash-then-sign



# Application 2: Message authentication

- Sender, Receiver, share a secret key
- Compute an *authentication tag*
  - $tag = \text{MAC}(key, M)$
- Sender sends ( $M, tag$ )
- Receiver verifies that  $tag$  matches  $M$
- Attacker cannot forge tags without key

# Authentication with HMAC

[Bellare-Canetti-Krawczyk 1996]

- Simple key-prepend/append have problems when used with a Merkle-Damgård hash
  - $\text{tag} = H(\text{key} \mid M)$  subject to extension attacks
  - $\text{tag} = H(M \mid \text{key})$  relies on collision resistance
- HMAC: Compute  $\text{tag} = H(\text{key} \mid H(\text{key} \mid M))$ 
  - About as fast as key-prepend for a MD hash
- Relies only on PRF quality of hash
  - $M \stackrel{?}{\approx} H(\text{key} \mid M)$  looks random when key is secret

# Authentication with HMAC

[Bellare-Canetti-Krawczyk 1996]

- Simple key-prepend/append have problems when used with a Merkle-Damgård hash
  - $\text{tag} = H(\text{key} \parallel M)$
  - $\text{tag} = H(M \parallel \text{key})$
- HMAC: Composed of two HMACs
  - About as fast as MD5/SHA1
- Relies only on PRF property of hash
  - $M \stackrel{?}{\approx} H(\text{key} \parallel M)$  looks random when key is secret

As a result, barely affected by collision attacks on MD5/SHA1

# Carter-Wegman authentication

[Wegman-Carter 1981,...]

- Compress message with hash,  $t = H(key_1, M)$
- Hide  $t$  using a PRF,  $tag = t \oplus PRF(key_2, nonce)$ 
  - PRF can be AES, HMAC, RC4, etc.
  - Only applied to a short nonce, typically not a performance bottleneck
- Secure if the PRF is good,  $H$  is “universal”
  - For  $M \neq M', \Delta, Pr_{key} [ H(key, M) \oplus H(key, M') = \Delta ] < \epsilon$
  - Not cryptographic, can be very fast

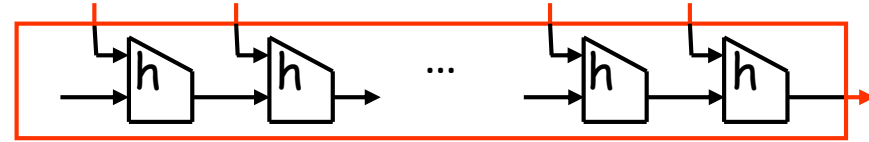
# Fast Universal Hashing

- “Universality” is combinatorial, provable
  - no need for “security margins” in design
- Many works on fast implementations
  - From inner-product,  $H_{k_1, k_2}(M_1, M_2) = (K_1 + M_1) \cdot (K_2 + M_2)$ 
    - [H-Krawczyk’97, Black et al.’99, ...]
  - From polynomial evaluation  $H_k(M_1, \dots, M_L) = \sum_i M_i k^i$ 
    - [Krawczyk’94, Shoup’96, Bernstein’05, McGrew-Viega’06, ...]
- As fast as 2-3 cycle-per-byte (for long M’s)
  - Software implementation, contemporary CPUs

# Part III: Designing a hash function

Fugue: IBM's candidate for the NIST hash competition

# Design a compression function?



PROs: modular design, reduce to the “simpler problem” of compressing fixed-length strings

- Many things are known about transforming compression into hash

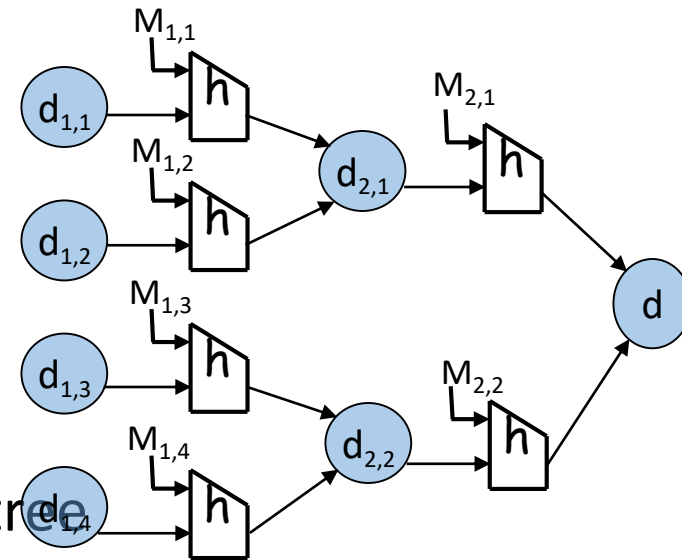
CONs: compression  $\rightarrow$  hash has its problems

- It's not free (e.g. message encoding)
- Some attacks based on the MD structure
  - Extension attacks ( rely on  $H(x|y)=h(H(x),y)$  )
  - “Birthday attacks” (herding, multicollisions, ...)

# Example attack: herding

[Kelsey-Kohno 2006]

- Find many off-line collisions
    - “Tree structure” with  $\sim 2^{n/3}$   $d_{i,j}$ 's
    - Takes  $\sim 2^{2n/3}$  time
  - Publish final  $d$
  - Then for any prefix  $P$ 
    - Find “linking block”  $L$  s.t.  $H(P|L)$  in the tree
    - Takes  $\sim 2^{2n/3}$  time
    - Read off the tree the suffix  $S$  to get to  $d$
- Show an extension of  $P$  s.t.  $H(P|L|S) = d$

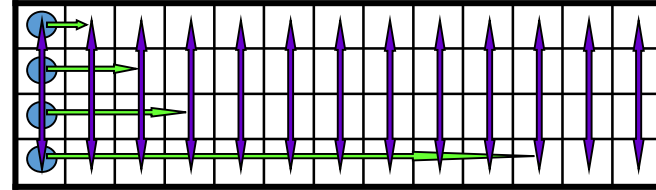


# The culprit: small intermediate state

- With a compression function, we:
  - Work hard on current message block
  - Throw away this work, keep only n-bit state
- Alternative: keep a large state
  - Work hard on current message block/word
  - Update some part of the big state
- More flexible approach
  - Also more opportunities to mess things up

# The hash function Grindahl

[Knudsen-Rechberger-Thomsen 2007]



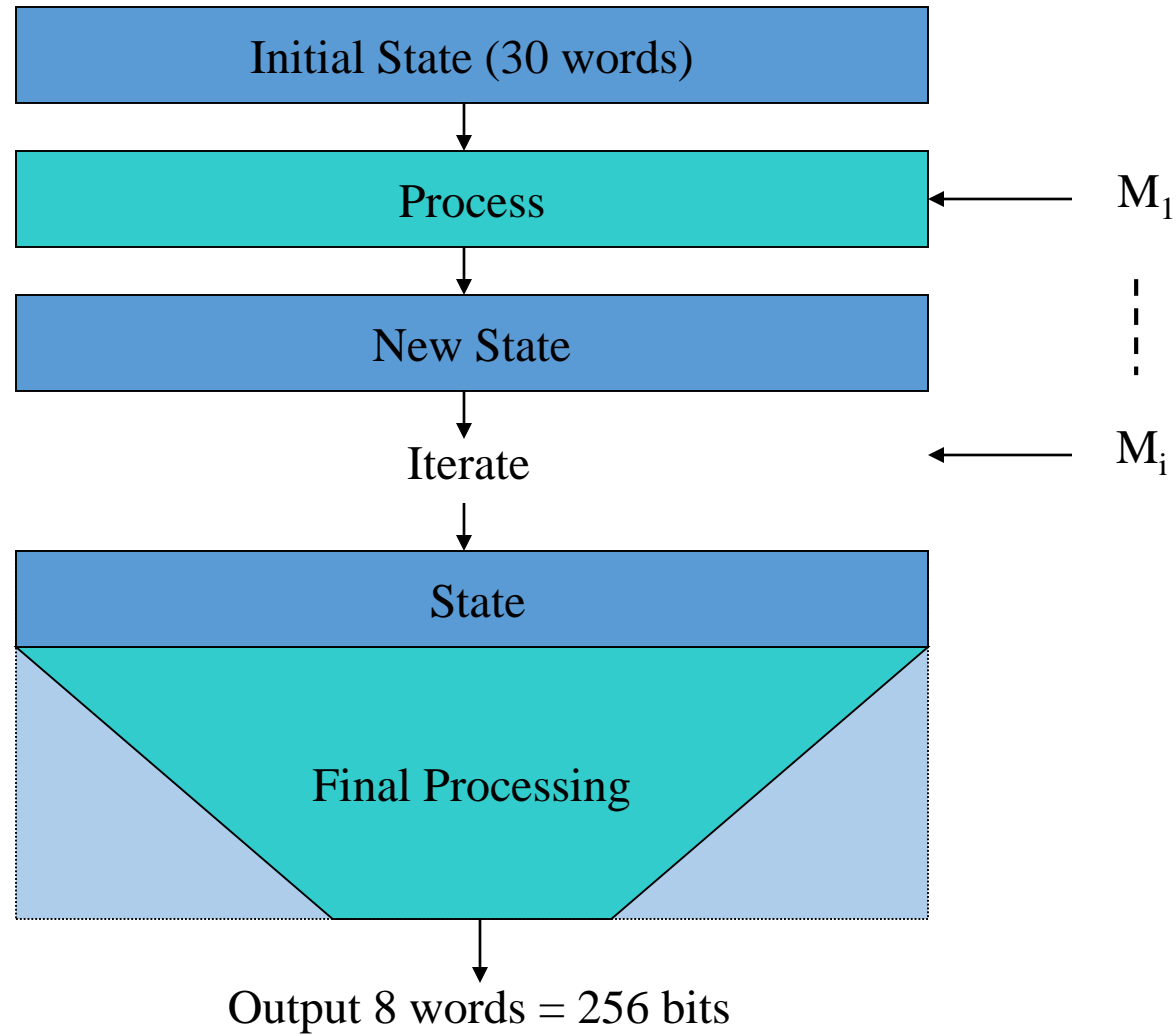
- State is 13 words = 52 bytes
- Process one 4-byte word at a time
  - One AES-like mixing step per word of input
- After some final processing, output 8 words
- Collision attack by Peyrin (2007)
  - Complexity  $\sim 2^{112}$  (still better than brute-force)
    - Recently improved to  $\sim 2^{100}$  [Khovratovich 2009]
  - “Start from a collision and go backwards”

# The hash function “Fugue”

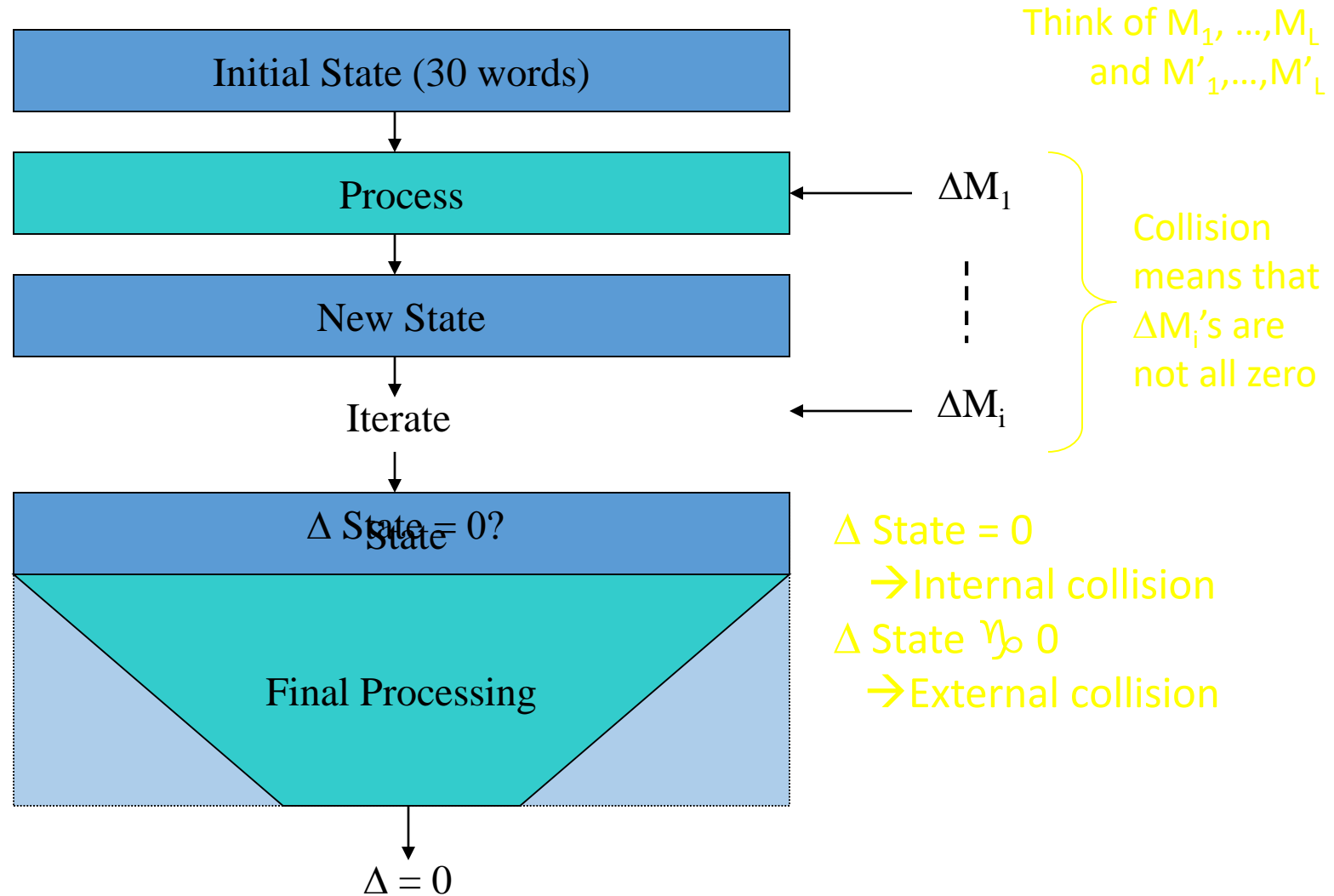
[H-Hall-Jutla 2008]

- Proof-driven design
  - Designed to enable analysis
    - Proofs that Peyrin-style attacks do not work
- State of 30 4-byte words = 120 bytes
- Two “super-mixing” rounds per word of input
  - Each applied to only 16 bytes of the state
  - With some extra linear diffusion
- Super-mixing is AES-like
  - But uses stronger MDS codes

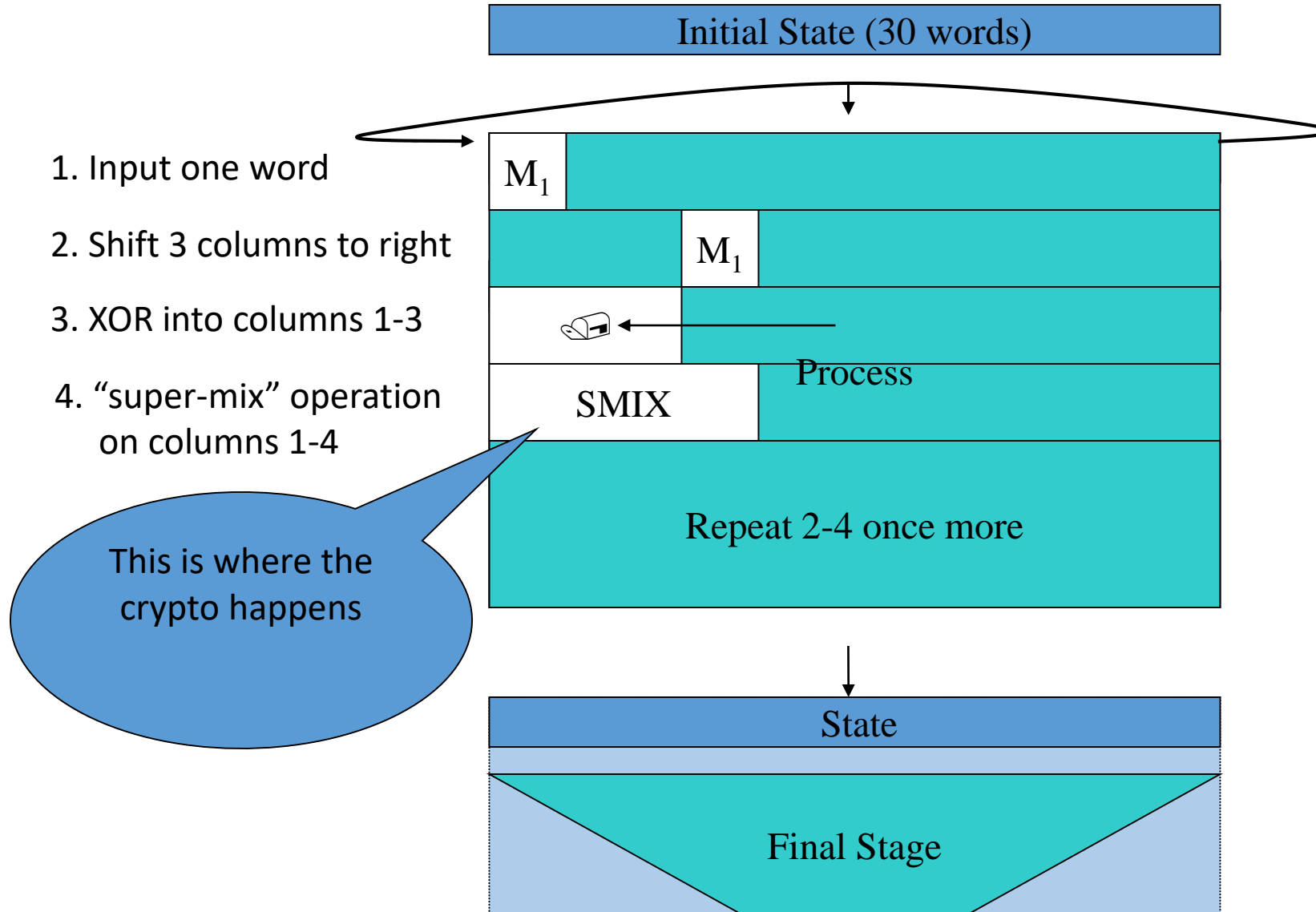
# Fugue-256



# Collision attacks

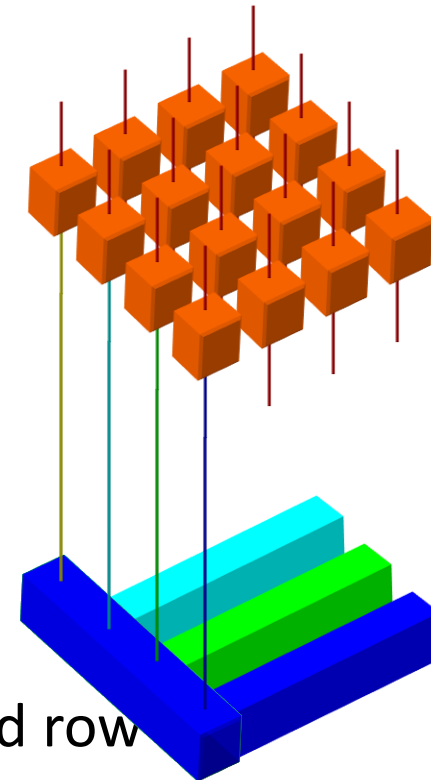


# Processing one input word



# SMIX in Fugue

- Similar to one AES round
  - Works on a 4x4 matrix of bytes
  - Starts with S-box substitution
    - Byte  $b$ ,  $S[256] = \{\dots\};$
    - $\ddots$
    - $\hat{b} = S[b];$
  - Does linear mixing
- Stronger mixing than AES
  - Diagonal bytes as in AES
  - Other bytes are mixed into both column and row

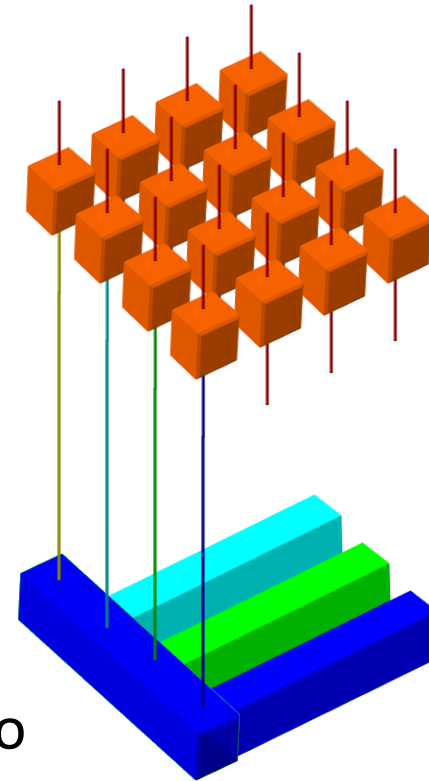


# SMIX in Fugue

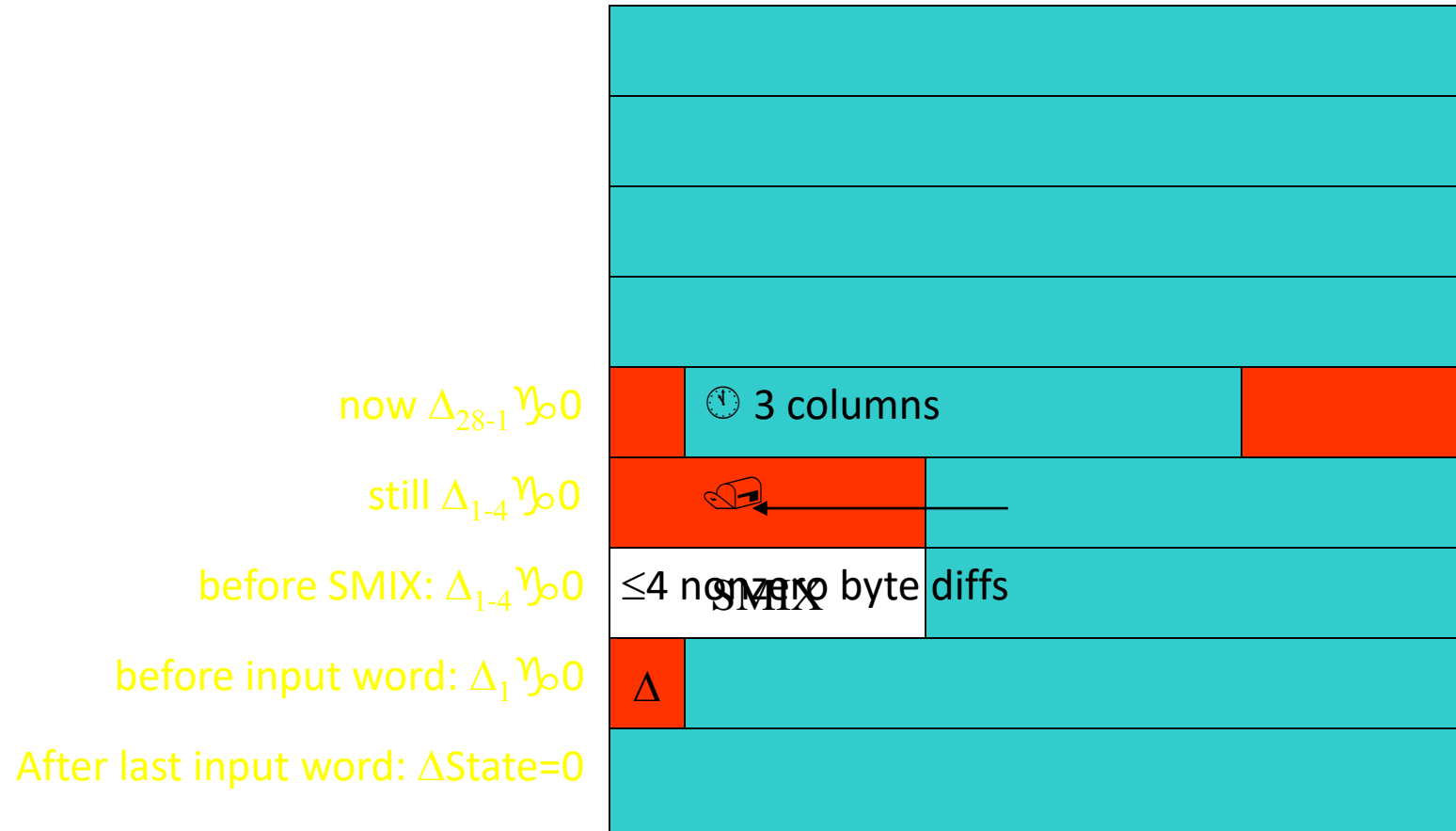
- In algebraic notation:

$$\begin{pmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_{16} \end{pmatrix} = \mathbf{M}_{16 \times 16} \times \begin{pmatrix} S[b_1] \\ S[b_2] \\ \vdots \\ S[b_{16}] \end{pmatrix}$$

- M generates a good linear code
  - If all the  $b'_i$  bytes but 4 are zero then  $\geq 13$  of the  $S[b_i]$  bytes must be nonzero
  - And other such properties



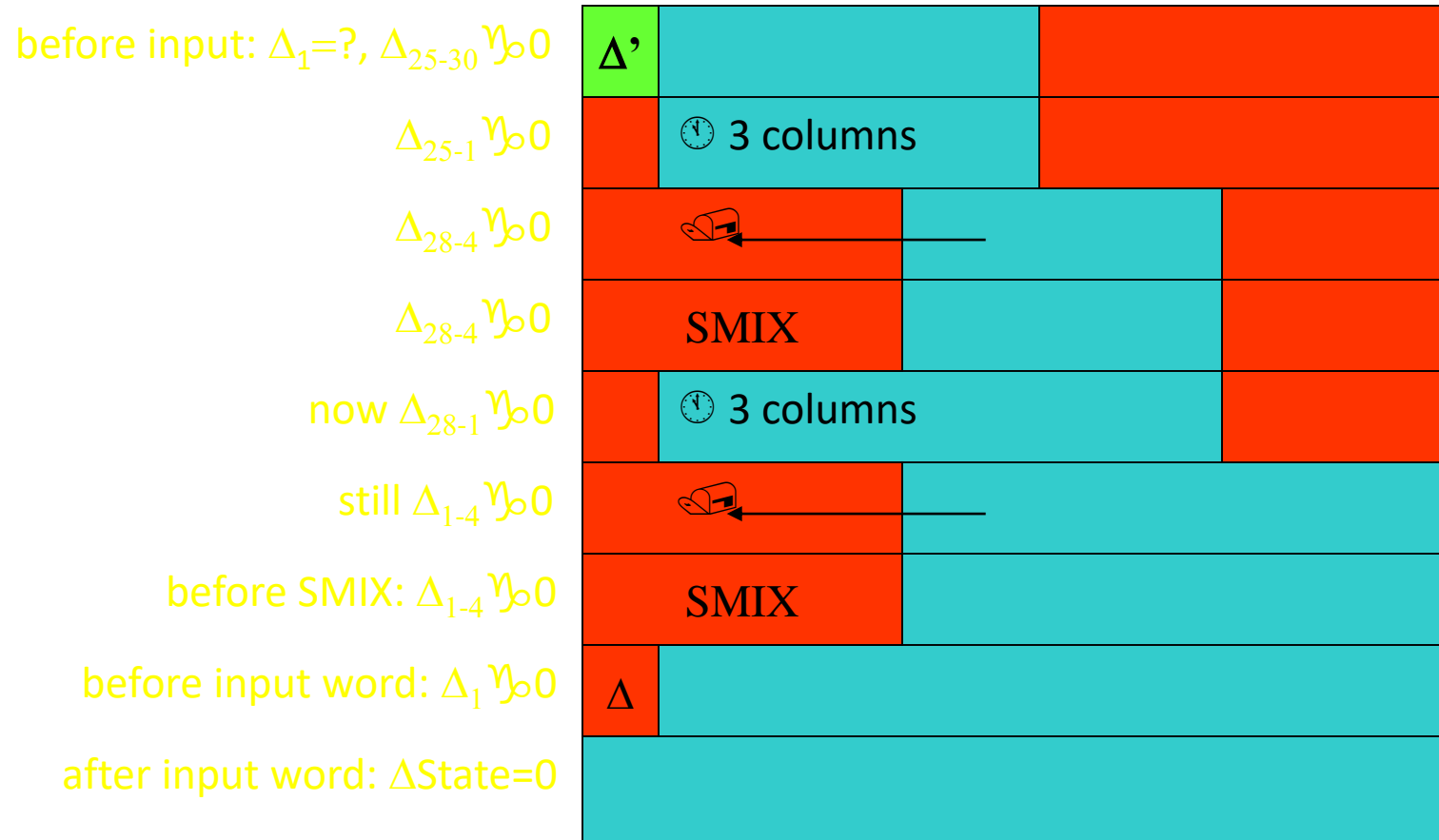
# Analyzing internal collisions\*



\* a bit oversimplified



# Analyzing internal collisions\*



\* a bit oversimplified

Table 8: Evolution of P...

	0	3																						
TIX <sub>0</sub>	0																							
$\Delta S_{[-1]}$	X1	0	0	0																				
SMIX	Y <sub>10</sub>	Y <sub>11</sub>	Y <sub>12</sub>	Y <sub>13</sub>																				
CMIX	Y <sub>10</sub>	Y <sub>11</sub>	Y <sub>12</sub>	Y <sub>13</sub>																				
ROR3	Y <sub>13</sub>	0	0	0	X1					Y <sub>10</sub>	Y <sub>11</sub>	Y <sub>12</sub>												
SMIX	Z <sub>10</sub>	Z <sub>11</sub>	Z <sub>12</sub>	Z <sub>13</sub>	X1					Y <sub>10</sub>	Y <sub>11</sub>	Y <sub>12</sub>												
CMIX	Z <sub>10</sub>	Z <sub>11</sub>	Z <sub>12</sub>	Z <sub>13</sub>	X1					Y <sub>10</sub>	Y <sub>11</sub>	Y <sub>12</sub>												
ROR3	Z <sub>13</sub>	0	0	0	X1					Y <sub>10</sub>	Y <sub>11</sub>	Y <sub>12</sub>	Z <sub>10</sub>	Z <sub>11</sub>	Z <sub>12</sub>									
TIX <sub>-1</sub>																								
$\Delta S_{[-2]}$	x <sub>2</sub>	Y <sub>10</sub>	0	0	X1	Z <sub>13</sub>	0	x <sub>2</sub>					Y <sub>10</sub>	Y <sub>11</sub>	Y <sub>12</sub>	Z <sub>10</sub>	Z <sub>11</sub>	Z <sub>12</sub>						
SMIX	y <sub>20</sub>	y <sub>21</sub>	y <sub>22</sub>	y <sub>23</sub>	X1	Z <sub>13</sub>	0	x <sub>2</sub>					Y <sub>10</sub>	Y <sub>11</sub>	Y <sub>12</sub>	Z <sub>10</sub>	Z <sub>11</sub>	Z <sub>12</sub>						
CMIX	y <sub>20</sub>	y <sub>21</sub>	y <sub>22</sub>	y <sub>23</sub>	X1	Z <sub>13</sub>	0	x <sub>2</sub>	X1				Y <sub>10</sub>	Y <sub>11</sub>	Y <sub>12</sub>	Z <sub>10</sub>	Z <sub>11</sub>	Z <sub>12</sub>						
ROR3	y <sub>23</sub>	X1	0	0	0	Z <sub>13</sub>	0	x <sub>2</sub>	0	X1			Y <sub>10</sub>	Y <sub>11</sub>	Y <sub>12</sub>	Z <sub>10</sub>	Z <sub>11</sub>	Z <sub>12</sub>	y <sub>20</sub>	y <sub>21</sub>	y <sub>22</sub>			
SMIX	z <sub>20</sub>	z <sub>21</sub>	z <sub>22</sub>	z <sub>23</sub>	0	Z <sub>13</sub>	0	x <sub>2</sub>	0	X1			Y <sub>10</sub>	Y <sub>11</sub>	Y <sub>12</sub>	Z <sub>10</sub>	Z <sub>11</sub>	Z <sub>12</sub>	y <sub>20</sub>	y <sub>21</sub>	y <sub>22</sub>			
CMIX	z <sub>20</sub>	z <sub>21</sub>	z <sub>22</sub>	z <sub>23</sub>	0	Z <sub>13</sub>	0	x <sub>2</sub>	0	X1	Z <sub>13</sub>		Y <sub>10</sub>	Y <sub>11</sub>	Y <sub>12</sub>	Z <sub>10</sub>	Z <sub>11</sub>	Z <sub>12</sub>	y <sub>20</sub>	y <sub>21</sub>	y <sub>22</sub>			
ROR3	z <sub>23</sub>	0	Z <sub>13</sub>	0	x <sub>2</sub>		X1		0	Z <sub>13</sub>			Y <sub>10</sub>	Y <sub>11</sub>	Y <sub>12</sub>	Z <sub>10</sub>	Z <sub>11</sub>	Z <sub>12</sub>	y <sub>20</sub>	y <sub>21</sub>	y <sub>22</sub>	z <sub>20</sub>	z <sub>21</sub>	z <sub>22</sub>
TIX <sup>††</sup>																								

Many nonzero byte differences before the SMIX operations

The analysis from previous slides was upto here




<sup>a</sup> All blank cells are zero. Primed variables are defined in Section 10.1.4. The shaded cells are the ones affected in that step. The boxed variables are the ones that are not determined by variables from earlier (lower) steps. Variables that are necessarily non-zero are in capital. Rounds are referred to by the subscript on the TIX step for that round. <sup>††</sup> Continued on next page.

Table 9: Evolution of Differential State for internal Collision (contd.)

	0	3	6	9	12	15	18	21	24	27	29		
TIX <sub>-2</sub>	$z_{23}$	0	$Z_{13}$	0	$x_2$		$X_1$		$Z_{13}$		$Y_{10}Y_{11}Y_{12}Z_{10}Z_{11}Z_{12}y_2''y_2'y_2z_2z_2'z_2'$		
$\Delta S_{[-3]}$	$x_3$	$y_2''$	$Z_{13}$	0	$x_2$		$z_2'y_2'X_1$	$x_3$	$Z_{13}$		$Y_{10}Y_{11}Y_{12}Z_{10}Z_{11}Z_{12}y_2''y_2'y_2z_2z_2'z_2'$		
SMIX	$y_3''$	$y_3'$	$y_3''$	$y_3''$	$x_2$		$z_2'y_2'X_1$	$x_3$	$Z_{13}$		$Y_{10}Y_{11}Y_{12}Z_{10}Z_{11}Z_{12}y_2''y_2'y_2z_2z_2'z_2'$		
CMIX	$y_3''$	$y_3'$	$y_3''$	$y_3''$	$x_2$		$z_2'y_2'X_1$	$x_3$	$Z_{13}$	0	$x_2$	$Y_{10}Y_{11}Y_{12}Z_{10}Z_{11}Z_{12}y_2''y_2'y_2z_2z_2'z_2'$	
ROR3	$y_3''$	$x_2$	0	0	$z_2'y_2'X_1$	$x_3$	$Z_{13}$	0	$x_2$	$Y_{10}Y_{11}Y_{12}Z_{10}Z_{11}Z_{12}y_2''y_2'y_2z_2z_2'z_2'$	$z_2'y_2'y_2z_2z_2'z_2'$		
SMIX	$z_3''$	$z_3'$	$z_3''$	$z_3''$	$z_2'y_2'X_1$	$x_3$	$Z_{13}$	0	$x_2$	$Y_{10}Y_{11}Y_{12}Z_{10}Z_{11}Z_{12}y_2''y_2'y_2z_2z_2'z_2'$	$z_2'y_2'y_2z_2z_2'z_2'$		
CMIX	$z_3''$	$z_3'$	$z_3''$	$z_3''$	$z_2'y_2'X_1$	$x_3$	$Z_{13}$	0	$x_2$	$Y_{10}Y_{11}Y_{12}Z_{10}Z_{11}Z_{12}y_2''y_2'y_2z_2z_2'z_2'$	$z_2'y_2'y_2z_2z_2'z_2'$		
ROR3	$z_3''$	$z_3'$	$z_3''$	$z_3''$	$z_2'y_2'X_1$	$x_3$	$Z_{13}$	0	$x_2$	$Y_{10}y_1'y_1'z_1'Z_{10}Z_{11}Z_{12}y_2''y_2'y_2z_2z_2'z_2'$	$z_2'y_2'y_2z_2z_2'z_2'$		
TIX <sub>-3</sub>	$z_3''$	0	$z_2'y_2'X_1$	$x_3$	0	0	$Z_{13}$	0	$x_2$	0	$Y_{10}y_1'y_1'z_1'Z_{10}Z_{11}Z_{12}y_2''y_2'y_2z_2z_2'z_2'$		
$\Delta S_{[-4]}$	$x_4$	$y_3''$	$z_2'y_2'X_1$	$x_3$	0	0	$Z_{13}$	$z_3''$	$x_2$	$x_4$	0	$Y_{10}y_1'y_1'z_1'Z_{10}Z_{11}Z_{12}y_2''y_2'y_2z_2z_2'z_2'$	
SMIX	$y_4''$	$y_4'$	$y_4''$	$y_4''$	$x_3$	0	0	$Z_{13}$	$z_3''$	$x_2$	$x_4$	0	$Y_{10}y_1'y_1'z_1'Z_{10}Z_{11}Z_{12}y_2''y_2'y_2z_2z_2'z_2'$
CMIX	$y_4''$	$y_4'$	$y_4''$	$y_4''$	$x_3$	0	0	$Z_{13}$	$z_3''$	$x_2$	$x_4$	0	$Y_{10}y_1'y_1'z_1'Z_{10}Z_{11}Z_{12}y_2''y_2'y_2z_2z_2'z_2'$
ROR3	$y_4''$	$x_3$	0	0	$Z_{13}$	$z_3''$	$x_2$	$x_4$	0	$Y_{10}y_1'y_1'z_1'Z_{10}Z_{11}Z_{12}y_2''y_2'y_2z_2z_2'z_2'$	$z_2'y_2'y_2z_2z_2'z_2'$		
SMIX	$z_4''$	$z_4'$	$z_4''$	$z_4''$	$Z_{13}$	$z_3''$	$x_2$	$x_4$	0	$Y_{10}y_1'y_1'z_1'Z_{10}Z_{11}Z_{12}y_2''y_2'y_2z_2z_2'z_2'$	$z_2'y_2'y_2z_2z_2'z_2'$		
CMIX	$z_4''$	$z_4'$	$z_4''$	$z_4''$	$Z_{13}$	$z_3''$	$x_2$	$x_4$	0	$Y_{10}y_1'y_1'z_1'Z_{10}Z_{11}Z_{12}y_2''y_2'y_2z_2z_2'z_2'$	$z_2'y_2'y_2z_2z_2'z_2'$		
ROR3	$z_4''$	$Z_{13}$	$z_3''$	$x_2$	$x_4$	0	$Y_{10}y_1'y_1'z_1'Z_{10}Z_{11}Z_{12}y_2''y_2'y_2z_2z_2'z_2'$	$z_2'y_2'y_2z_2z_2'z_2'$	$z_2'y_2'y_2z_2z_2'z_2'$	$z_2'y_2'y_2z_2z_2'z_2'$	$z_2'y_2'y_2z_2z_2'z_2'$		
TIX <sub>-4</sub>	$z_4''$	$Z_{13}$	$z_3''$	$x_2$	$x_4$	0	$Y_{10}y_1'y_1'z_1'Z_{10}Z_{11}Z_{12}y_2''y_2'y_2z_2z_2'z_2'$	$z_2'y_2'y_2z_2z_2'z_2'$	$z_2'y_2'y_2z_2z_2'z_2'$	$z_2'y_2'y_2z_2z_2'z_2'$	$z_2'y_2'y_2z_2z_2'z_2'$		
$\Delta S_{[-5]}$	$x_5$	$Z_{13}$	$z_3''$	$x_2$	$x_4$	0	$Y_{10}y_1'y_1'z_1'Z_{10}Z_{11}Z_{12}y_2''y_2'y_2z_2z_2'z_2'$	$z_2'y_2'y_2z_2z_2'z_2'$	$z_2'y_2'y_2z_2z_2'z_2'$	$z_2'y_2'y_2z_2z_2'z_2'$	$z_2'y_2'y_2z_2z_2'z_2'$		
		+					+	+					
		$y_4''$					$z_4''$	$x_5$					

<sup>a</sup> Primed variables are defined in Sections 10.1.5 and 10.1.6.

# Analyzing internal collisions

- What does this mean? Consider this attack:
  - Attacker feeds in random  $M_1, M_2, \dots$  and  $M'_1, M'_2, \dots$
  - Until  $\text{State}_L$    $\text{State}'_L = \text{some "good } \Delta \text{"}$
  - Then it searches for suffixed  $(M_{L+1}, \dots, M_{L+4}), (M'_{L+1}, \dots, M'_{L+4})$  that will induce internal collision

Theorem\*: For any fixed  $\Delta$ ,

$$\Pr[ \exists \text{ suffixes that induce collision } ] < 2^{-150}$$

\* Relies on a very mild independence assumptions

# Analyzing internal collisions

- Why do we care about this analysis?
- Peyrin's attacks are of this type
- All differential attacks can be seen as (optimizations of) this attack
  - Entities that are not controlled by attack are always presumed random
- A known "collision trace" is as close as we can get to understanding collision resistance

# Fugue: concluding remarks

- Similar analysis also for external collisions
  - “Unusually thorough” level of analysis
- Performance comparable to SHA-256
  - But more amenable to parallelism
- One of 14 submissions that were selected by NIST to advance to 2<sup>nd</sup> round of the SHA3 competition

# Morals

- Hash functions are very useful
- We want them to behave “just like random functions”
  - But they don’t really
- Applications should be designed to rely on “as weak as practical” properties of hashing
  - E.g., TCR/eTCR rather than collision-resistance
- A taste of how a hash function is built

# Reference and source

---

- 1. Bitcoin & Cryptocurrency Technologies: Bitcoin Mining, Blockchain Basics And Cryptocurrency Trading & Investing For Beginners | 7 Books In 1 by Boris Weiser (Author)
- 2. The Basics of Bitcoins and Blockchains: An Introduction to Cryptocurrencies and the Technology that Powers Them (Cryptography, Crypto Trading, Derivatives, Digital Assets) – Illustrated, September 15, 2018 by Antony Lewis (Author)
- 3. Blockchain Revolution: How the Technology Behind Bitcoin and Other Cryptocurrencies Is Changing the World – June 12, 2018 by Don Tapscott (Author), Alex Tapscott (Author)
- 4. Blockchain Technology for IoT Applications (Blockchain Technologies) 1st ed. 2021 Edition
- by Seok-Won Lee (Editor), Irish Singh (Editor), Masoud Mohammadian (Editor)
- 5. Blockchain Basics: A Non-Technical Introduction in 25 Steps by Daniel Drescher (Author) Format
- 6. Cryptoassets: The Innovative Investor's Guide to Bitcoin and Beyond – October 19, 2017 by Chris Burniske (Author), Jack Tatar (Author)

Thank you!