Cryptographic Hash Functions
and their many applications

Shai Halevi – IBM Research

USENIX Security – August 2009

Thanks to Charanjit Jutla and Hugo Krawczyk

What are hash functions?

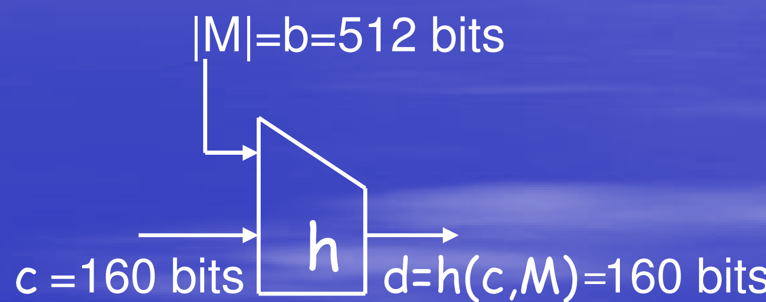
- Just a method of compressing strings
  - E.g., $H : \{0,1\}^* \rightarrow \{0,1\}^{160}$
  - Input is called "message", output is "digest"
- Why would you want to do this?
  - Short, fixed-size better than long, variable-size
    - True also for non-crypto hash functions
  - Digest can be added for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

But not always…

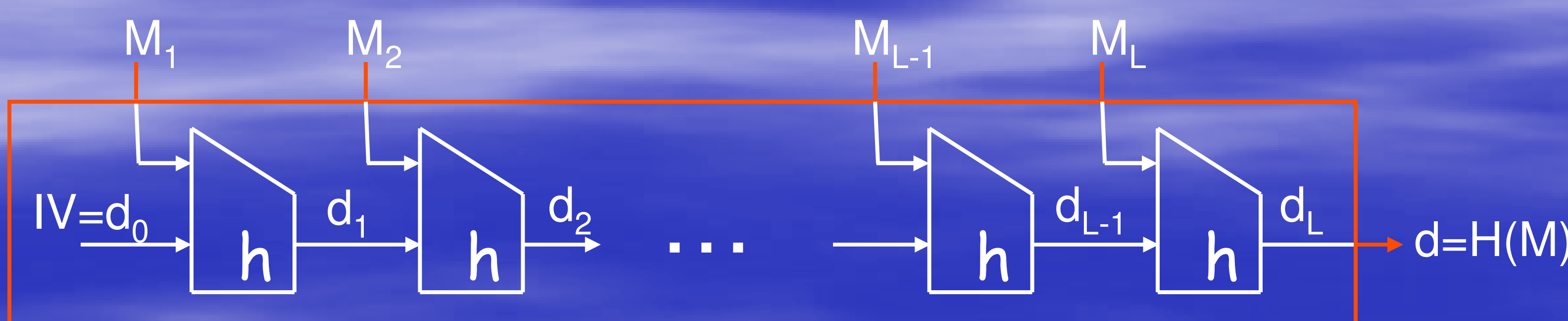
How are they built?

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?

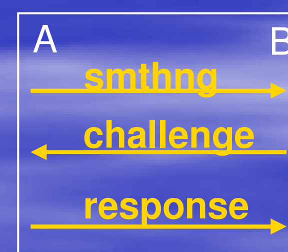
"Modern, collision resistant hash functions were designed to create small, fixed size message digests so that a digest could act as a proxy for a possibly very large variable length message in a **digital signature algorithm**, such as RSA or DSA. These hash functions have since been widely used for many other "ancillary" applications, including hash-based **message authentication codes**, **pseudo random number generators**, and **key derivation functions**."

"Request for Candidate Algorithm Nominations",
-- NIST, November 2007

Some examples

- Signatures: sign(M) = RSA^{-1}(H(M))
- Message-authentication: tag=H(key,M)
- Commitment: commit(M) = H(M,…)
- Key derivation: AES-key = H(DH-value)
- Removing interaction [Fiat-Shamir, 1987]
  - Take interactive identification protocol
  - Replace one side by a hash function

    Challenge = H(smthng, context)
  - Get non-interactive signature scheme



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 $M_1, M_2, \ldots$, 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

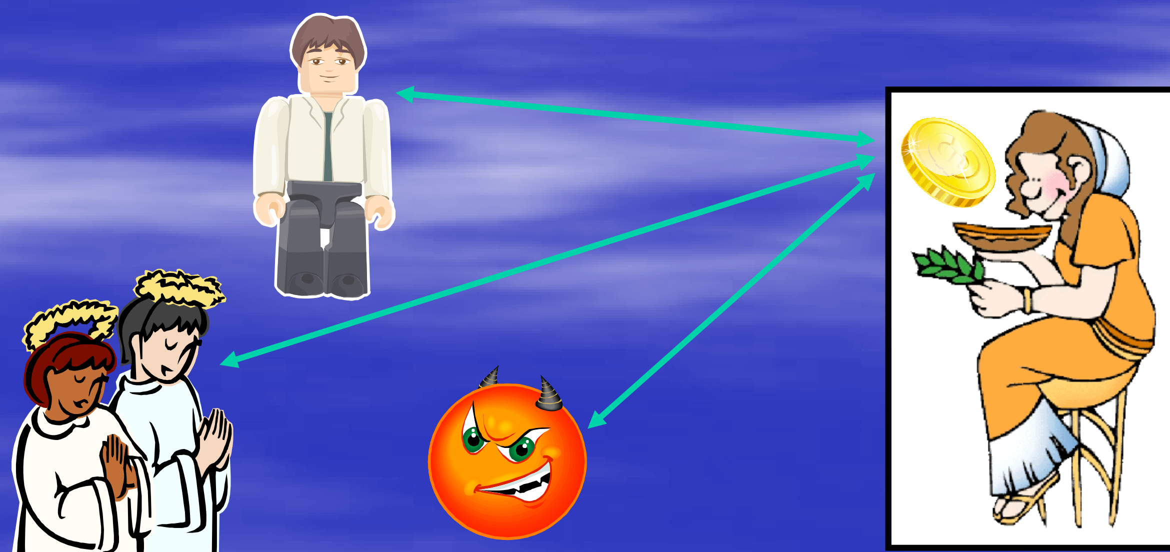
  – Cannot find fixed-points (d = H(d))

  – 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:

  $SIG2^H(key,msg)$: interprate msg as code Π
  - If $\Pi(i)=H(i)$ for $i=1,2,3,\ldots,|msg|$, then output key
  - Else output the same as $SIG1^H(key,msg)$

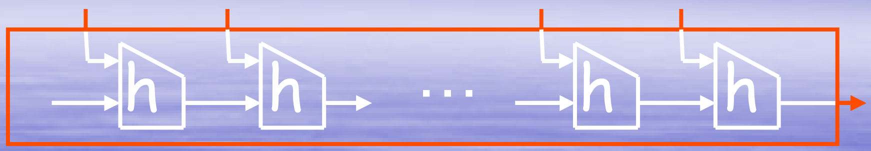
Some Technicalities

- 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(*key*|M) subject to extension attack
      - H(key | M|M') = h(H(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, $h(c,M)=E_M(c)\oplus c$
  - Provide yet more structure, can lead to attacks on provable ROM schemes [H-Krawczyk 2007]

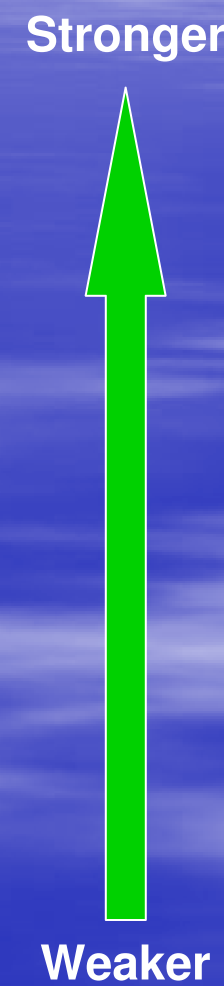
Part II: Using hash functions in applications

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(*salt*,M)
- Hashing random messages
  – M random, d=H(M)
- Hashing with a secret key
  – Attacker chooses M, d=H(*key*,M)



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 $M \mapsto H(\mathit{key}, M)$ for secret *key* looks random to an attacker
- Universal hashing*
  - For all $M \neq M'$, $\Pr_{\mathit{key}}[\, H(\mathit{key}, M) = H(\mathit{key}, M') \,] < \varepsilon$

* Combinatorial, not cryptographic

Application 1: Digital signatures

- Hash-then-sign paradigm
  - First shorten the message, d = H(M)
  - Then sign the digest, s = 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')

same salt (since salt is explicitly signed)

- Attack is inherently on-line
  - Only rely on target collision resistance

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')

attacker can use different salt‘

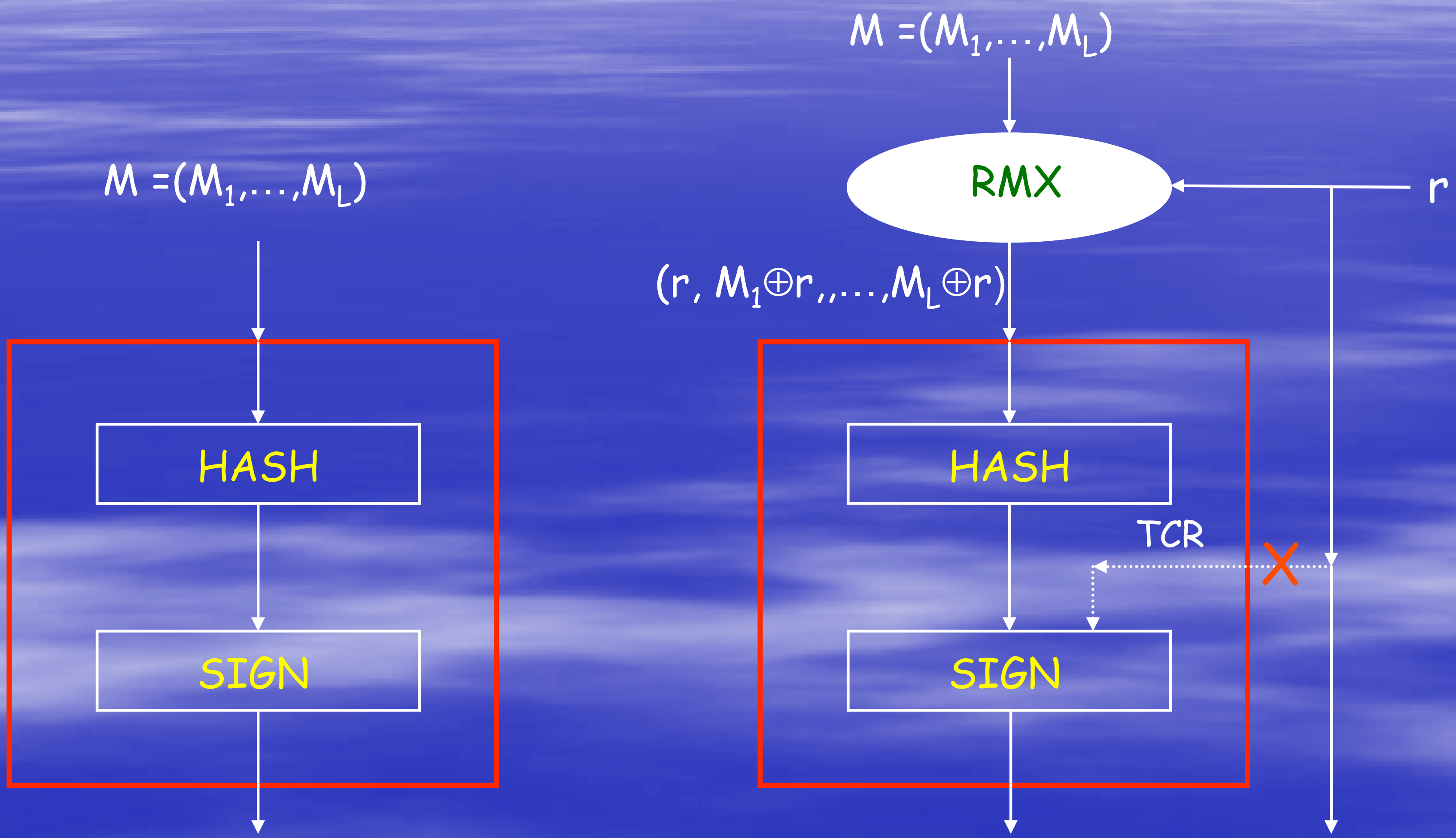
- Attack is still inherently on-line

Randomized hashing with RMX

[H-Krawczyk 2006]

- Use simple message-randomization
  - RMX: $M=(M_1,M_2,\ldots,M_L)$, $r \mapsto$
    $(r, M_1 \oplus r, M_2 \oplus r, \ldots, M_L \oplus r)$
- Hash(RMX(r,M)) is eTCR when:
  - Hash is Merkle-Damgård, and
  - Compression function is ~ 2nd-preimage-resistant
- Signature: [r, SIGN(Hash(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 = 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
  - – tag=H(*key* | M) subject to extension attacks
  - – tag=H(M | *key*) relies on collision resistance
- HMAC: Compute tag = H(key | H(key | M))
  - – About as fast as key-prepend for a MD hash
- Relies only on PRF quality of hash
  - – M↦H(key|M) looks random when key is secret

Authentication with HMAC

[Bellare-Canetti-Krawczyk 1996]

- Simple key-prepend/append have problems when used ... amgård hash
  - ta ... attacks
  - t ... stance
- HMA ... (key | M))
  - About as fas ... or a MD hash
- Relies only on PRF property of hash
  - M↦H(key|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 = $t \oplus PRF(key_2,\text{nonce})$
  - 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 $M \neq M'$, Δ, $\Pr_{key}[\, H(key,M) \oplus H(key,M') = \Delta \,] < \varepsilon$)
  - 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

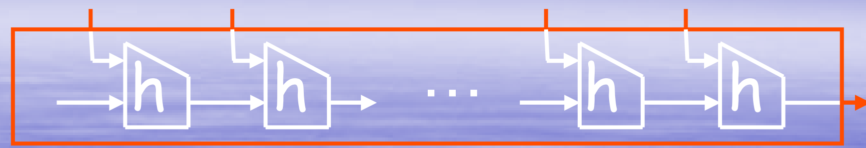
  From inner-product, $H_{k1,k2}(M_1,M_2)=(K_1+M_1)\cdot(K_2+M_2)$
  - [H-Krawczyk'97, Black et al.'99, …]

  From polynomial evaluation $H_k(M_1,\ldots,M_L)=\Sigma_i\, M_i\, k^i$
  - [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→hash has its problems

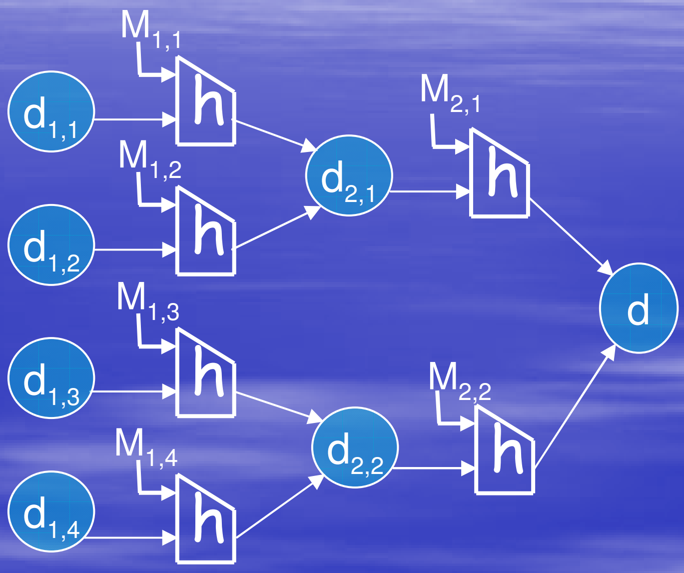
- It's not free (e.g. message encoding)
- Some attacks based on the MD structure
  - Extension attacks (rely on $H(x|y)=h(H(x),y)$)
  - "Birthday attacks" (herding, multicollisions, …)

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

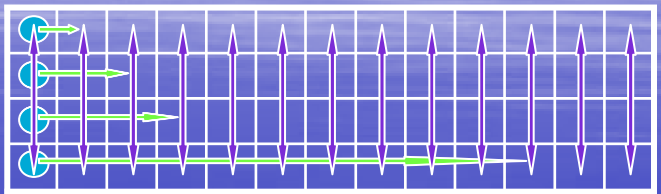


The culprit: small intermediate state

- 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]



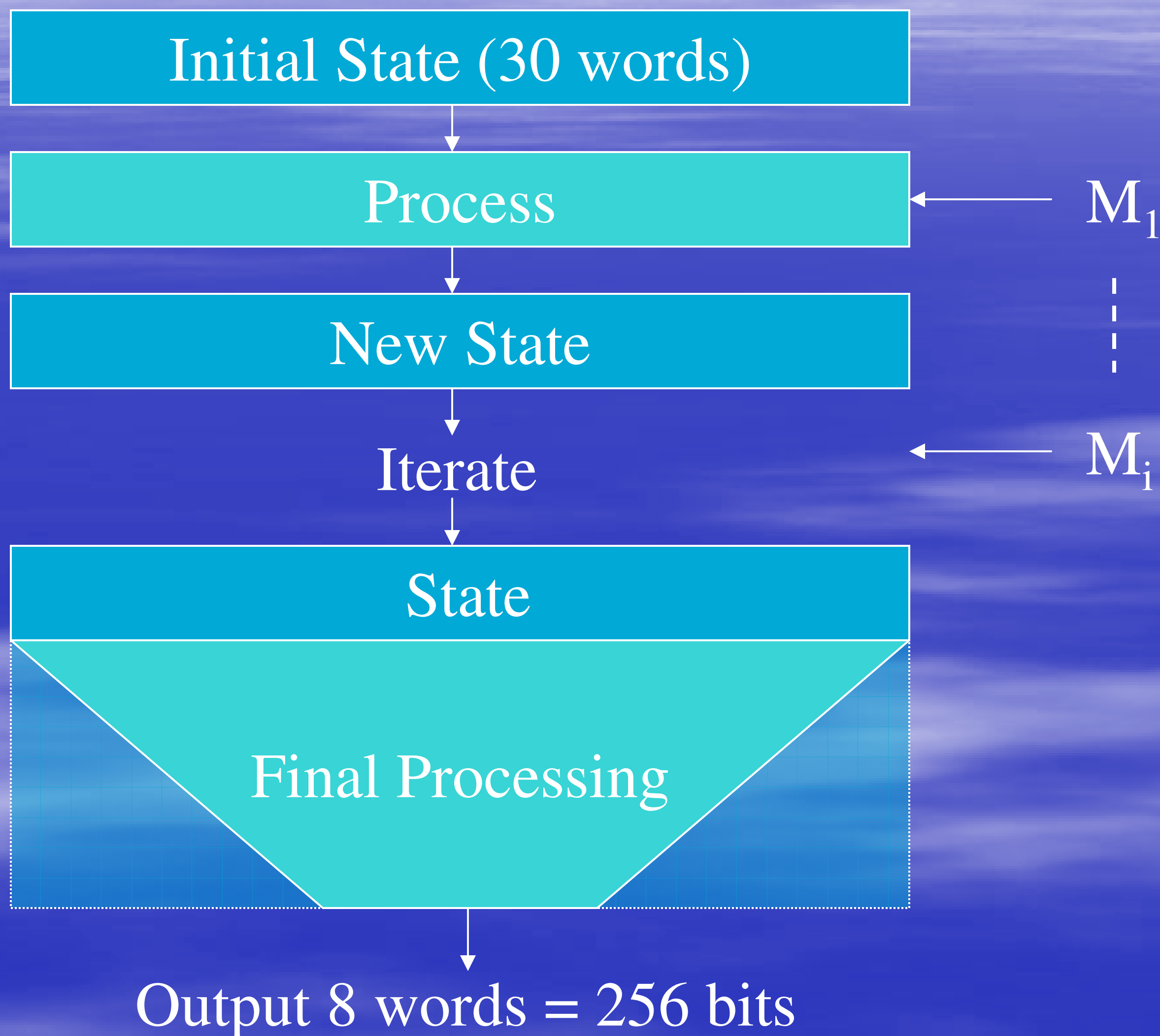
- State is 13 words = 52 bytes
- Process one 4-byte word at a time
  - One AES-like mixing step per word of input
- After some final processing, output 8 words
- Collision attack by Peyrin (2007)
  - Complexity ~ 2^{112} (still better than brute-force)
    - Recently improved to ~ 2^{100} [Khovratovich 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

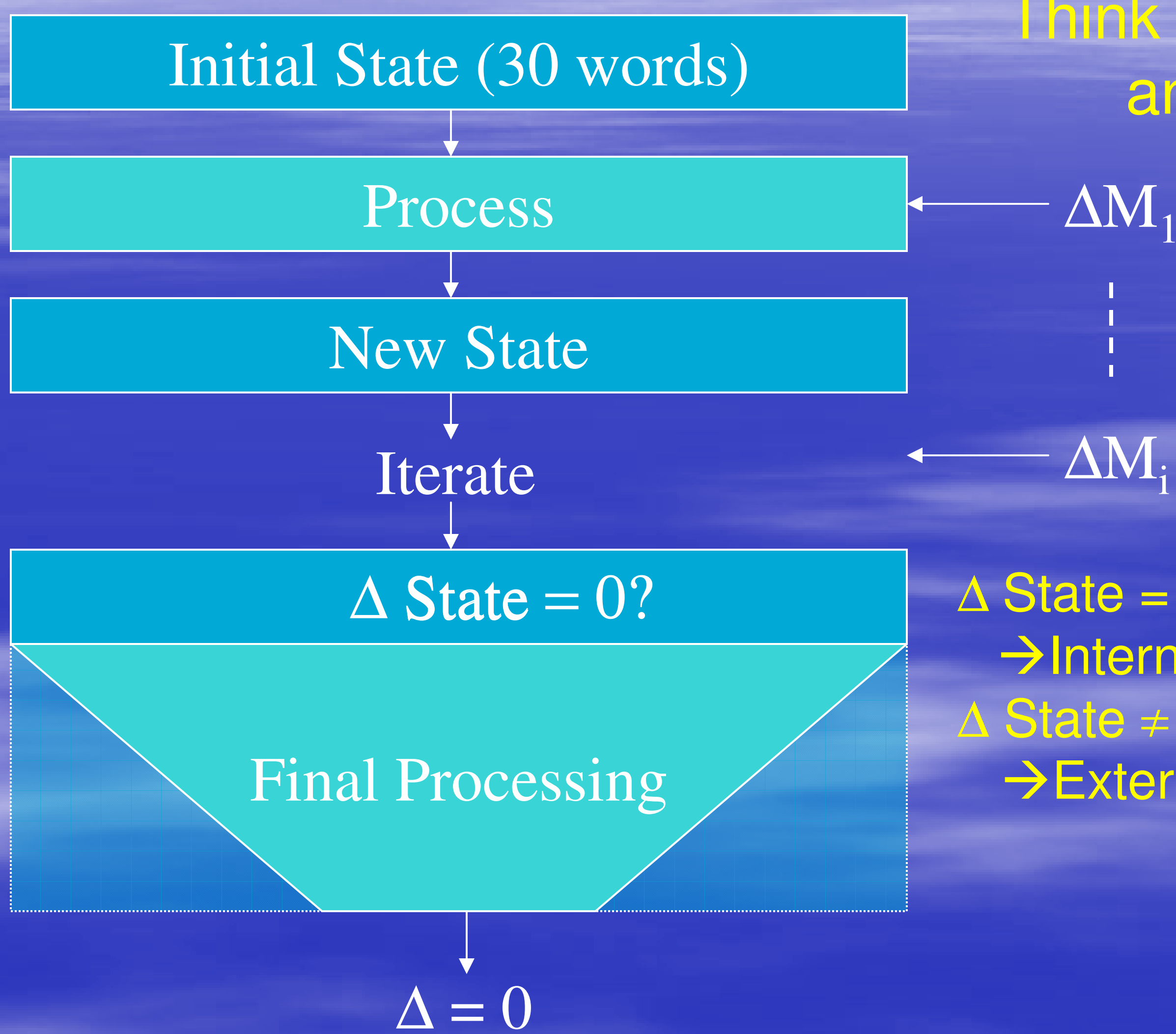
Fugue-256



Initial State (30 words)
Process
M_1
New State
Iterate
M_i
State
Final Processing
Output 8 words = 256 bits

Collision attacks

Think of $M_1, \ldots, M_L$ and $M'_1, \ldots, M'_L$

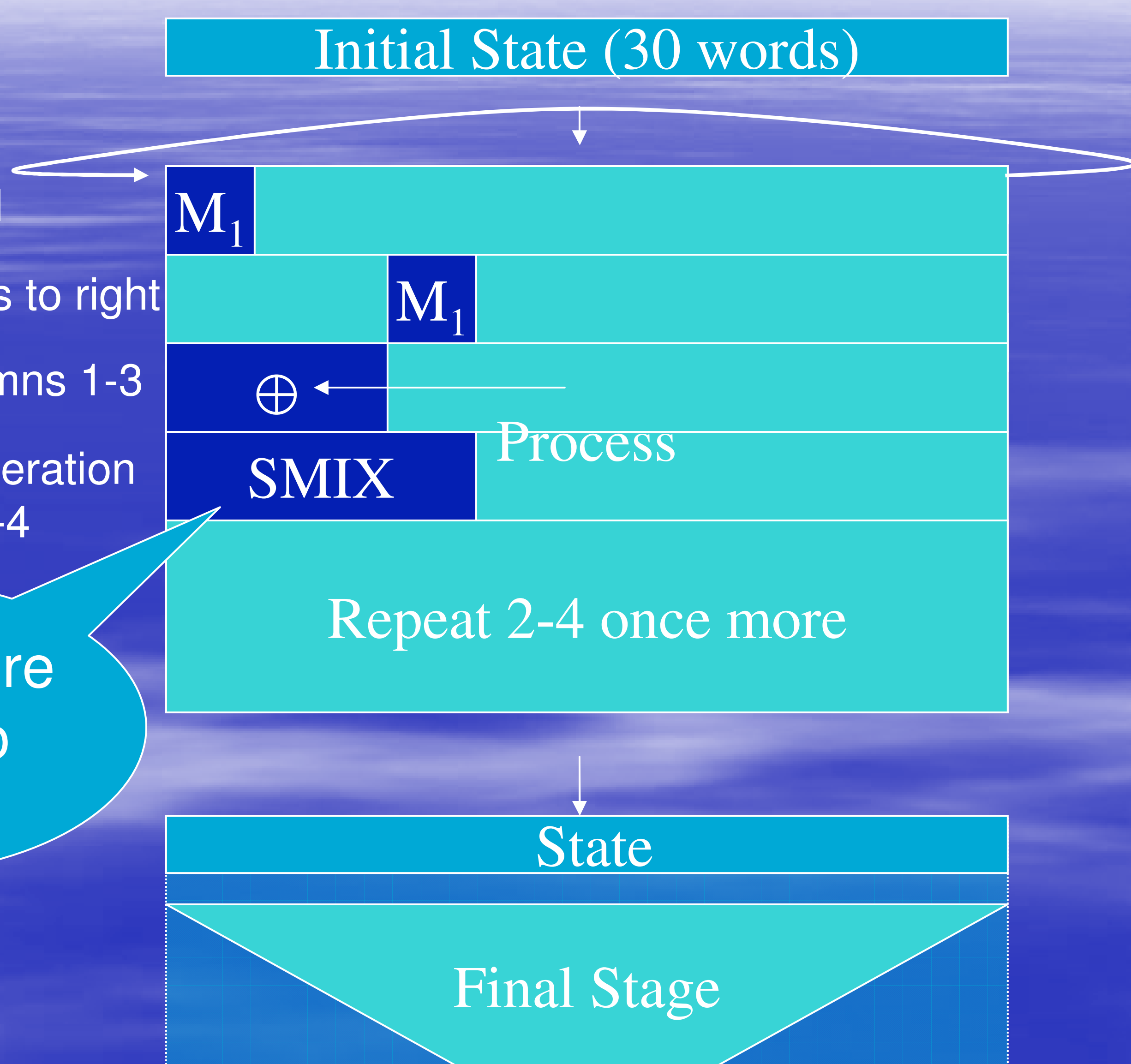


Collision means that ΔM_i's are not all zero

Δ State = 0
→Internal collision
Δ State ≠ 0
→External collision

Processing one input word

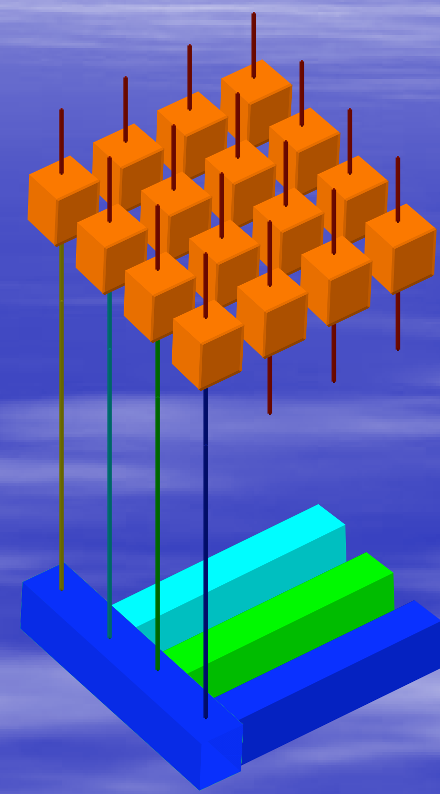
1. Input one word
2. Shift 3 columns to right
3. XOR into columns 1-3
4. “super-mix” operation on columns 1-4



This is where the crypto happens

SMIX in Fugue

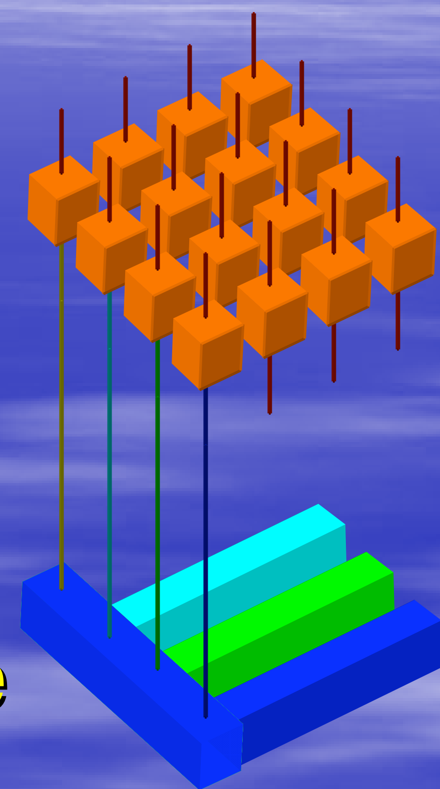
- Similar to one AES round
  - Works on a 4x4 matrix of bytes
  - Starts with S-box substitution
    - `Byte b, S[256] = {...};`
      `...`
    - `b = S[b];`
  - Does linear mixing
- Stronger mixing than AES
  - Diagonal bytes as in AES
  - Other bytes are mixed into both column and row



SMIX in Fugue

- In algebraic notation:

$$\begin{pmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_{16} \end{pmatrix} = \mathbf{M}_{16x16} \times \begin{pmatrix} S[b_1] \\ S[b_2] \\ \vdots \\ S[b_{16}] \end{pmatrix}$$



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

Analyzing internal collisions*

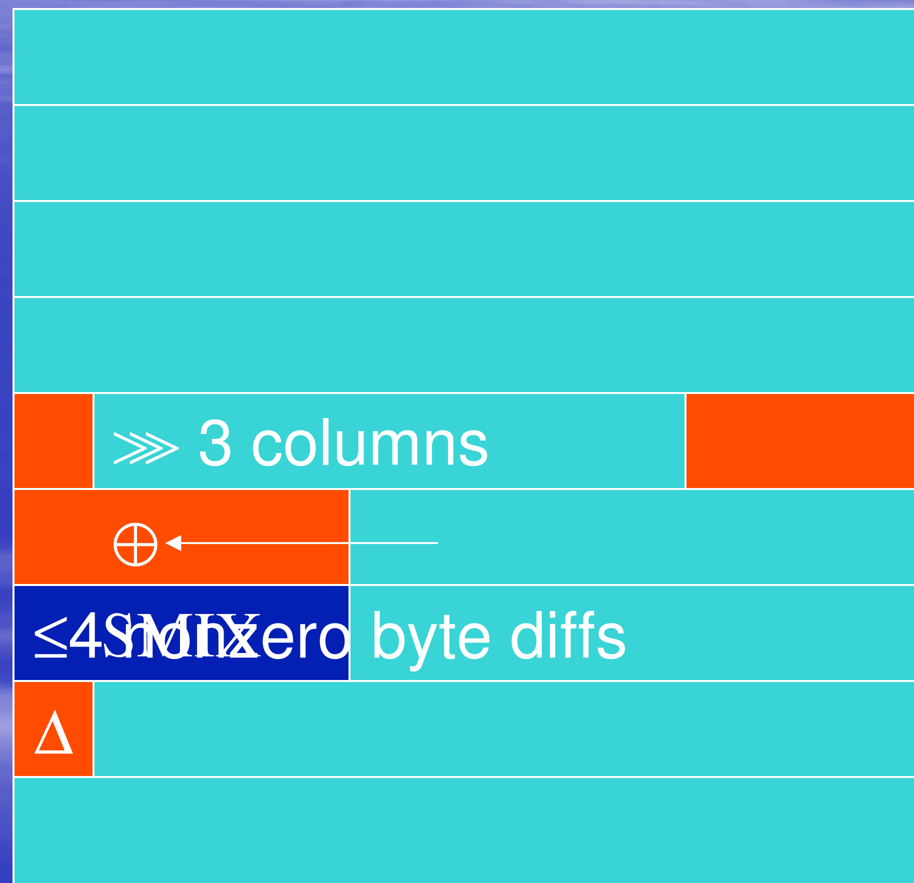
now $\Delta_{28\text{-}1} \neq 0$

still $\Delta_{1\text{-}4} \neq 0$

before SMIX: $\Delta_{1\text{-}4} \neq 0$

before input word: $\Delta_1 \neq 0$

After last input word: ΔState=0



* a bit oversimplified

Analyzing internal collisions*

$\Delta_{25\text{-}1} \neq 0$

$\Delta_{28\text{-}4} \neq 0$

$\Delta_{28\text{-}4} \neq 0$

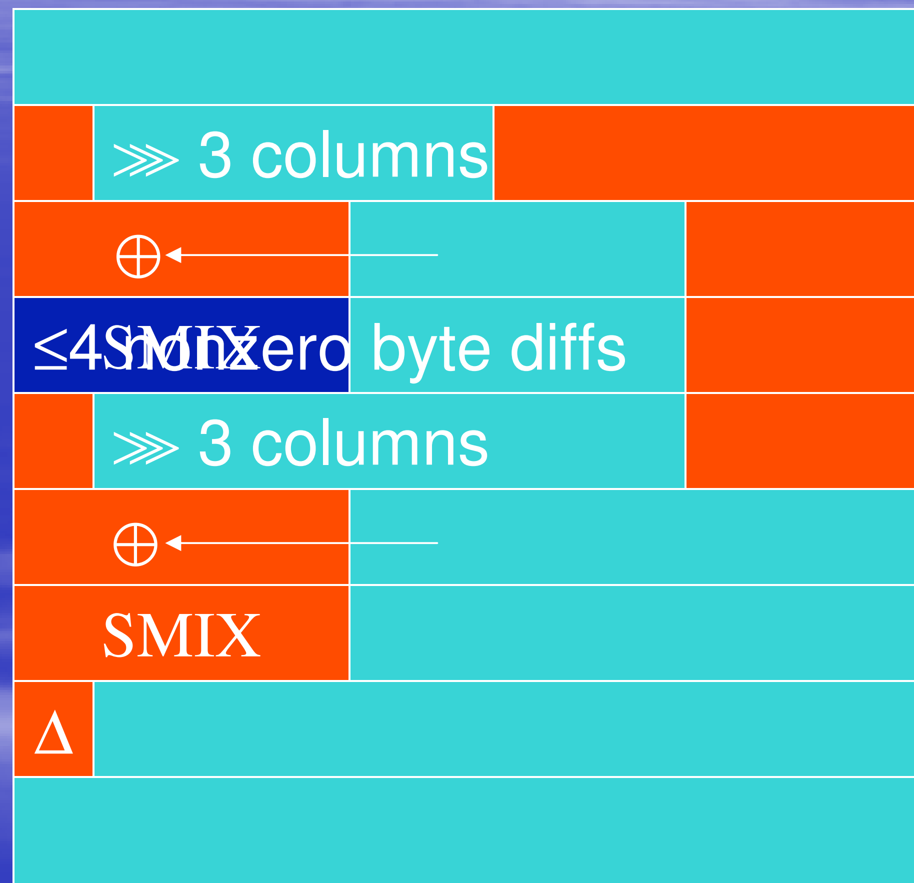
now $\Delta_{28\text{-}1} \neq 0$

still $\Delta_{1\text{-}4} \neq 0$

before SMIX: $\Delta_{1\text{-}4} \neq 0$

before input word: $\Delta_{1} \neq 0$

after input word: ΔState=0



* a bit oversimplified

Analyzing internal collisions*

before input: $\Delta_1=?$, $\Delta_{25\text{-}30}\neq 0$

$\Delta_{25\text{-}1}\neq 0$

$\Delta_{28\text{-}4}\neq 0$

$\Delta_{28\text{-}4}\neq 0$

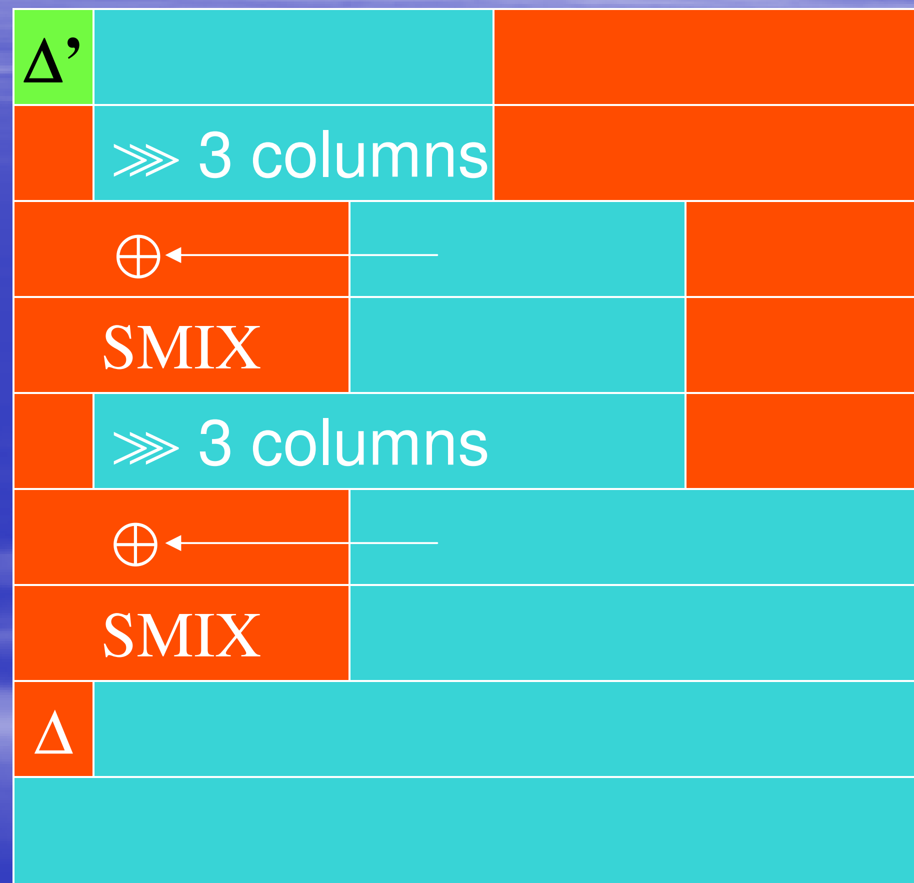
now $\Delta_{28\text{-}1}\neq 0$

still $\Delta_{1\text{-}4}\neq 0$

before SMIX: $\Delta_{1\text{-}4}\neq 0$

before input word: $\Delta_1\neq 0$

after input word: ΔState=0



* a bit oversimplified

The analysis from previous slides was upto here

Many nonzero byte differences before the SMIX operations

Table 8: Evolution of D

	0
TIX_0	
$\Delta S[-1]$	$X1$ 0 0 0
SMIX	
	$Y1_0$ $Y1_1$ $Y1_2$ $Y1_3$
CMIX	
	$Y1_0$ $Y1_1$ $Y1_2$ $Y1_3$
ROR3	
	$\boxed{Y1_3}$ 0 0 0 … $X1$ … $Y1_0 Y1_1 Y1_2$
SMIX	
	$Z1_0$ $Z1_1$ $Z1_2$ $Z1_3$ … $X1$ … $Y1_0 Y1_1 Y1_2$
CMIX	
	$Z1_0$ $Z1_1$ $Z1_2$ $Z1_3$ … $X1$ … $Y1_0 Y1_1 Y1_2$
ROR3	
	$Z1_3$ 0 0 0 $X1$ … $Y1_0 Y1_1 Y1_2 Z1_0 Z1_1 Z1_2$
TIX_{-1}	

	0	3	6	9	12	15	18	21	24	27	29
$\Delta S[-2]$	$x2$ $Y1_0$ 0	0 $X1$		$Z1_3$ 0 $x2$					$Y1_0 Y1_1 Y1_2 Z1_0 Z1_1 Z1_2$		
SMIX											
	$y2_0$ $y2_1$ $y2_2$	$y2_3$ $X1$		$Z1_3$ 0 $x2$					$Y1_0 Y1_1 Y1_2 Z1_0 Z1_1 Z1_2$		
CMIX											
	$y2'_0$ $y2_1$ $y2_2$	$y2_3$ $X1$		$Z1_3$ 0 $x2$		$X1$			$Y1_0 Y1_1 Y1_2 Z1_0 Z1_1 Z1_2$		
ROR3											
	$\boxed{y2_3}$ $X1$ 0	0 0 $Z1_3$ 0 $x2$ 0			$X1$			$Y1_0 Y1_1 Y1_2 Z1_0 Z1_1 Z1_2 y2'_0 y2_1 y2_2$			
SMIX											
	$z2_0$ $z2_1$ $z2_2$	$z2_3$ 0 $Z1_3$ 0 $x2$ 0			$X1$			$Y1_0 Y1_1 Y1_2 Z1_0 Z1_1 Z1_2 y2'_0 y2_1 y2_2$			
CMIX											
	$z2_0$ $z2'_1$ $z2_2$	$z2_3$ 0 $Z1_3$ 0 $x2$ 0			$X1$	$Z1_3$		$Y1_0 Y1_1 Y1_2 Z1_0 Z1_1 Z1_2 y2'_0 y2_1 y2_2$			
ROR3											
	$z2_3$ 0 $Z1_3$	0 $x2$		$X1$	0 $Z1_3$		$Y1_0 Y1_1 Y1_2 Z1_0 Z1_1 Z1_2 y2'_0 y2_1 y2_2 z2_0 z2'_1 z2_2$				
$TIX^{\dagger\dagger}$											

[a] 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 earier (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. [††] 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_{-2}	$z2_3$	0	$Z1_3$	0	$x2$					$X1$				$Z1_3$					$Y1_0$	$Y1_1$	$Y1_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$
$\Delta S[-3]$ SMIX	$x3$	$y2'_0$	$Z1_3$	0	$x2$				$z2_3^\dagger$	$X1$	$x3$			$Z1_3$					$Y1_0$	$Y1_1$	$Y1_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$
CMIX	$y3_0$	$y3_1$	$y3_2$	$y3_3$	$x2$				$z2_3$	$X1$	$x3$			$Z1_3$					$Y1_0$	$Y1_1$	$Y1_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$
ROR3	$y3'_0$	$y3_1$	$y3_2$	$y3_3$	$x2$				$z2_3$	$X1$	$x3$			$Z1_3$	0	$x2$			$Y1_0$	$Y1_1$	$Y1_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$
SMIX	$\boxed{y3_3}$	$x2$	0	0		$z2_3$	$X1$	$x3$			$Z1_3$	0	$x2$			$Y1_0$	$Y1_1$	$Y1_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$
CMIX	$z3_0$	$z3_1$	$z3_2$	$z3_3$		$z2_3$	$X1$	$x3$			$Z1_3$	0	$x2$			$Y1_0$	$Y1_1$	$Y1_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$
ROR3	$z3_0$	$z3'_1$	$z3'_2$	$z3_3$		$z2_3$	$X1$	$x3$			$Z1_3$	0	$x2$			$Y1_0$	$y1'_1$	$y1'_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$
TIX_{-3}	$z3_3$	0	$z2_3$	$X1$	$x3$	0	0	$Z1_3$	0	$x2$	0	0	$Y1_0$	$y1'_1$	$y1'_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$	$z3_0$	$z3'_1$	$z3'_2$
$\Delta S[-4]$ SMIX	$x4$	$y3'_0$	$\boxed{z2_3 \mid X1}$		$x3$	0	0	$Z1_3$	$z3_3$	$x2$	$x4$	0	$Y1_0$	$y1'_1$	$y1'_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$	$z3_0$	$z3'_1$	$z3'_2$
CMIX	$y4_0$	$y4_1$	$y4_2$	$y4_3$	$x3$	0	0	$Z1_3$	$z3_3$	$x2$	$x4$	0	$Y1_0$	$y1'_1$	$y1'_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$	$z3_0$	$z3'_1$	$z3'_2$
ROR3	$y4'_0$	$y4_1$	$y4_2$	$y4_3$	$x3$	0	0	$Z1_3$	$z3_3$	$x2$	$x4$	0	$Y1_0$	$y1'_1$	$y1'_2$	$z1'_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$	$z3_0$	$z3'_1$	$z3'_2$
SMIX	$\boxed{y4_3 \mid x3}$		0	0	$Z1_3$	$z3_3$	$x2$	$x4$	0	$Y1_0$	$y1'_1$	$y1'_2$	$z1'_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$	$z3_0$	$z3'_1$	$z3'_2$	$y4'_0$	$y4_1$	$y4_2$
CMIX	$z4_0$	$z4_1$	$z4_2$	$z4_3$	$Z1_3$	$z3_3$	$x2$	$x4$	0	$Y1_0$	$y1'_1$	$y1'_2$	$z1'_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$	$z3_0$	$z3'_1$	$z3'_2$	$y4'_0$	$y4_1$	$y4_2$
ROR3	$z4'_0$	$z4'_1$	$z4'_2$	$z4_3$	$Z1_3$	$z3_3$	$x2$	$x4$	0	$Y1_0$	$y1'_1$	$y1'_2$	$z1'_0$	$Z1_1$	$Z1_2$	$y2''_0$	$y2'_1$	$y2'_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$	$z3_0$	$z3'_1$	$z3'_2$	$y4'_0$	$y4_1$	$y4_2$
TIX_{-4}	$\boxed{z4_3}$	$Z1_3$	$z3_3$	$x2$	$x4$	0	$Y1_0$	$y1'_1$	$y1'_2$	$z1'_0$	$Z1_1$	$Z1_2$	$y2''_0$	$y2'_1$	$y2'_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$	$z3_0$	$z3'_1$	$z3'_2$	$y4'_0$	$y4_1$	$y4_2$	$z4'_0$	$z4'_1$	$z4'_2$
$\Delta S[-5]$	$x5$	$Z1_3 + y4'_0$	$z3_3$	$x2$	$x4$	0	$Y1_0$	$y1'_1$	$y1'_2 + z4_3$	$z1'_0$	$Z1_1 + x5$	$Z1_2$	$y2''_0$	$y2'_1$	$y2'_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$	$z3_0$	$z3'_1$	$z3'_2$	$y4'_0$	$y4_1$	$y4_2$	$z4'_0$	$z4'_1$	$z4'_2$
	0			3			6			9			12			15			18			21			24			27		29

[a] 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 $M_1, M_2, \ldots$ and $M'_1, M'_2, \ldots$
  – Until $\text{State}_L \oplus \text{State}'_L$ = some "good Δ"
  – Then it searches for suffixed $(M_{L+1}, \ldots, M_{L+4})$, $(M'_{L+1}, \ldots, M'_{L+4})$ that will induce internal collision

Theorem*: For any fixed Δ,
  $\Pr[\,\exists$ 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^{nd} 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

Thank you!