Finding Bugs in Cryptographic Hash Function Implementations

Nicky Mouha[1] M S Raunak[1,2] D. Richard Kuhn[1] Raghu Kacker[1]

[1]NIST, Gaithersburg, USA

[2]Loyola University Maryland, Baltimore, USA

ASK 2017
December 9, 2017

Outline

Introduction

- What are cryptographic hash functions?

Status of Hash Functions

- MD4, MD5, SHA-0, SHA-1, SHA-2, SHA-3,...

Finding Hash Function Implementation Bugs

- New testing approach

What are Hash Functions?

(Cryptographic) Hash Function

- Turn a message into a unique 'fingerprint'
- Input: variable-size message
- Output: fixed-size hash value (e.g., 256 bits)

What are Hash Functions?

(Cryptographic) Hash Function

- Turn a message into a unique 'fingerprint'
- Input: variable-size message
- Output: fixed-size hash value (e.g., 256 bits)

Examples

- $H(\text{“tax”}) =$
  `06565e5611f23fdf8cc43e5077b92b54`

What are Hash Functions?

(Cryptographic) Hash Function

- Turn a message into a unique 'fingerprint'
- Input: variable-size message
- Output: fixed-size hash value (e.g., 256 bits)

Examples

- $H(\text{“tax”}) =$
  `06565e5611f23fdf8cc43e5077b92b54`
- $H(\text{“fax”}) =$
  `236c3b7f761221f195b428aca2f06c4b`

What are Hash Functions?

(Cryptographic) Hash Function

- Turn a message into a unique 'fingerprint'
- Input: variable-size message
- Output: fixed-size hash value (e.g., 256 bits)

Examples

- $H(\text{“tax”}) =$
  `06565e5611f23fdf8cc43e5077b92b54`
- $H(\text{“fax”}) =$
  `236c3b7f761221f195b428aca2f06c4b`
- $H(\text{“taxi”}) =$
  `35e768d3043c55b5540979da7feaf81c`

What are Hash Functions?

(Cryptographic) Hash Function

- Turn a message into a unique 'fingerprint'
- Input: variable-size message
- Output: fixed-size hash value (e.g., 256 bits)

Examples

- $H(\text{“tax”}) =$
  `06565e5611f23fdf8cc43e5077b92b54`
- $H(\text{“fax”}) =$
  `236c3b7f761221f195b428aca2f06c4b`
- $H(\text{“taxi”}) =$
  `35e768d3043c55b5540979da7feaf81c`
- $H(\text{“3.14159...”}\ (10\,000\ \text{digits})) =$
  `fceb6f18bfb443fd5bcaa1dd97041ca8`

What are Hash Functions?

Cryptographic Properties

- Preimage resistance
- Second preimage resistance
- Collision resistance

What are Hash Functions?

Cryptographic Properties

- Preimage resistance
- Second preimage resistance
- Collision resistance

Functionality

- Efficient to compute
- Compact description

What are Hash Functions?

Cryptographic Properties

- Preimage resistance
- Second preimage resistance
- Collision resistance

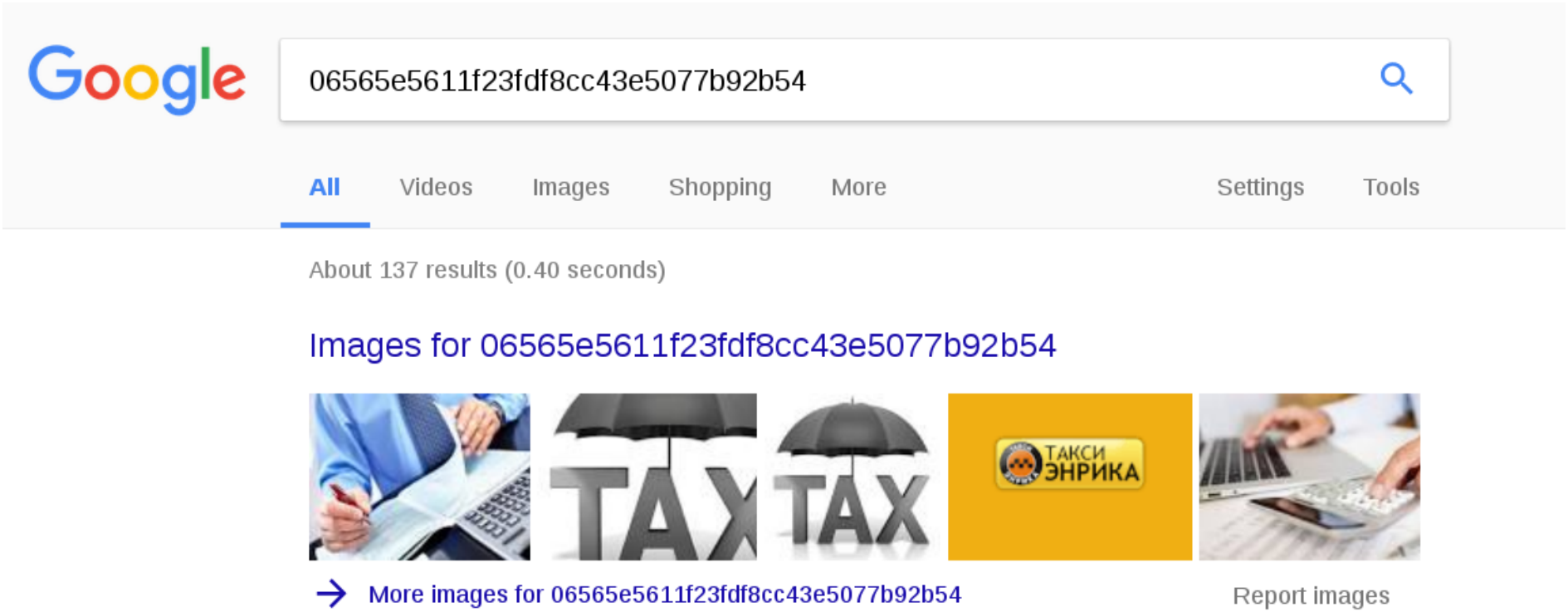
Functionality

- Efficient to compute
- Compact description

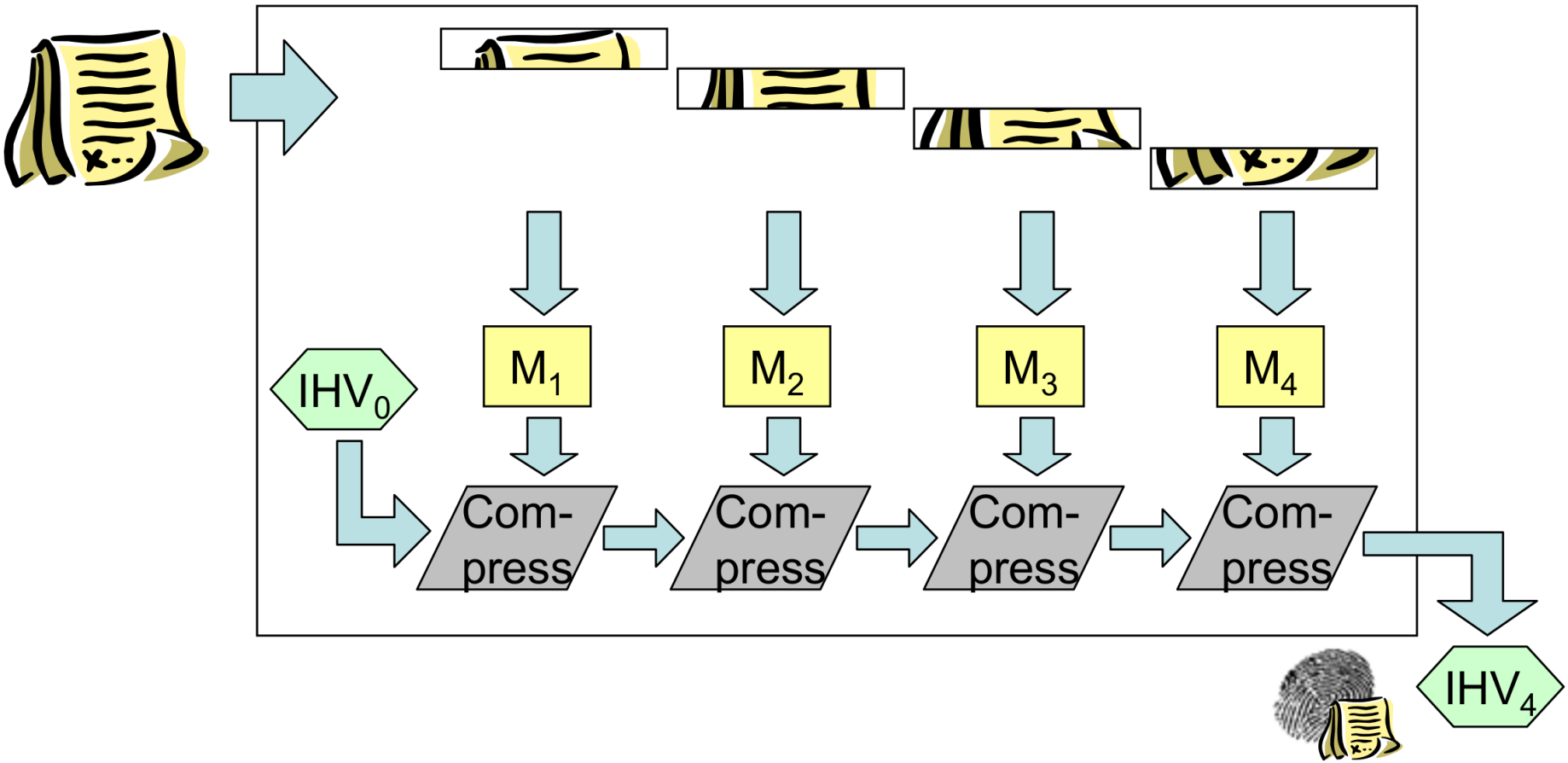
Implicit Assumption

- Input uniquely determines output

What are Hash Functions?



Iterated Hash Functions



Hash Functions: Current Status

Name	Released	Collision attack
MD4		
MD5		
SHA-0		
SHA-1		
SHA-2		
SHA-3		

Hash Functions: Current Status

Name	Released	Collision attack
MD4	Rivest (1990)	
MD5		
SHA-0		
SHA-1		
SHA-2		
SHA-3		

Hash Functions: Current Status

Name	Released	Collision attack
MD4	Rivest (1990)	
MD5	Rivest (1991)	
SHA-0		
SHA-1		
SHA-2		
SHA-3		

Hash Functions: Current Status

Name	Released	Collision attack
MD4	Rivest (1990)	
MD5	Rivest (1991)	
SHA-0	NSA (1993)	
SHA-1		
SHA-2		
SHA-3		

Hash Functions: Current Status

Name	Released	Collision attack
MD4	Rivest (1990)	
MD5	Rivest (1991)	
SHA-0	NSA (1993)	
SHA-1	NSA (1995)	
SHA-2		
SHA-3		

Hash Functions: Current Status

Name	Released	Collision attack
MD4	Rivest (1990)	Dobbertin (1995)
MD5	Rivest (1991)	
SHA-0	NSA (1993)	
SHA-1	NSA (1995)	
SHA-2		
SHA-3		

Hash Functions: Current Status

Name	Released	Collision attack
MD4	Rivest (1990)	Dobbertin (1995)
MD5	Rivest (1991)	
SHA-0	NSA (1993)	Chabaud-Joux (1998) (“theoretical”)
SHA-1	NSA (1995)	
SHA-2		
SHA-3		

Hash Functions: Current Status

Name	Released	Collision attack
MD4	Rivest (1990)	Dobbertin (1995)
MD5	Rivest (1991)	
SHA-0	NSA (1993)	Chabaud-Joux (1998) (“theoretical”)
SHA-1	NSA (1995)	
SHA-2	NSA (2001)	
SHA-3		

Hash Functions: Current Status

Name	Released	Collision attack
MD4	Rivest (1990)	Dobbertin (1995)
MD5	Rivest (1991)	Wang et al. (2004)
SHA-0	NSA (1993)	Chabaud-Joux (1998) (“theoretical”)
SHA-1	NSA (1995)	
SHA-2	NSA (2001)	
SHA-3		

Hash Functions: Current Status

Name	Released	Collision attack
MD4	Rivest (1990)	Dobbertin (1995)
MD5	Rivest (1991)	Wang et al. (2004)
SHA-0	NSA (1993)	Chabaud-Joux (1998) ("theoretical") Joux et al. (2004)
SHA-1	NSA (1995)	
SHA-2	NSA (2001)	
SHA-3		

Hash Functions: Current Status

Name	Released	Collision attack
MD4	Rivest (1990)	Dobbertin (1995)
MD5	Rivest (1991)	Wang et al. (2004)
SHA-0	NSA (1993)	Chabaud-Joux (1998) ("theoretical") Joux et al. (2004)
SHA-1	NSA (1995)	Wang et al. (2005) ("theoretical")
SHA-2	NSA (2001)	
SHA-3		

Hash Functions: Current Status

Name	Released	Collision attack
MD4	Rivest (1990)	Dobbertin (1995)
MD5	Rivest (1991)	Wang et al. (2004)
SHA-0	NSA (1993)	Chabaud-Joux (1998) ("theoretical") Joux et al. (2004)
SHA-1	NSA (1995)	Wang et al. (2005) ("theoretical")
SHA-2	NSA (2001)	
SHA-3	Daemen et al. (2008)	

Hash Functions: Current Status

Name	Released	Collision attack
MD4	Rivest (1990)	Dobbertin (1995)
MD5	Rivest (1991)	Wang et al. (2004)
SHA-0	NSA (1993)	Chabaud-Joux (1998) ("theoretical") Joux et al. (2004)
SHA-1	NSA (1995)	Wang et al. (2005) ("theoretical")
SHA-2	NSA (2001)	
SHA-3	Daemen et al. (2008) NIST standard (2014)	

Hash Functions: Current Status

Name	Released	Collision attack
MD4	Rivest (1990)	Dobbertin (1995)
MD5	Rivest (1991)	Wang et al. (2004)
SHA-0	NSA (1993)	Chabaud-Joux (1998) ("theoretical") Joux et al. (2004)
SHA-1	NSA (1995)	Wang et al. (2005) ("theoretical") Stevens et al. (2017)
SHA-2	NSA (2001)	
SHA-3	Daemen et al. (2008) NIST standard (2014)	

Hash Functions: Current Status

Name	Released	Collision attack
MD4	Rivest (1990)	Dobbertin (1995)
MD5	Rivest (1991)	Wang et al. (2004)
SHA-0	NSA (1993)	Chabaud-Joux (1998) ("theoretical") Joux et al. (2004)
SHA-1	NSA (1995)	Wang et al. (2005) ("theoretical") Stevens et al. (2017)
SHA-2	NSA (2001)	No attack
SHA-3	Daemen et al. (2008) NIST standard (2014)	

Hash Functions: Current Status

Name	Released	Collision attack
MD4	Rivest (1990)	Dobbertin (1995)
MD5	Rivest (1991)	Wang et al. (2004)
SHA-0	NSA (1993)	Chabaud-Joux (1998) ("theoretical") Joux et al. (2004)
SHA-1	NSA (1995)	Wang et al. (2005) ("theoretical") Stevens et al. (2017)
SHA-2	NSA (2001)	No attack
SHA-3	Daemen et al. (2008) NIST standard (2014)	No attack

NIST SHA-3 Competition

Timeline

- 2007: Call for submissions

NIST SHA-3 Competition

Timeline

- 2007: Call for submissions
- 2008: 64 submissions received

NIST SHA-3 Competition

Timeline

- 2007: Call for submissions
- 2008: 64 submissions received
- 2008: 51 first-round candidates

NIST SHA-3 Competition

Timeline

- 2007: Call for submissions
- 2008: 64 submissions received
- 2008: 51 first-round candidates
- 2009: 14 second-round candidates

NIST SHA-3 Competition

Timeline

- 2007: Call for submissions
- 2008: 64 submissions received
- 2008: 51 first-round candidates
- 2009: 14 second-round candidates
- 2010: 5 finalists

NIST SHA-3 Competition

Timeline

- 2007: Call for submissions
- 2008: 64 submissions received
- 2008: 51 first-round candidates
- 2009: 14 second-round candidates
- 2010: 5 finalists
- 2012: Winner: Keccak

NIST SHA-3 Competition

Timeline

- 2007: Call for submissions
- 2008: 64 submissions received
- 2008: 51 first-round candidates
- 2009: 14 second-round candidates
- 2010: 5 finalists
- 2012: Winner: Keccak
- 2014: Draft standard

NIST SHA-3 Competition

Timeline

- 2007: Call for submissions
- 2008: 64 submissions received
- 2008: 51 first-round candidates
- 2009: 14 second-round candidates
- 2010: 5 finalists
- 2012: Winner: Keccak
- 2014: Draft standard
- 2015: Final standard: SHA-3

NIST SHA-3 Competition

(Some) Submission Requirements

- Resistance against (second-)preimages and collisions
- Output sizes: 224, 256, 384 and 512 bits (at least)

Submission Package

- A full specification (incl. security analysis)
- At least a reference implementation (in C)
- Answers to NIST-provided test vectors
- Intellectual property statements,...

Testing Implementations

Paper

- "Finding Bugs in Cryptographic Hash Function Implementations"
- Preprint: `https://eprint.iacr.org/2017/891.pdf`

Results

- Tested all SHA-3 candidate reference implementations
- Bugs found in almost half of implementations!

Testing: General Principles

Every Implementation Can Contain Bugs

- Correct refence implementation: important!
- Finding bugs: testing

Testing Requires an Oracle o to Test Function f

- Suppose: program under test produces y for input x, i.e., $f(x) = y$
- Suppose: according to oracle, expected output of program when correctly implemented is y', i.e., $o(x) = y'$
- Test case passes iff $y = y'$
- Automated oracles are rarely available
- Human oracles can be wrong...

Non-testable Programs

Programs with No Simple Way to Design Test Oracles

- Expected output cannot be known a priori
- Test oracle is as expensive as the program

Examples of 'Non-testable' Programs

- Scientific computations
- Machine learning algorithms
- Simulation software and simulation models
- Cryptographic functions

Strategy for testing cryptographic functions

Construct Tests from Cryptographic Properties

- The implementations should satisfy them as well!

Three Types of Tests

- Bit-Contribution Test
- Bit-Exclusion Test
- Metamorphic Update Test

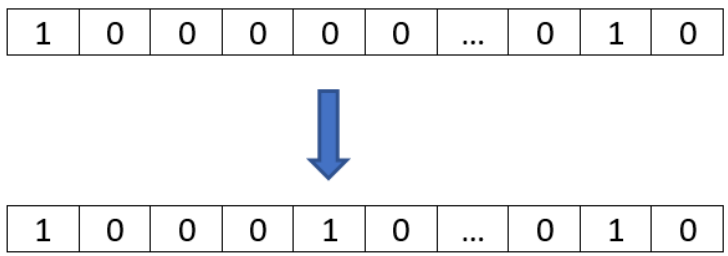
Apply Tests to SHA-3 Competition

- All reference implementations

Bit-Contribution Test

Approach

- Take a message m of length n
- Flip one input bit
- Did the output $H(m)$ change?



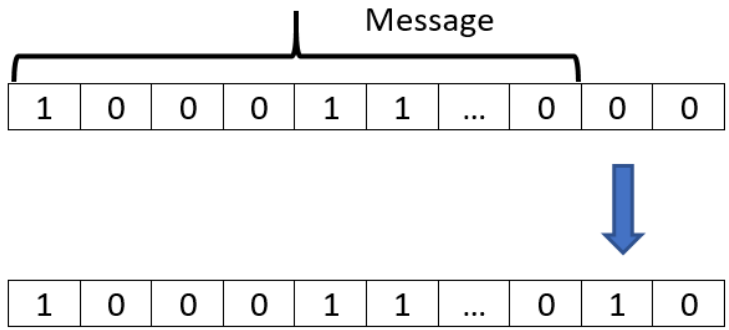
Repeat for:

- Lengths up 2-4$\times$ block size
- Flip every bit of the message

Bit-Exclusion Test

Approach

- Take a message m of length n
- Flip one bit after the last bit of m
- Did the output $H(m)$ stay the same?



Flipping one bit outside the message

Repeat for:

- Lengths up 2-4$\times$ block size
- Flip first bit, second bit,... beyond m

Metamorphic Update Test

Approach

- Take a message m of length n
- Split m into m_1 and m_2
- Compute $\texttt{Hash}(m)$
- Compute $\texttt{Init}$, $\texttt{Update}(m_1)$, $\texttt{Update}(m_2)$, $\texttt{Final}()$
- Are both ouputs the same?

Repeat for:

- Lengths up 2-4$\times$ block size
- All positions where the message can be split

Combinatorial Update Test

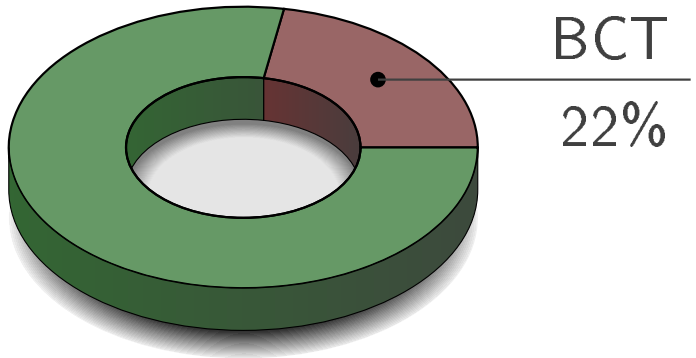
Variant of Metamorphic Update Test

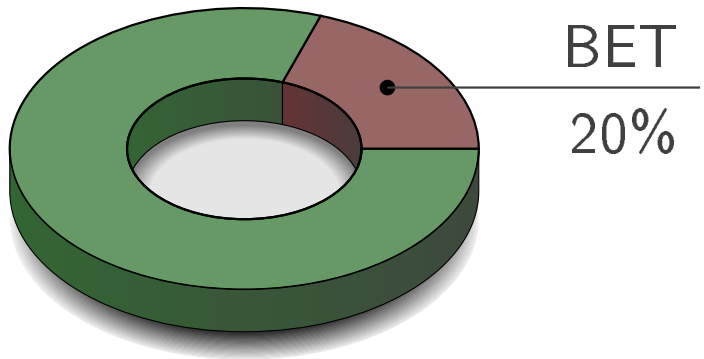
- Split message into several ‘chunks’
- Construct tests using combinatorial approach

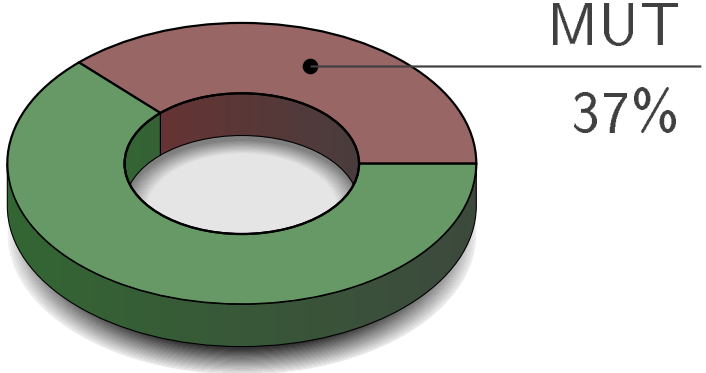
Results

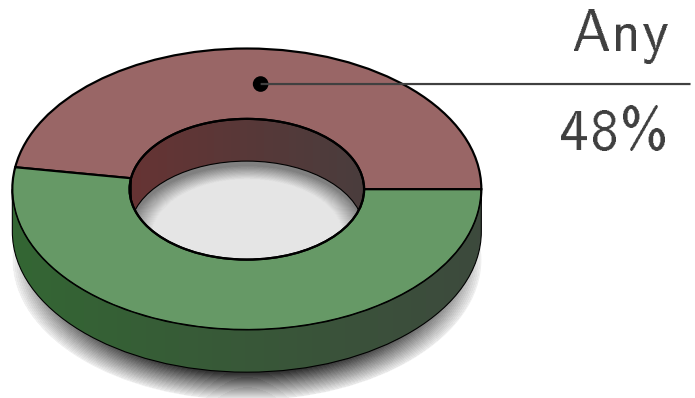
- Same bugs found
- Much smaller number of test cases

Results: All 86 Implementations

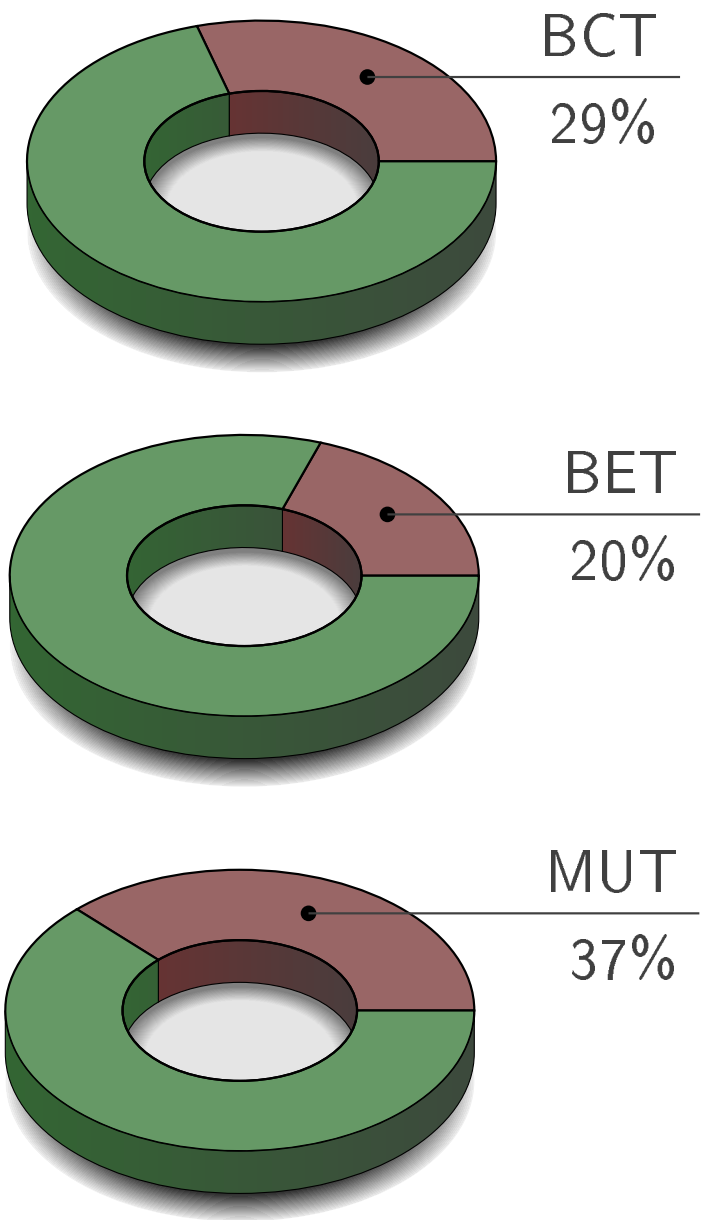


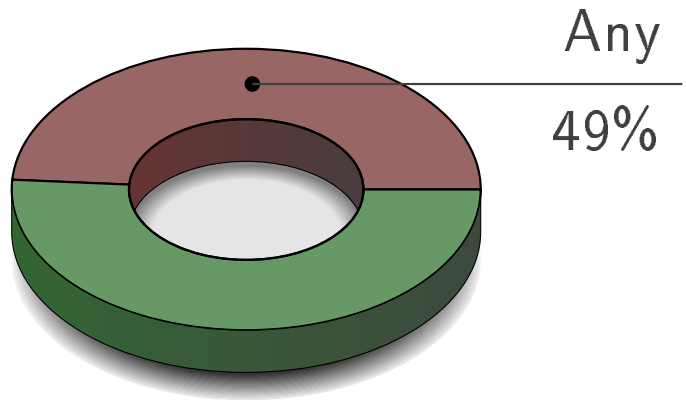




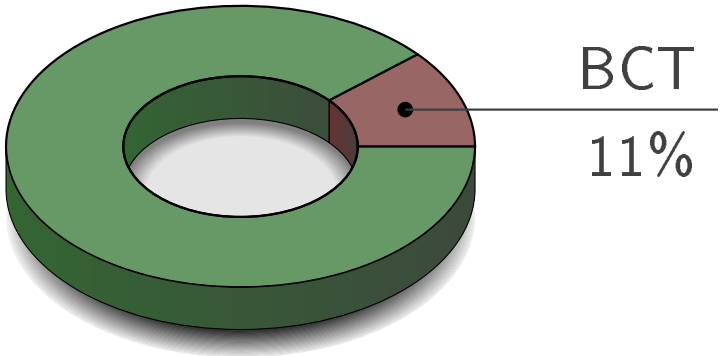


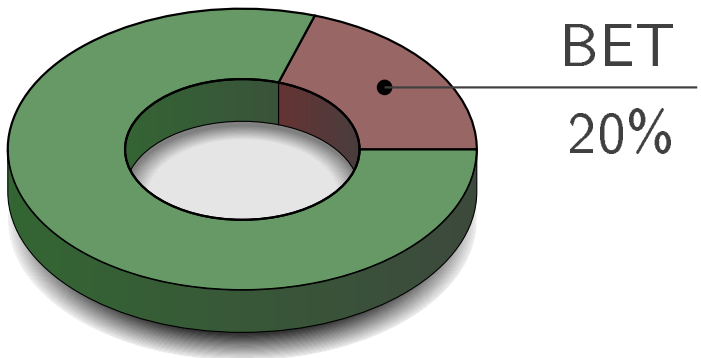
Results: 51 Initial Implementations

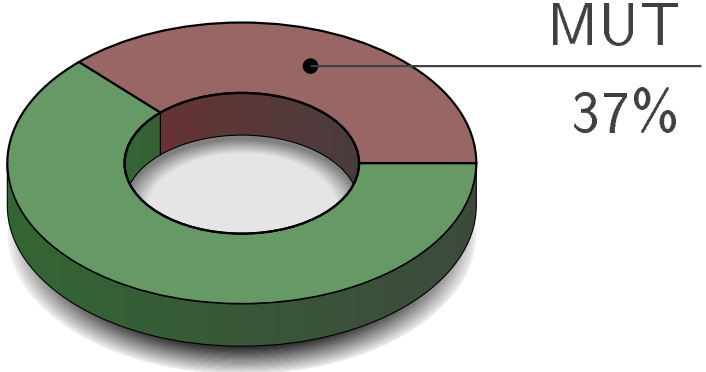


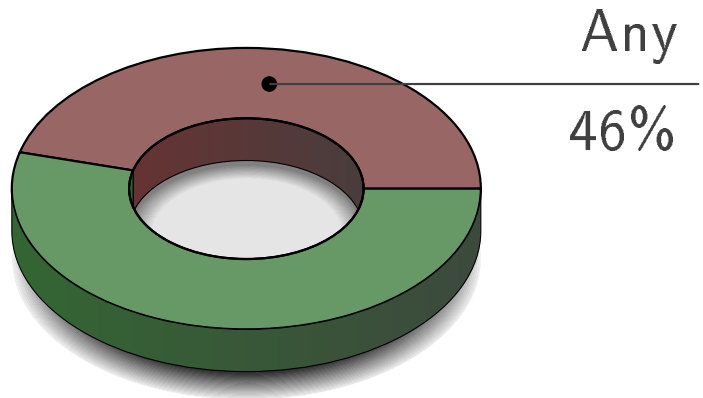


Results: 35 Updated Implementations









Other Tests

Dynamic Analysis Tools

- Buffer errors, memory leaks, null dereferences,...
- Bugs in five submissions (Fortify 2009)

Other Tests

Dynamic Analysis Tools

- Buffer errors, memory leaks, null dereferences,...
- Bugs in five submissions (Fortify 2009)

Code Coverage Testing

- "Cryptographic functions result in abnormally straight line code, it's common for a typical input to exercise every instruction." (Langley)
- Our analysis of SHA-3 finalists: "SHA-3 candidates typically achieve complete code coverage of all API-required functionality"

BLAKE

BLAKE SHA-3 Submission

- Designed by Aumasson et al.
- One of five finalists

Implementation Bug

- Bug in all submitted versions
- Undiscovered for seven years (!)
- Rediscovered by our Metamorphic Update Test

Details...

- “fixed a bug that gave incorrect hashes in specific use cases”
- “found by a careful user”

BLAKE: Source Code

blake_ref.c, lines 298 to 312, incorrect

```
/* compress remaining data filled with new bits */
if( left && ( ((databitlen >> 3) & 0x3F) >= fill ) ) {
  memcpy( (void *) (state->data32 + left),
  (void *) data, fill );
  /* update counter */
  state->t32[0] += 512;
  if (state->t32[0] == 0)
    state->t32[1]++;

  compress32( state, state->data32 );
  data += fill;
  databitlen  -= (fill << 3);

  left = 0;
}
```

BLAKE: Source Code

blake_ref.c, lines 298 to 312, **corrected**

```
/* compress remaining data filled with new bits */
if( left && ( ((databitlen >> 3)       ) >= fill ) ) {
  memcpy( (void *) (state->data32 + left),
  (void *) data, fill );
  /* update counter */
  state->t32[0] += 512;
  if (state->t32[0] == 0)
    state->t32[1]++;

  compress32( state, state->data32 );
  data += fill;
  databitlen  -= (fill << 3);

  left = 0;
}
```

BLAKE: Bug Explained

Example

- Init()
- Call Update() on 1-byte message
- Call Update() on 64-byte message (full block)
- Final()

BLAKE: Bug Explained

Example

- Init()
- Call Update() on 1-byte message
- Call Update() on 64-byte message (full block)
- Final()

Analysis

- First Update() gets “forgotten”

BLAKE: Bug Explained

Example

- Init()
- ~~Call Update() on 1-byte message~~
- Call Update() on 64-byte message (full block)
- Final()

Analysis

- First Update() gets “forgotten”
- Without first Update(): same hash value!

BLAKE: Bug Explained

Example

- Init()
- ~~Call Update() on 1-byte message~~
- Call Update() on 64-byte message (full block)
- Final()

Analysis

- First Update() gets "forgotten"
- Without first Update(): same hash value!

"Second Preimage"

- Modify the message without changing the hash value!

LANE SHA-3 Submission

- Designed by Indesteege
- Implemented by Mouha
- First-round candidate

LANE

LANE SHA-3 Submission

- Designed by Indesteege
- Implemented by Mouha
- First-round candidate

Implementation Bug

- Bug in all submitted versions
- Rediscovered by our Bit-Contribution Test
- Fixed on LANE website
- But still fails Metamorphic Update Test...

LANE

LANE SHA-3 Submission

- Designed by Indesteege
- Implemented by Mouha
- First-round candidate

Implementation Bug

- Bug in all submitted versions
- Rediscovered by our Bit-Contribution Test
- Fixed on LANE website
- But still fails Metamorphic Update Test...

Details...

- Same hash for all 505-bit messages
- More general: 505 to 511-bit messages (modulo 512)

LANE: Source Code

`laneref.c`, lines 135 to 144, incorrect

```
if (state->databitcount & 0x1ff) {
  /* number of bytes in buffer that are (partially) filled */
  const DataLength n =
              (((state->databitcount - 1) >> 3) + 1) & 0x3f;
  if (n < BLOCKSIZE)
    memset(state->buffer + n, 0, BLOCKSIZE-n);
  /* zero-pad partial byte */
  state->buffer[(state->databitcount >> 3) & 0x3f]
    &= ~(0xff >> (state->databitcount & 0x7));
  Lane256Transform(state, state->buffer, state->databitcount);
}
```

LANE: Source Code

`laneref.c`, lines 135 to 144, **corrected**

```
if (state->databitcount & 0x1ff) {
  /* number of bytes in buffer that are (partially) filled */
  const DataLength n =
              (((state->databitcount & 0x1ff) - 1) >> 3) + 1;
  if (n < BLOCKSIZE)
    memset(state->buffer + n, 0, BLOCKSIZE-n);
  /* zero-pad partial byte */
  state->buffer[(state->databitcount >> 3) & 0x3f]
    &= ~(0xff >> (state->databitcount & 0x7));
  Lane256Transform(state, state->buffer, state->databitcount);
}
```

Fugue

Fugue SHA-3 Submission

- Designed by Halevi, Hall and Jutla (IBM)
- One of fourteen second-round candidates

Implementation Bug

- Last incomplete byte: erroneously zeroed out!
- Makes finding second-preimages trivial

Our Test Suite

- Our Bit-Contribution Test finds this bug
- Our Bit-Exclusion Test finds another bug...

Fugue: Source Code

SHA3api_ref.c, lines 72 to 87, incorrect

```
HashReturn Final (hashState *state, BitSequence *hashval)
{
    if (!state || !state->Cfg)
        return FAIL;
    if (state->TotalBits&31)
    {
        int need = 32-(state->TotalBits&31);
        memset ((uint8*)state->Partial
            +  ((state->TotalBits&31)   /8),0,need/8);
        Next_Fugue (state, state->Partial, 1);
    }
    state->TotalBits = BE2HO_8 (state->TotalBits);
    Next_Fugue (state, (uint32*)&state->TotalBits, 2);
    Done_Fugue (state, (uint32*)hashval, NULL);
    return SUCCESS;
}
```

Fugue: Source Code

SHA3api_ref.c, lines 72 to 87, **corrected**

```
HashReturn Final (hashState *state, BitSequence *hashval)
{
    if (!state || !state->Cfg)
        return FAIL;
    if (state->TotalBits&31)
    {
        int need = 32-(state->TotalBits&31);
        memset ((uint8*)state->Partial
            + (((state->TotalBits&31)+7)/8),0,need/8);
        Next_Fugue (state, state->Partial, 1);
    }
    state->TotalBits = BE2HO_8 (state->TotalBits);
    Next_Fugue (state, (uint32*)&state->TotalBits, 2);
    Done_Fugue (state, (uint32*)hashval, NULL);
    return SUCCESS;
}
```

Conclusion

New Method to Test Hash Function Implementations

- Tests based on cryptographic hash function properties
- Systematically search for violations in implementations

Target: SHA-3 Competition

- Tests applied to all reference implementations
- Found bugs in half of implementations
- Preprint: `https://eprint.iacr.org/2017/891.pdf`

Questions?