Gimli: A Cross-Platform Permutation

Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, **Benoît Viguier**

Advances in permutation-based cryptography, Milan, October 10, 2018

What is a Permutation?

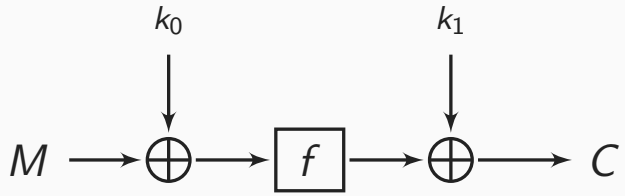
Definition: A Permutation is a keyless block cipher.

What is a Permutation?

Definition: A Permutation is a keyless block cipher.

What is a Permutation?

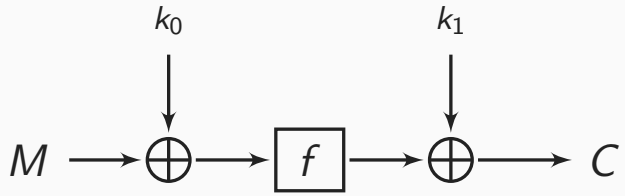
Definition: A Permutation is a keyless block cipher.



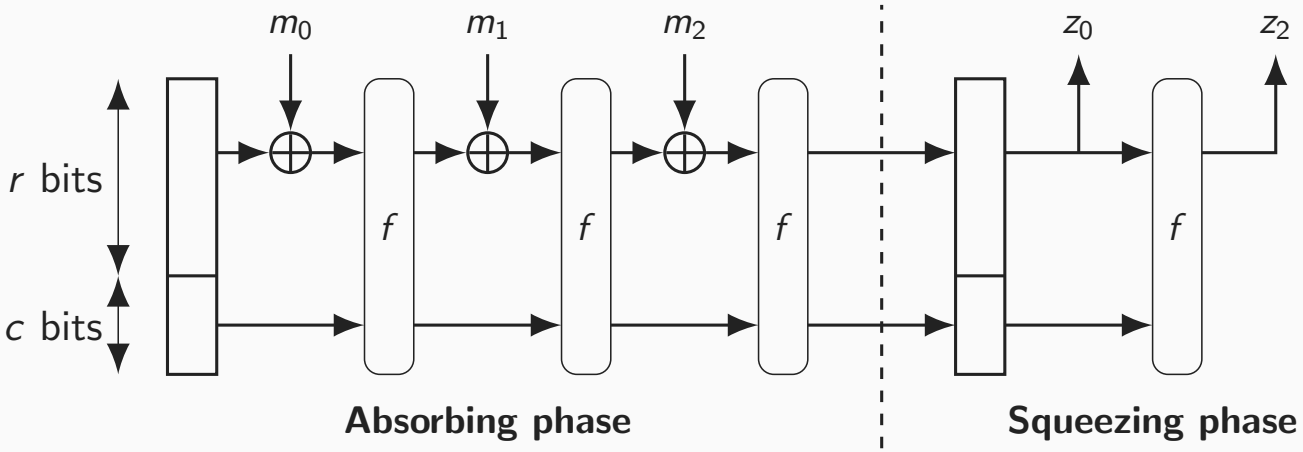
Even-Mansour construction

What is a Permutation?

Definition: A Permutation is a keyless block cipher.



Even-Mansour construction



Sponge construction

Why Gimli?

Currently we have:

Permutation	width in bits	Benefits
AES	128	very fast *if the instruction is available*.
Chaskey	128	lightning fast *on Cortex-M0/M3/M4*
Keccak-*f*	200,400,800,1600	low-cost masking
Salsa20,ChaCha20	512	very fast *on CPUs with vector units*.

Why Gimli?

Currently we have:

Permutation	Hindrance
AES	**Not that fast without HW**.
Chaskey	**Low security margin, slow with side-channel protection**
Keccak-f	**Huge state (800,1600)**
Salsa20,ChaCha20	**Horrible on HW**.

Why Gimli?

Currently we have:

Permutation	Hindrance
AES	**Not that fast without HW.**
Chaskey	**Low security margin, slow with side-channel protection**
Keccak-f	**Huge state (800,1600)**
Salsa20,ChaCha20	**Horrible on HW.**

Can we have a permutation that is not too big, nor too small and good in all these areas?

Yes!

Source: *Wikipedia*, *Fair Use*

What is Gimli?

GIMLI is:

- a 384-bit permutation (just the right size)
  - Sponge with $c = 256, r = 128 \implies$ 128 bits of security
  - Cortex-M3/M4: full state in registers
  - AVR, Cortex-M0: 192 bits (half state) fit in registers

What is Gimli?

Gimli is:

- a 384-bit permutation (just the right size)
  - Sponge with $c = 256, r = 128 \implies$ 128 bits of security
  - Cortex-M3/M4: full state in registers
  - AVR, Cortex-M0: 192 bits (half state) fit in registers
- with high cross-platform performances
- designed for:
  - energy-efficient hardware
  - side-channel-protected hardware
  - microcontrollers
  - compactness
  - vectorization
  - short messages
  - high security level

Specifications: State

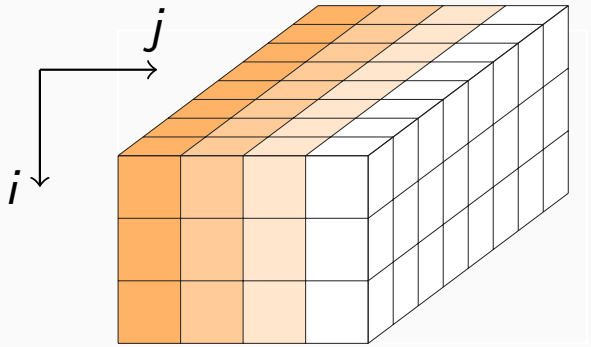
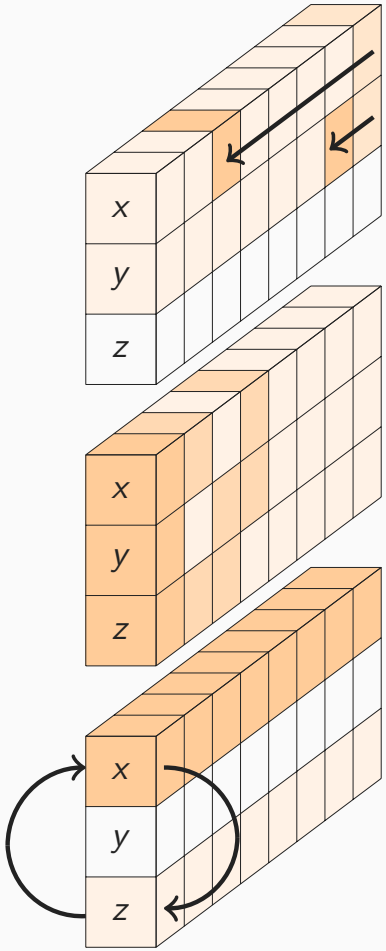


Figure: State Representation

384 bits represented as:

- a parallelepiped with dimensions $3 \times 4 \times 32$ (Keccak-like)
- or, as a 3×4 matrix of 32-bit words.

Specifications: Non-linear layer



In parallel:

$x \leftarrow x \lll 24$

$y \leftarrow y \lll 9$

In parallel:

$x \leftarrow x \oplus (z \ll 1) \oplus ((y \wedge z) \ll 2)$

$y \leftarrow y \oplus \quad x \quad \oplus ((x \vee z) \ll 1)$

$z \leftarrow z \oplus \quad y \quad \oplus ((x \wedge y) \ll 3)$

In parallel:

$x \leftarrow z$

$z \leftarrow x$

Figure: The bit-sliced 9-to-3-bit SP-box applied to a column

Specifications: Linear layer

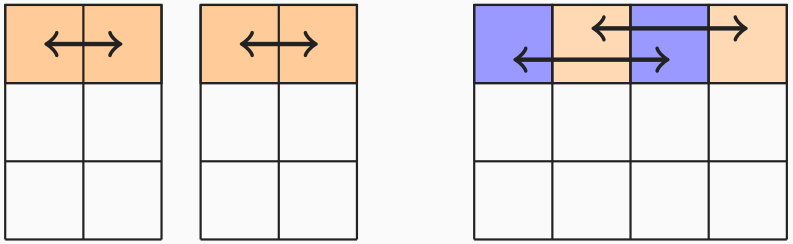


Figure: The linear layer

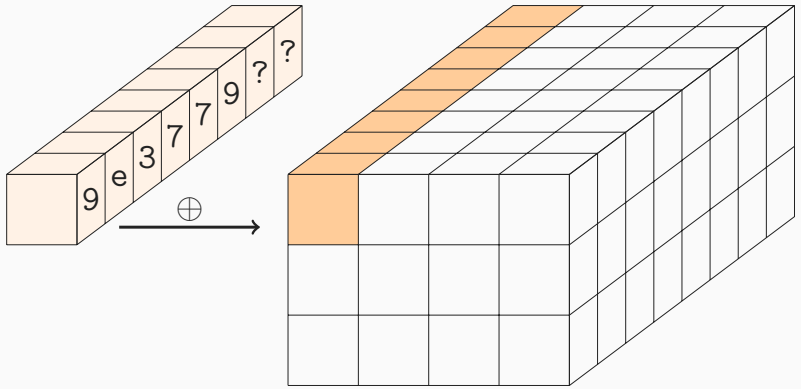
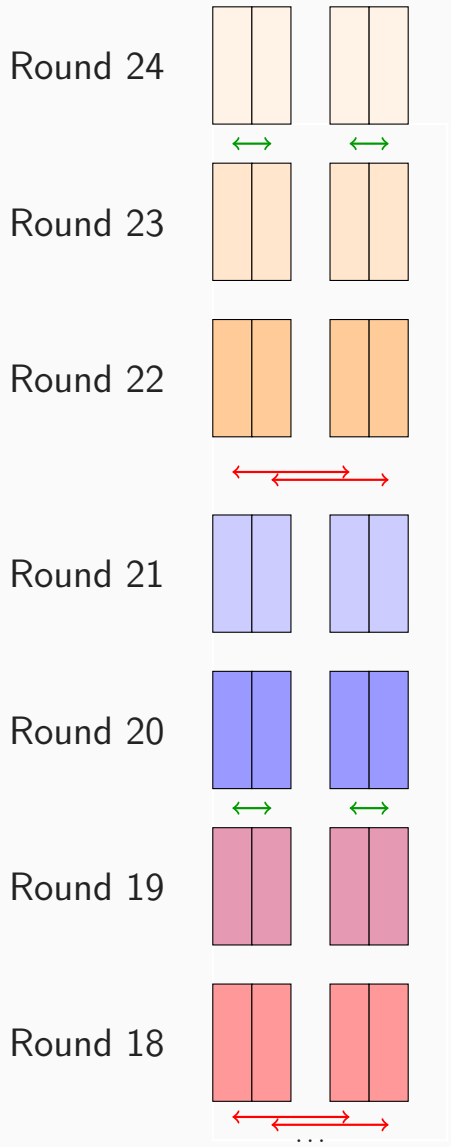


Figure: Constant addition `0x9e3779??`

Gimli in C

```
extern void Gimli(uint32_t *state) {

  uint32_t round, column, x, y, z;

  for (round = 24; round > 0; --round) {

    for (column = 0; column < 4; ++column) {
      x = rotate(state[    column], 24);                 // x <<< 24
      y = rotate(state[4 + column],  9);                 // y <<< 9
      z =        state[8 + column];

      state[8 + column] = x ^ (z << 1) ^ ((y & z) << 2);
      state[4 + column] = y ^ x        ^ ((x | z) << 1);
      state[column]     = z ^ y        ^ ((x & y) << 3);
    }

    if ((round & 3) == 0) { // small swap: pattern s...s...s... etc.
      x = state[0]; state[0] = state[1]; state[1] = x;
      x = state[2]; state[2] = state[3]; state[3] = x;
    }
    if ((round & 3) == 2) { // big swap: pattern ..S...S...S. etc.
      x = state[0]; state[0] = state[2]; state[2] = x;
      x = state[1]; state[1] = state[3]; state[3] = x;
    }

    if ((round & 3) == 0) { // add constant: pattern c...c...c... etc.
      state[0] ^= (0x9e377900 | round);
    }
  }
}
```

Specifications: Rounds



Non-linear layer

Small Swap & Round constant addition

Non-linear layer

Non-linear layer

Big Swap

Non-linear layer

Non-linear layer

Small Swap & Round constant addition

Non-linear layer

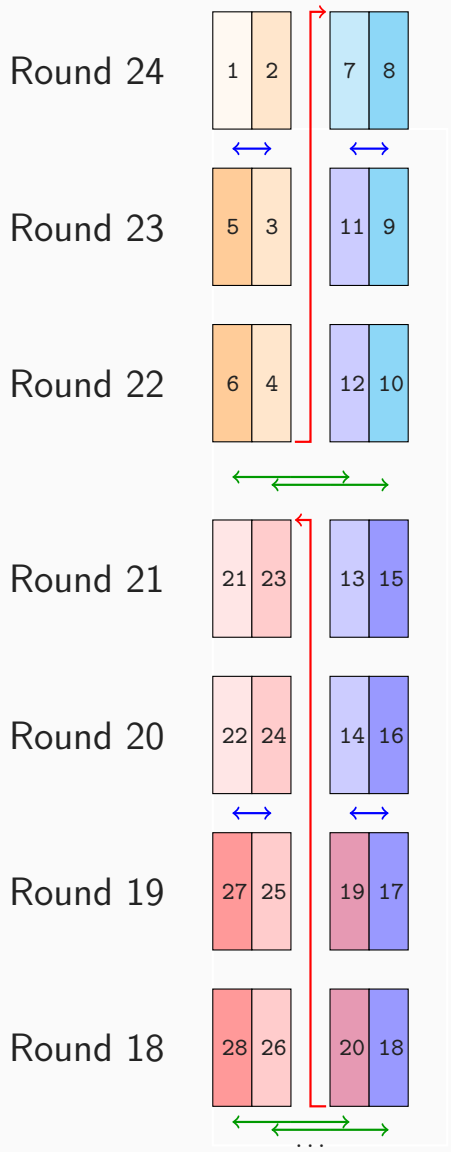
Non-linear layer

Big Swap

...

Figure: 7 first rounds of GIMLI

Unrolled AVR & Cortex-M0




```
1.   SP-box col.  0
2.   SP-box col.  1
swap word s0,0 and s0,1
3.   SP-box col.  1
4.   SP-box col.  1
5.   SP-box col.  0
6.   SP-box col.  0
store columns 0,1 ; load columns 2,3
7.   SP-box col.  2
8.   SP-box col.  3
swap word s0,2 and s0,3
9.   SP-box col.  3
10.  SP-box col.  3
11.  SP-box col.  2
12.  SP-box col.  2
push word s0,2, s0,3 ; load word s0,0, s0,1
13.  SP-box col.  2
14.  SP-box col.  2
15.  SP-box col.  3
16.  SP-box col.  3
swap word s0,2 and s0,3
17.  SP-box col.  3
18.  SP-box col.  3
19.  SP-box col.  2
20.  SP-box col.  2
store columns 2,3 ; load columns 0,1
...
```

Figure: Computation order on AVR & Cortex-M0

Implementation in Assembly

```
# Rotate
x ← x ⋘ 24
y ← y ⋘ 9
u ← x
.
.
```

```
# Compute x
v ← z ≪ 1
x ← z ∧ y
x ← x ≪ 2
x ← u ⊕ x
x ← x ⊕ v
```

```
# Compute y
v ← y
y ← u ∨ z
y ← y ≪ 1
y ← u ⊕ y
y ← y ⊕ v
```

```
# Compute z
u ← u ∧ v
u ← u ≪ 3
z ← z ⊕ v
z ← z ⊕ u
.
```

The SP-box requires only 2 additional registers u and v.

Rotate for free on Cortex-M3/M4

```
# Rotate
x ← x ⋘ 24

u ← x
```

```
# Compute x
v ← z ≪ 1
x ← z ∧ (y ⋘ 9)
x ← x ≪ 2
x ← u ⊕ x
x ← x ⊕ v
```

```
# Compute y
v ← y
y ← u ∨ z
y ← y ≪ 1
y ← u ⊕ y
y ← y ⊕ (v ⋘ 9)
```

```
# Compute z
u ← u ∧ (v ⋘ 9)
u ← u ≪ 3
z ← z ⊕ (v ⋘ 9)
z ← z ⊕ u
```

Remove y <<< 9.

Shift for free on Cortex-M3/M4

```
# Rotate
x ← x ⋘ 24

u ← x
```

```
# Compute x

x ← z ∧ (y ⋘ 9)

x ← u ⊕ (x ≪ 2)
x ← x ⊕ (z ≪ 1)
```

```
# Compute y
v ← y
y ← u ∨ z

y ← u ⊕ (y ≪ 1)
y ← y ⊕ (v ⋘ 9)
```

```
# Compute z
u ← u ∧ (v ⋘ 9)

z ← z ⊕ (v ⋘ 9)
z ← z ⊕ (u ≪ 3)
```

Get rid of the other shifts.

Free mov on Cortex-M3/M4

```
# Rotate
x ← x ⋘ 24
```

```
# Compute x

u ← z ∧ (y ⋘ 9)

u ← x ⊕ (u ≪ 2)
u ← u ⊕ (z ≪ 1)
```

```
# Compute y
v ← y
y ← x ∨ z

y ← x ⊕ (y ≪ 1)
y ← y ⊕ (v ⋘ 9)
```

```
# Compute z
x ← x ∧ (v ⋘ 9)

z ← z ⊕ (v ⋘ 9)
z ← z ⊕ (x ≪ 3)
```

Remove the last mov:

u contains the new value of x

y contains the new value of y

z contains the new value of z

Free mov on Cortex-M3/M4

```
# Rotate
x ← x ⋘ 24
```

```
# Compute x

u ← z ∧ (y ⋘ 9)

u ← x ⊕ (u ≪ 2)
u ← u ⊕ (z ≪ 1)
```

```
# Compute y

v ← x ∨ z

v ← x ⊕ (v ≪ 1)
v ← v ⊕ (y ⋘ 9)
```

```
# Compute z
x ← x ∧ (y ⋘ 9)

z ← z ⊕ (y ⋘ 9)
z ← z ⊕ (x ≪ 3)
```

Remove the last mov:

u contains the new value of x

v contains the new value of y

z contains the new value of z

Free swap on Cortex-M3/M4

```
# Rotate
x ← x ⋘ 24
```

```
# Compute x
u ← z ∧ (y ⋘ 9)
u ← x ⊕ (u ≪ 2)
u ← u ⊕ (z ≪ 1)
```

```
# Compute y
v ← x ∨ z
v ← x ⊕ (v ≪ 1)
v ← v ⊕ (y ⋘ 9)
```

```
# Compute z
x ← x ∧ (y ⋘ 9)
z ← z ⊕ (y ⋘ 9)
z ← z ⊕ (x ≪ 3)
```

Swap x and z:

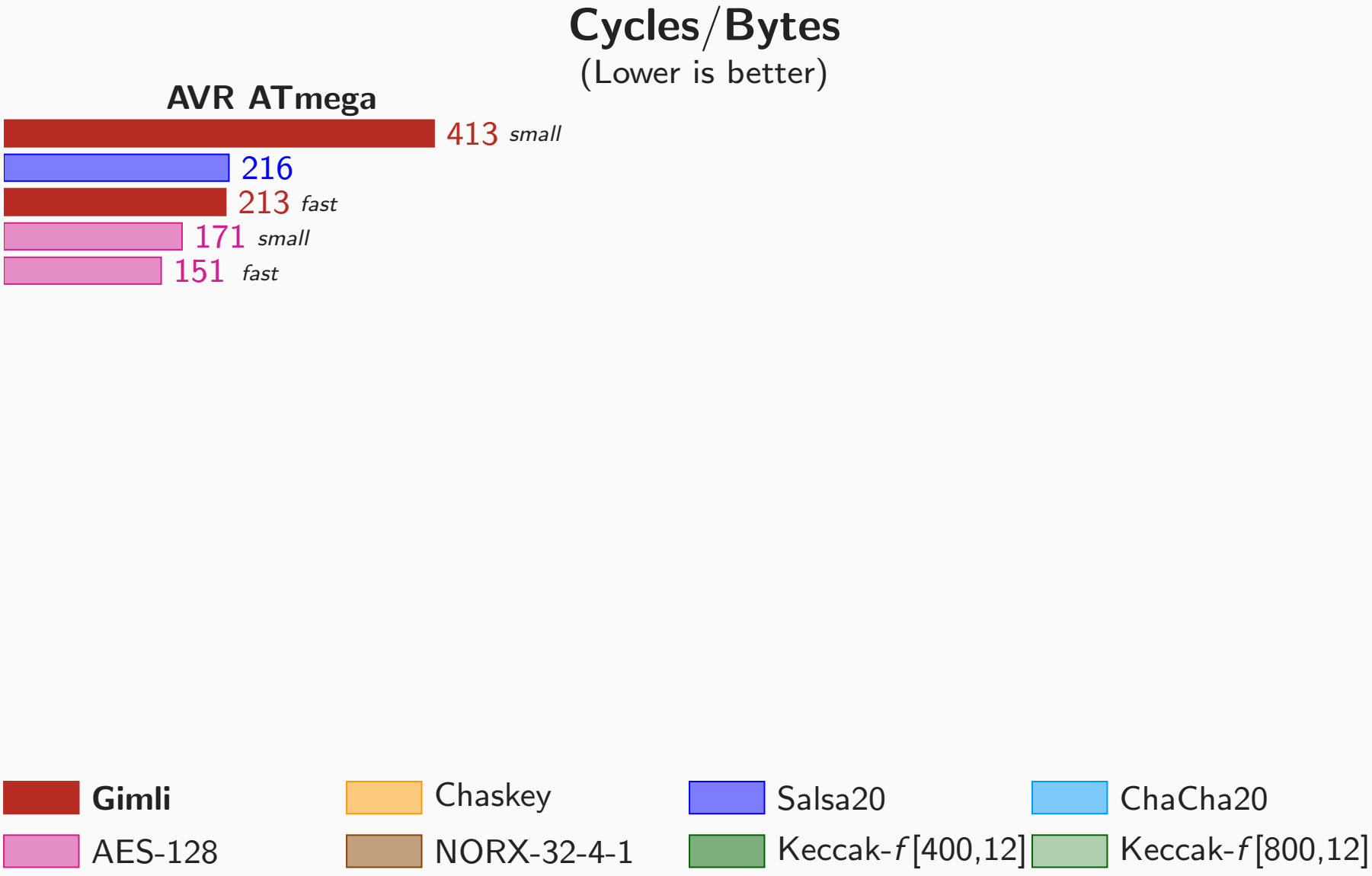
u contains the new value of z

v contains the new value of y

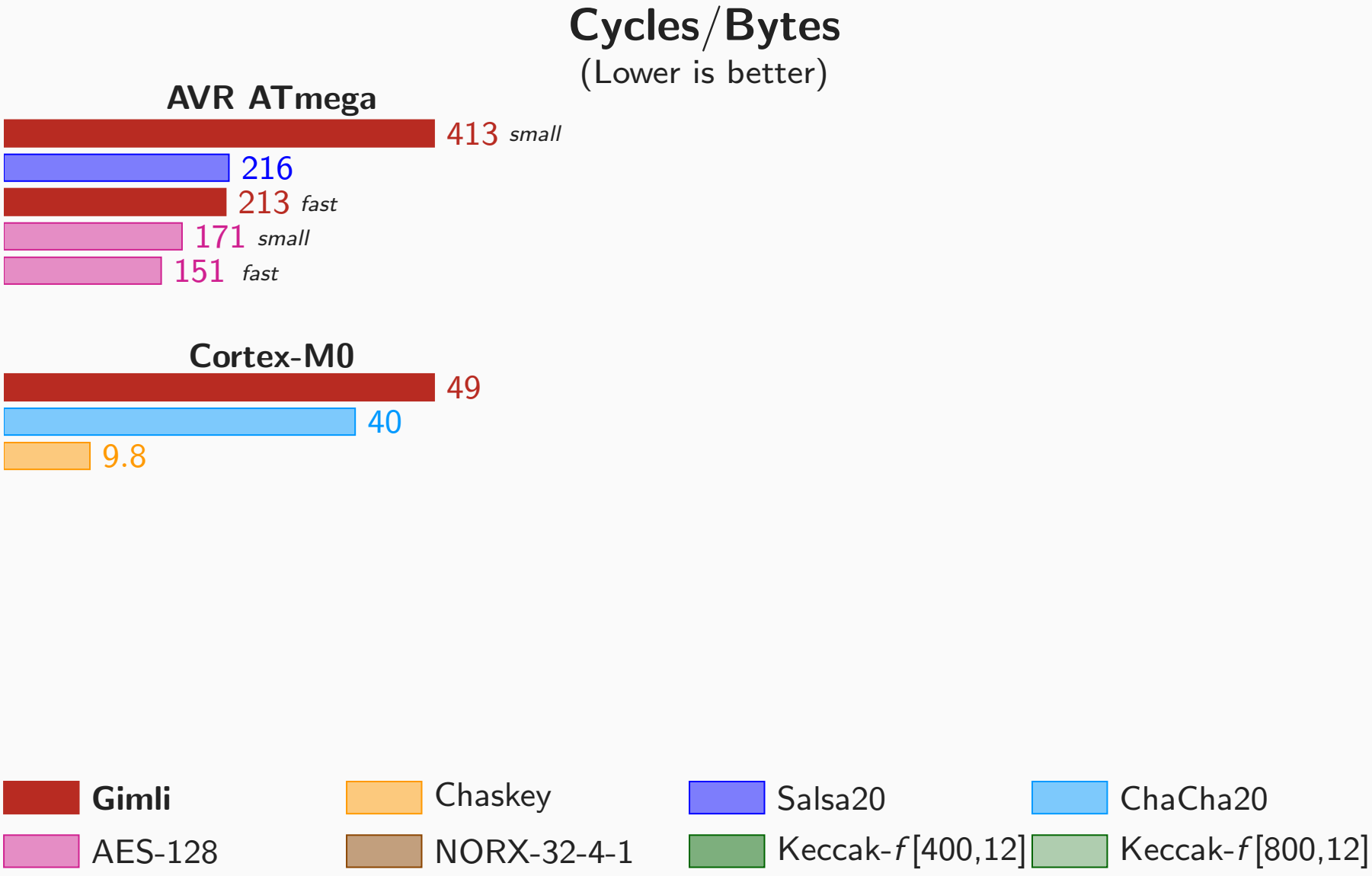
z contains the new value of x

SP-box requires a total of 10 instructions.

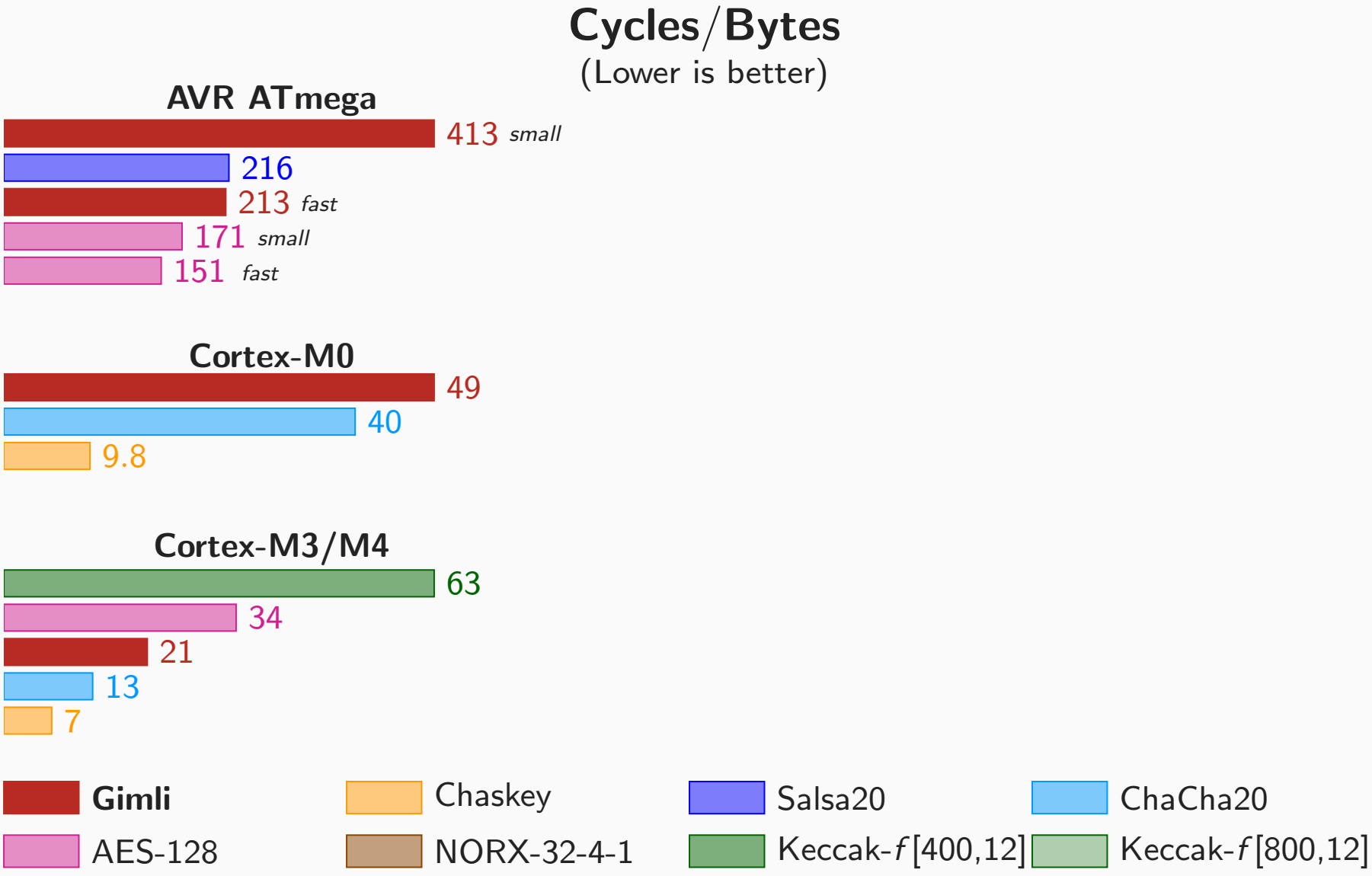
How fast is Gimli? (Software)



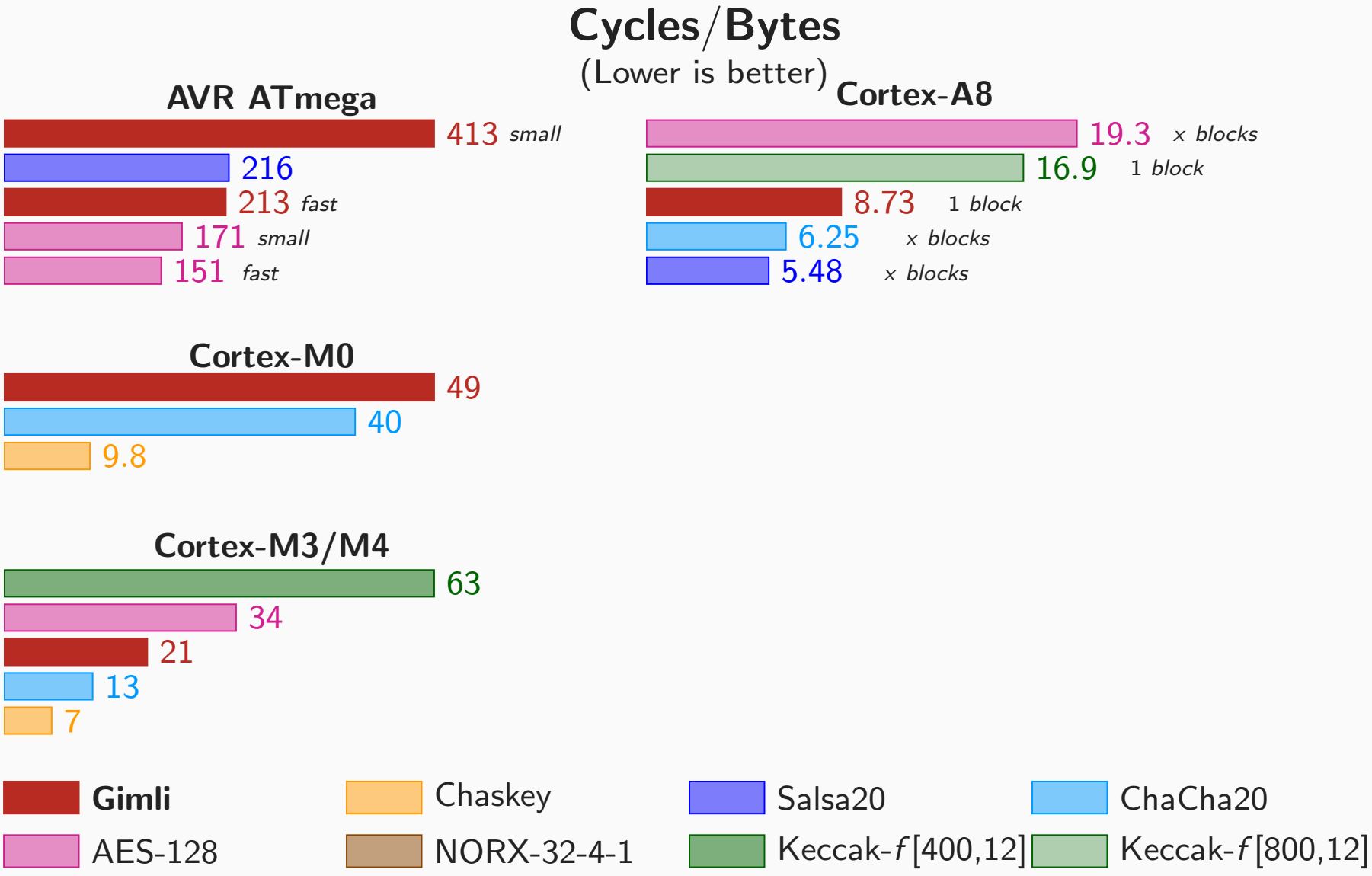
How fast is Gimli? (Software)



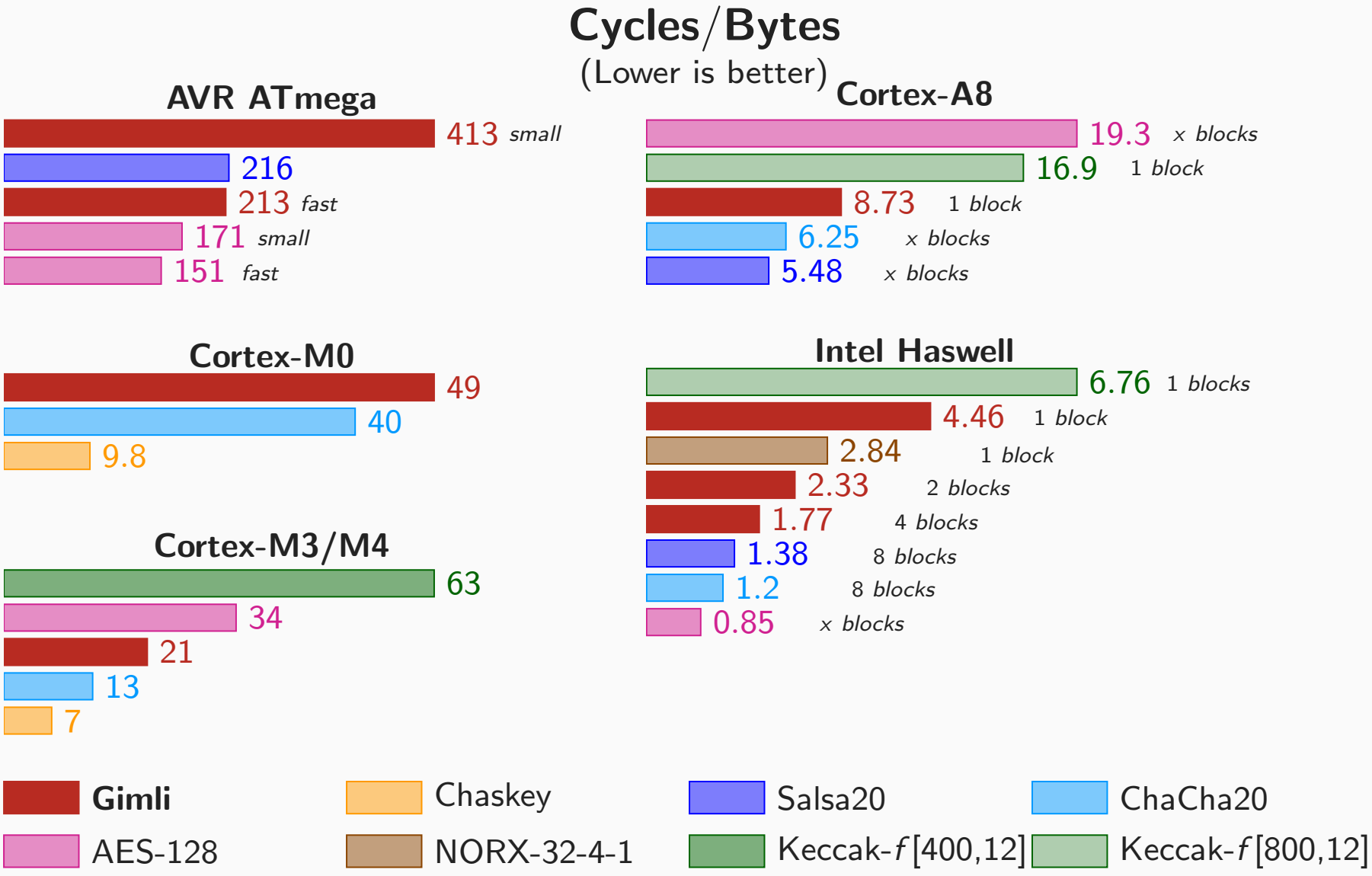
How fast is Gimli? (Software)



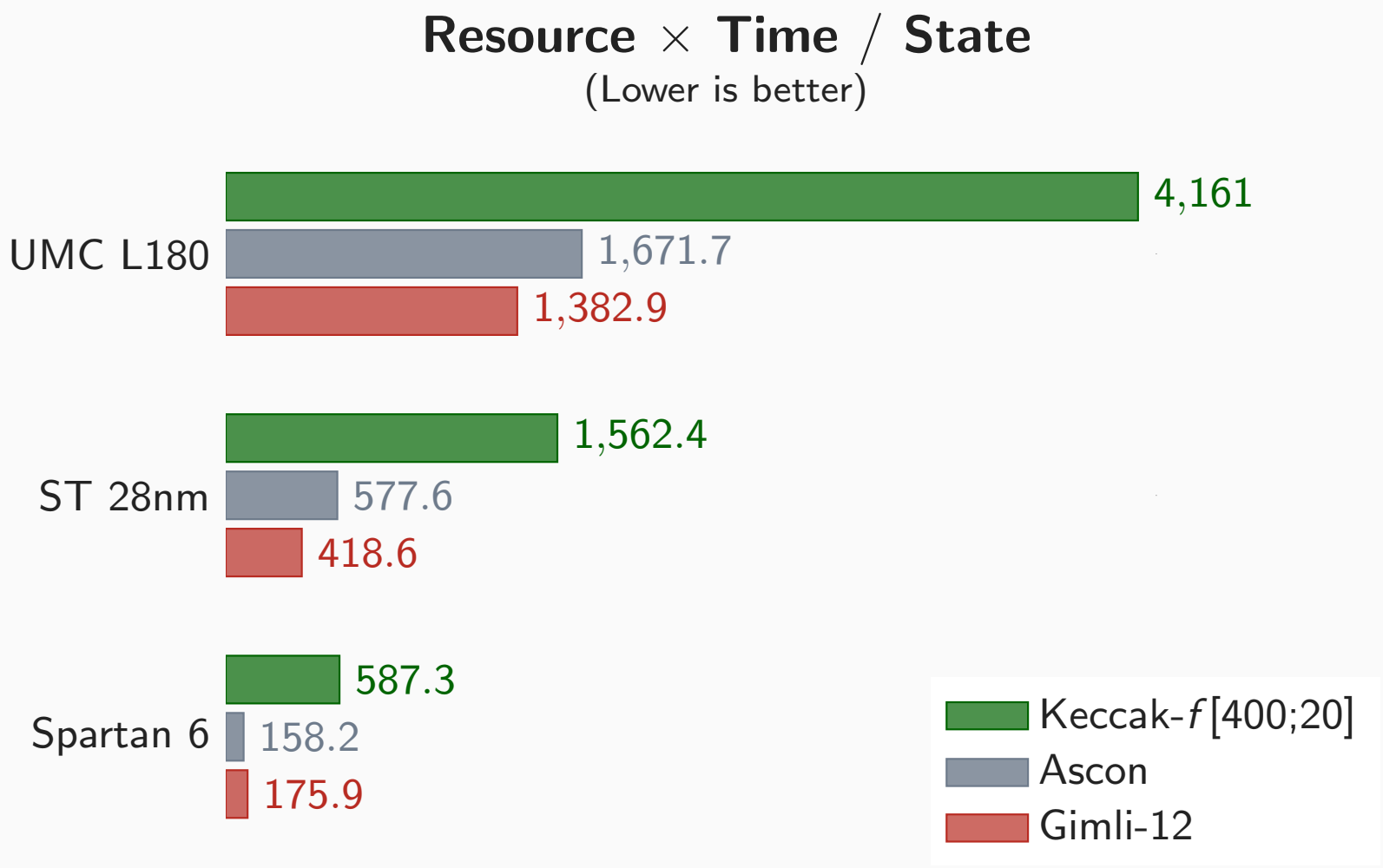
How fast is Gimli? (Software)



How fast is Gimli? (Software)



How efficient is Gimli? (Hardware)



latency: 2 cycles

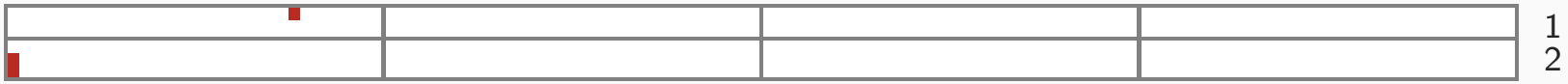
How secure is Gimli?

- Simple diffusion
  - avalanche effect shown after 10 rounds.
  - each bit influences the full state after 8 rounds.

Worst-case propagation in Gimli over 8 rounds.

How secure is Gimli?

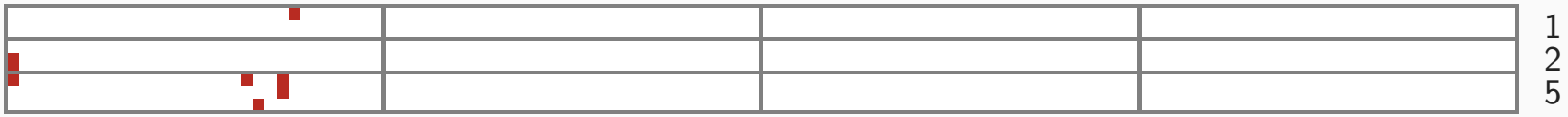
- Simple diffusion
  - avalanche effect shown after 10 rounds.
  - each bit influences the full state after 8 rounds.



Worst-case propagation in Gimli over 8 rounds.

How secure is Gimli?

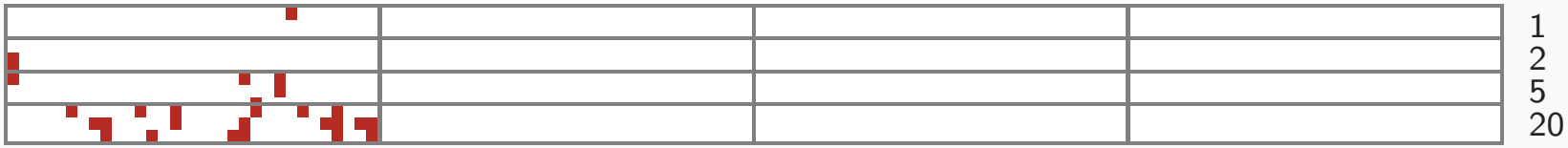
- Simple diffusion
  - avalanche effect shown after 10 rounds.
  - each bit influences the full state after 8 rounds.



Worst-case propagation in Gimli over 8 rounds.

How secure is Gimli?

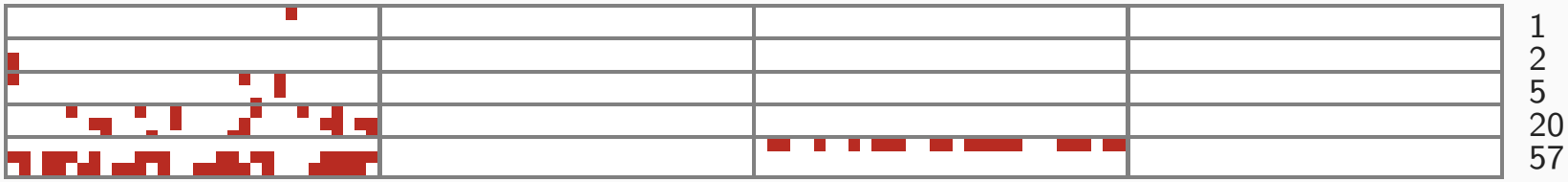
- Simple diffusion
  - avalanche effect shown after 10 rounds.
  - each bit influences the full state after 8 rounds.



Worst-case propagation in Gimli over 8 rounds.

How secure is Gimli?

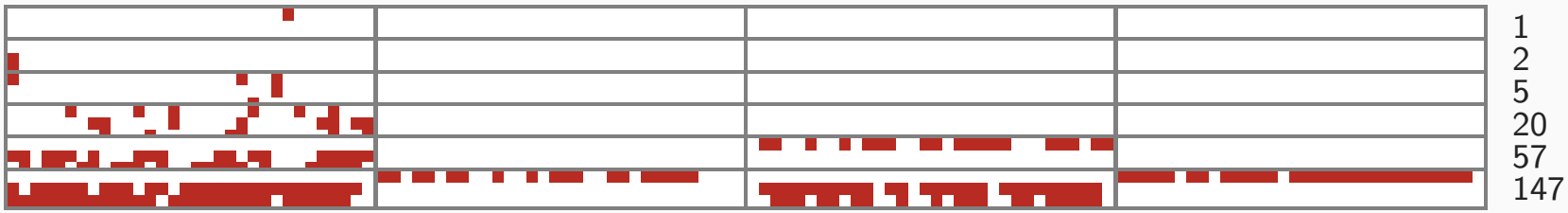
- Simple diffusion
  - avalanche effect shown after 10 rounds.
  - each bit influences the full state after 8 rounds.



Worst-case propagation in Gimli over 8 rounds.

How secure is Gimli?

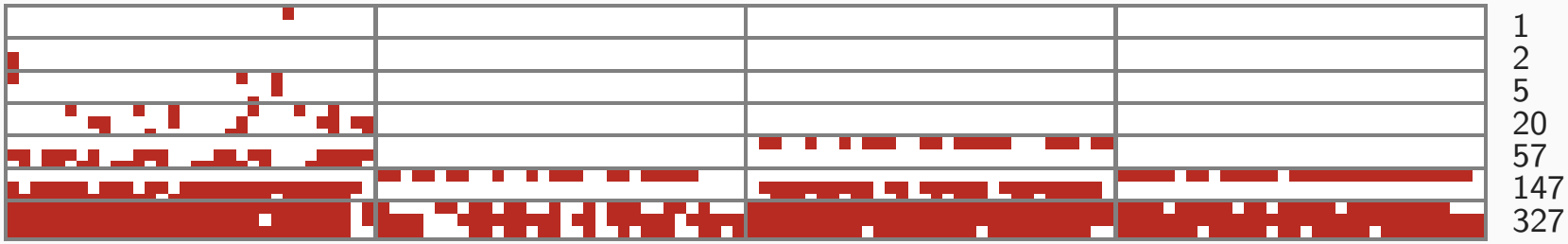
- Simple diffusion
  - avalanche effect shown after 10 rounds.
  - each bit influences the full state after 8 rounds.



Worst-case propagation in Gimli over 8 rounds.

How secure is Gimli?

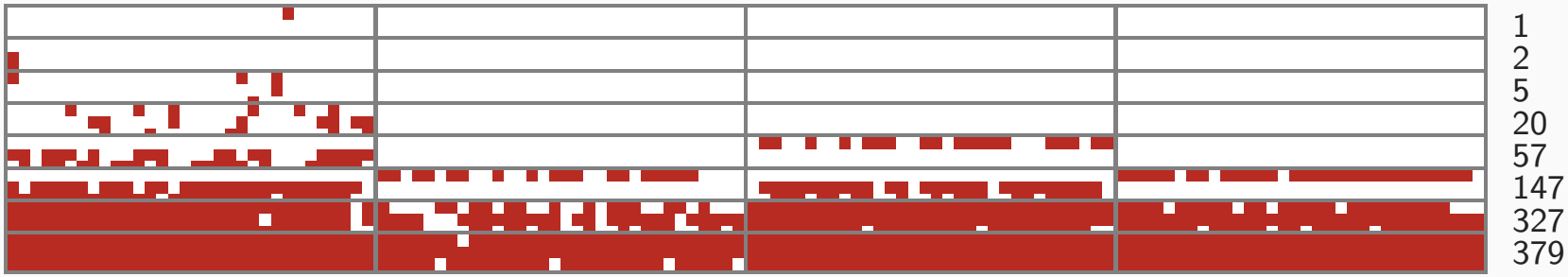
- Simple diffusion
  - avalanche effect shown after 10 rounds.
  - each bit influences the full state after 8 rounds.



Worst-case propagation in Gimli over 8 rounds.

How secure is Gimli?

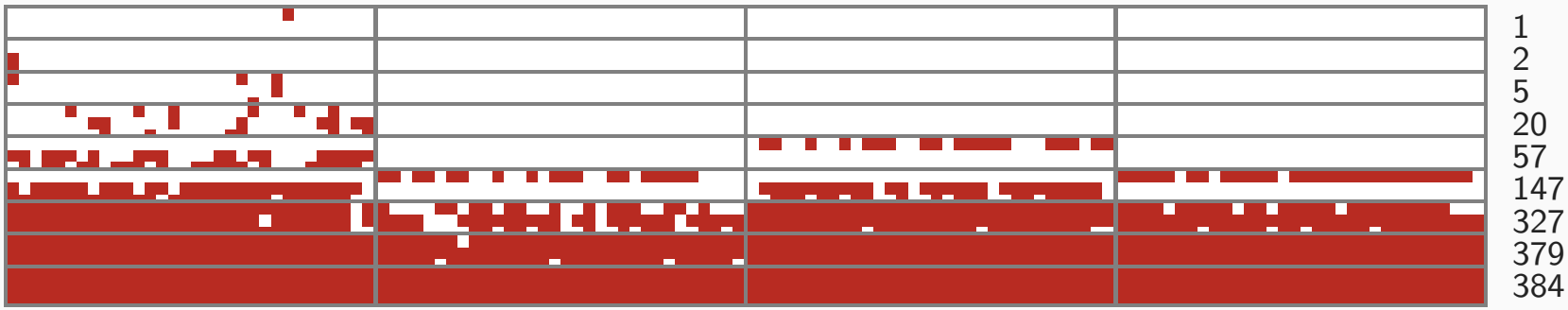
- Simple diffusion
  - avalanche effect shown after 10 rounds.
  - each bit influences the full state after 8 rounds.



Worst-case propagation in Gimli over 8 rounds.

How secure is Gimli?

- Simple diffusion
  - avalanche effect shown after 10 rounds.
  - each bit influences the full state after 8 rounds.



Worst-case propagation in Gimli over 8 rounds.

How secure is Gimli?

Round	col_0	col_1	col_2	col_3	Weight
0	0x80404180 0x80002080 0x80002080	0x00020100 - 0x80010080	- - -	- - -	18
1	0x80800100 0x80400000 0x80400080	- - -	- - -	- - -	8
2	0x80000000 0x80000000 0x80000000	- - -	- - -	- - -	0
3	- - 0x80000000	- - -	- - -	- - -	0
4	0x00800000 - -	- - -	- - -	- - -	2
5	- 0x00000001 0x00800000	- - -	- - -	- - -	4
6	0x01008000 0x00000200 0x01000000	- - -	- - -	- - -	6
7	- 0x01040002 0x03008000	- - -	- - -	- - -	14
8	0x02020480 0x0a00040e 0x06010000	- - -	- 0x06000c00 0x00010002	- - -	-

Optimal differential trail for 8-round probability 2^{-52}

How secure is Gimli?

- Differential propagation
  - Optimal 8-round trail with probability of 2^{-52}
- Algebraic Degree and Integral distinguishers
  - z_0 has an algebraic degree of 367 after 11 rounds (upper bound)
  - 11-round integral distinguisher with 96 active bits.
  - 13-round integral distinguisher with 192 active bits.

Mike Attacks!

- August 1^{st}, `eprint.iacr.org/2017/743`
- Claim against 192-bit key.
- Requires:
  - "$2^{138.5}$ work".
  - "2^{129} bits of memory".

i.e. more hardware and more time than naive brute-force attack.
(2^{80} parallel units, each searching 2^{112} keys.)

Image: *Wikipedia*, *Fair Use*

Mike Attacks!

- August 1^{st}, `eprint.iacr.org/2017/743`
- Claim against 192-bit key.
- Requires:
  - "$2^{138.5}$ work".
  - "2^{129} bits of memory".

  i.e. more hardware and more time than naive brute-force attack. (2^{80} parallel units, each searching 2^{112} keys.)
- "golden collision" techniques by van Oorschot–Wiener (1996) reduce the cost in memory but increase the work. Still worse than brute-force.

Image: *Wikipedia*, *Fair Use*

Mike Attacks!

- August 1^{st}, `eprint.iacr.org/2017/743`
- Claim against 192-bit key.
- Requires:
  - "$2^{138.5}$ work".
  - "2^{129} bits of memory".

  i.e. more hardware and more time than naive brute-force attack.
  (2^{80} parallel units, each searching 2^{112} keys.)
- "golden collision" techniques by van Oorschot–Wiener (1996) reduce the cost in memory but increase the work. Still worse than brute-force.
- Standard practice in designing PRF such as ChaCha20 add words to positions that **maximize** diffusion.
  Hamburg's attack requires to add key words to positions selected to *minimize* diffusion.
- Practical attack not feasible in the foreseeable future, even with quantum computers.

Image: *Wikipedia*, *Fair Use*

TweetGimli @TweetGimli

```
#include<stdint.h>
#define R(V)x=S[V],S[V]=S[V^y],S[V^y]=x,
void gimli(uint32_t*S){for(uint32_t r=24,x,y,z,*T;r--;y=72>>r%4*2&3,R(0)R(3)
```

TweetGimli @TweetGimli

```
*S^=y&1?0x9e377901+r:0)for(T=S+4;T-->S;*T=z^y^8*(x&y),T[4]=y^x^2*(x|z),T[8]=x^2*z^4*(y&
z))x=*T<<24|*T>>8,y=T[4]<<9|T[4]>>23,z=T[8];}
```

authorcontact-Gimli@box.cr.yp.to
https://gimli.cr.yp.to

Special Thanks to *Lorenz Panny*, *Peter Taylor* and *Orson Peters* for the *Code Golfing*.