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**

CHES, Taipei, September 27, 2017

What is a Permutation?

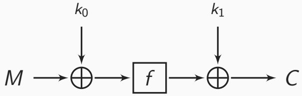
Definition: A Permutation is a keyless block cipher.

Definition: A Permutation is a keyless block cipher.

“Wait, what?”

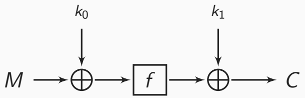
What is a Permutation?

Definition: A Permutation is a keyless block cipher.

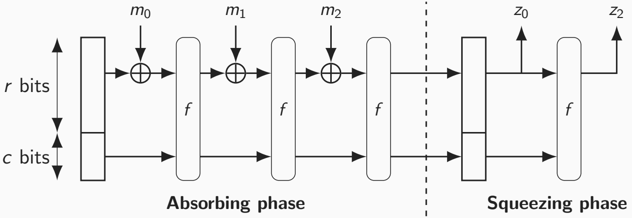


Even-Mansour construction

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**.

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

What is Gimli?

GIMLI is:

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

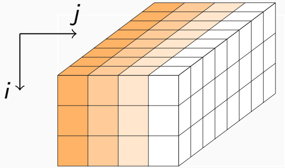
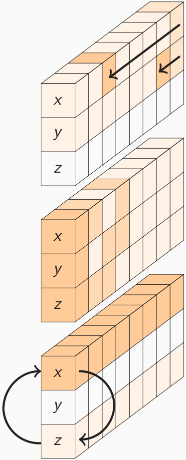


Figure: State Representation

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-bits SP-box applied to a column

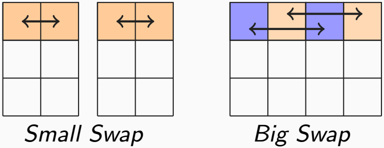


Figure: The linear layer

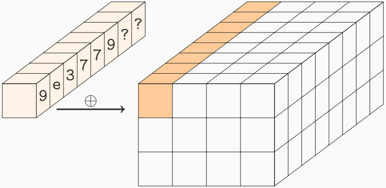


Figure: Constant addition `0x9e3779??`

Gimli in C

```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

Round 24 — Non-linear layer

Small Swap & Round constant addition

Round 23 — Non-linear layer

Round 22 — Non-linear layer

Big Swap

Round 21 — Non-linear layer

Round 20 — Non-linear layer

Small Swap & Round constant addition

Round 19 — Non-linear layer

Round 18 — Non-linear layer

Big Swap

...

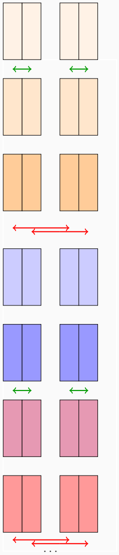
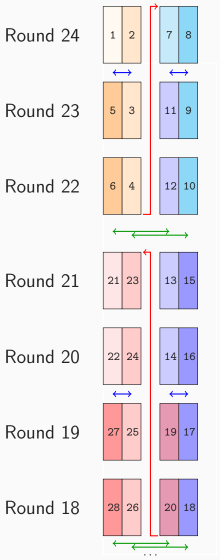


Figure: 7 first rounds of GIMLI



1. SP-box col. 0
2. SP-box col. 1

swap word $s_{0,0}$ and $s_{0,1}$

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 $s_{0,2}$ and $s_{0,3}$

9. SP-box col. 3
10. SP-box col. 3
11. SP-box col. 2
12. SP-box col. 2

push word $s_{0,2}$, $s_{0,3}$; load word $s_{0,0}$, $s_{0,1}$

13. SP-box col. 2
14. SP-box col. 2
15. SP-box col. 3
16. SP-box col. 3

swap word $s_{0,2}$ and $s_{0,3}$

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	# Compute x	# Compute y	# Compute z
$x \leftarrow x \lll 24$	$v \leftarrow z \ll 1$	$v \leftarrow y$	$u \leftarrow u \wedge v$
$y \leftarrow y \lll 9$	$x \leftarrow z \wedge y$	$y \leftarrow u \vee z$	$u \leftarrow u \ll 3$
$u \leftarrow x$	$x \leftarrow x \ll 2$	$y \leftarrow y \ll 1$	$z \leftarrow z \oplus v$
	$x \leftarrow u \oplus x$	$y \leftarrow u \oplus y$	$z \leftarrow z \oplus u$
	$x \leftarrow x \oplus v$	$y \leftarrow y \oplus v$	

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

# Rotate	# Compute x	# Compute y	# Compute z
x ← x ⋘ 24	v ← z ≪ 1	v ← y	u ← u ∧ (v ⋘ 9)
	x ← z ∧ (y ⋘ 9)	y ← u ∨ z	u ← u ≪ 3
u ← x	x ← x ≪ 2	y ← y ≪ 1	z ← z ⊕ (v ⋘ 9)
	x ← u ⊕ x	y ← u ⊕ y	z ← z ⊕ u
	x ← x ⊕ v	y ← y ⊕ (v ⋘ 9)	

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	# Compute x	# Compute y	# Compute z
x ← x ⋘ 24			x ← x ∧ (y ⋘ 9)
	u ← z ∧ (y ⋘ 9)	v ← x ∨ z	
			z ← z ⊕ (y ⋘ 9)
	u ← x ⊕ (u ≪ 2)	v ← x ⊕ (v ≪ 1)	z ← z ⊕ (x ≪ 3)
	u ← u ⊕ (z ≪ 1)	v ← v ⊕ (y ⋘ 9)	

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	# Compute x	# Compute y	# Compute z
$x \leftarrow x \lll 24$	$u \leftarrow z \wedge (y \lll 9)$	$v \leftarrow x \vee z$	$x \leftarrow x \wedge (y \lll 9)$
	$u \leftarrow x \oplus (u \ll 2)$	$v \leftarrow x \oplus (v \ll 1)$	$z \leftarrow z \oplus (y \lll 9)$
	$u \leftarrow u \oplus (z \ll 1)$	$v \leftarrow v \oplus (y \lll 9)$	$z \leftarrow z \oplus (x \ll 3)$

Swap x and z:

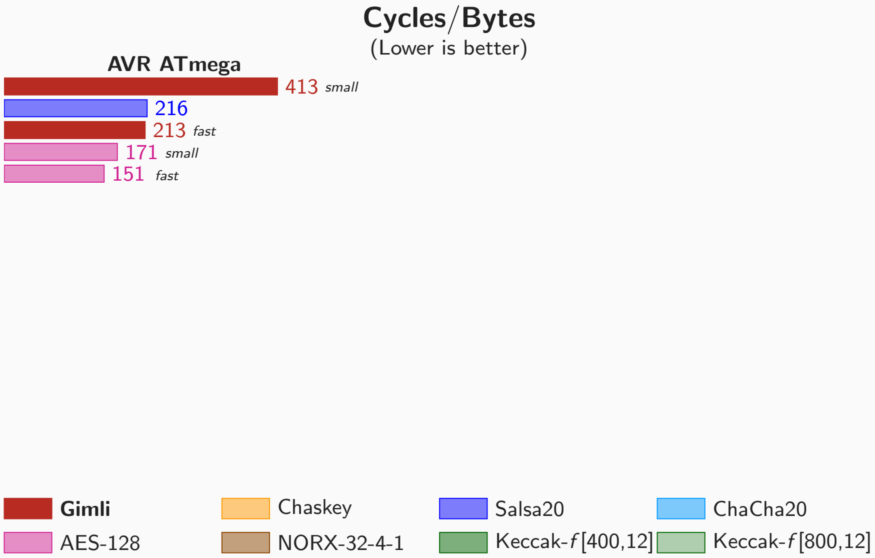
u contains the new value of z

v contains the new value of y

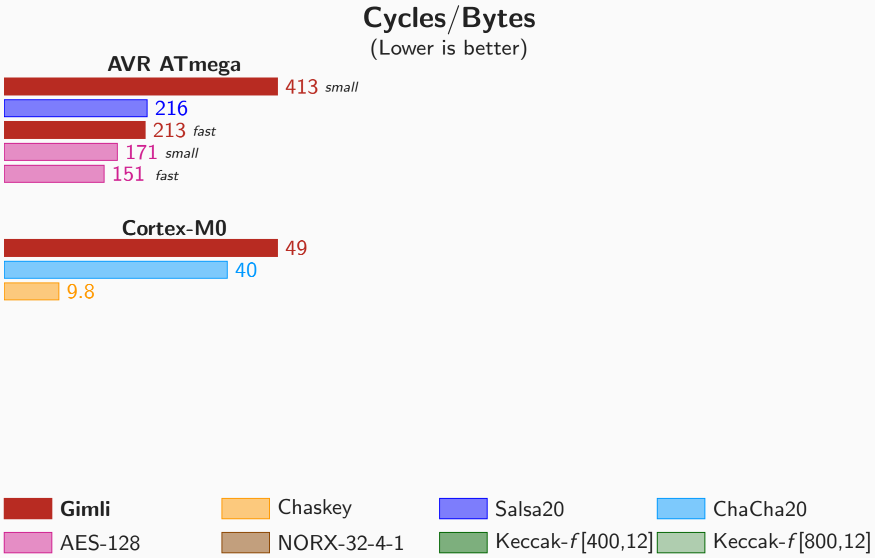
z contains the new value of x

SP-box requires a total of 10 instructions.

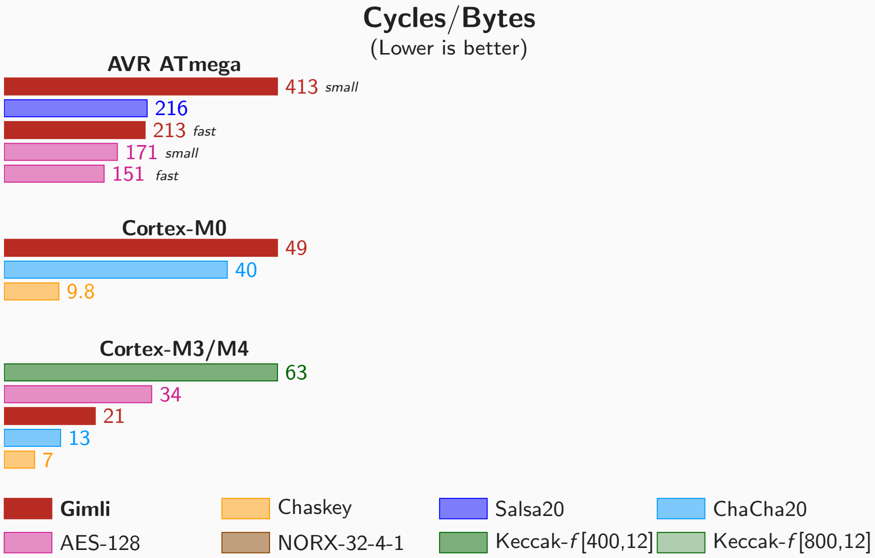
How fast is Gimli? (Software)



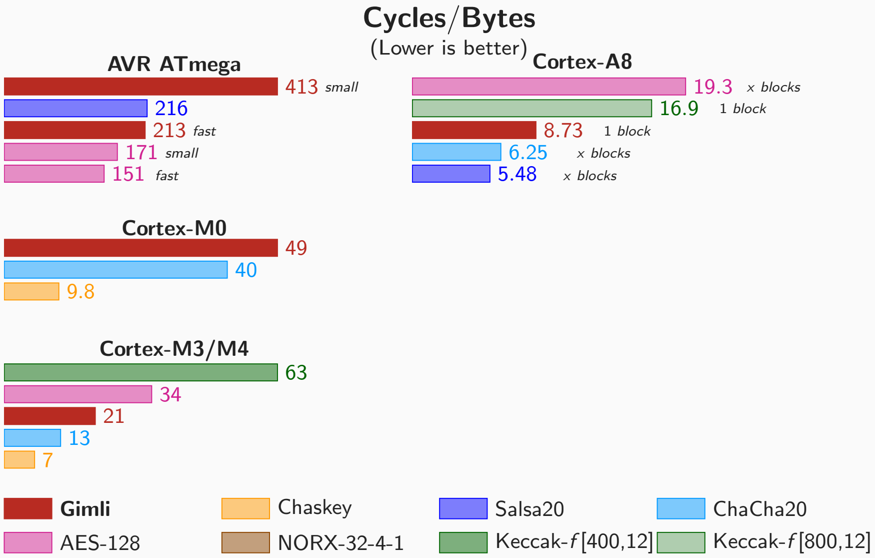
How fast is Gimli? (Software)



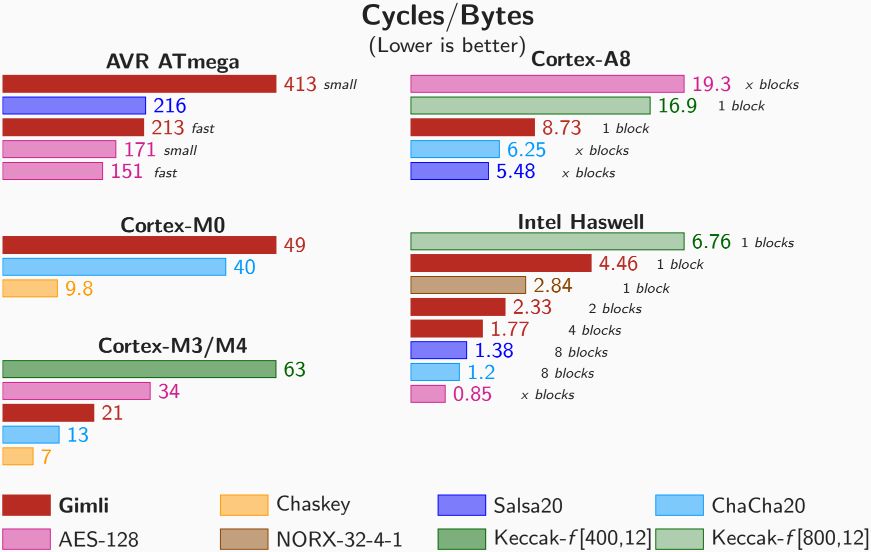
How fast is Gimli? (Software)



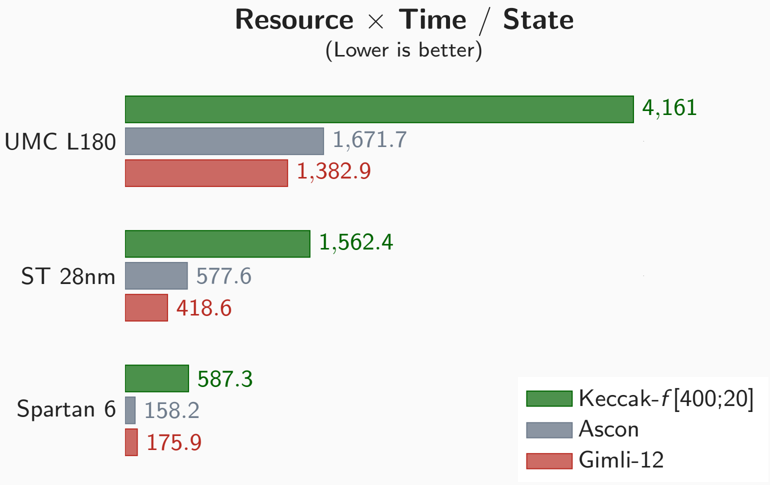
How fast is Gimli? (Software)



How fast is Gimli? (Software)



How efficient is Gimli? (Hardware)



latency : 2 cycles

How secure is Gimli?

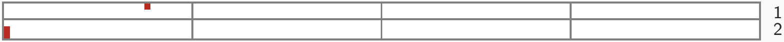
- Simple diffusion
  - avalanche effect shown after 10 rounds.
  - each bit influences the full state after 8 rounds.


1

Worse case propagation in Gimli over 8 rounds.

How secure is Gimli?

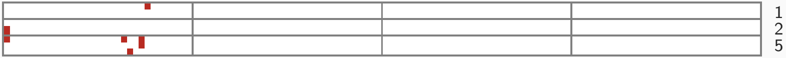
- Simple diffusion
  - avalanche effect shown after 10 rounds.
  - each bit influences the full state after 8 rounds.



Worse case propagation in Gimli over 8 rounds.

How secure is Gimli?

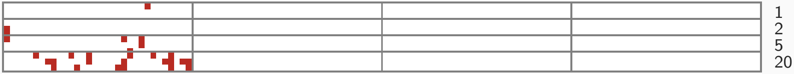
- Simple diffusion
  - avalanche effect shown after 10 rounds.
  - each bit influences the full state after 8 rounds.



Worse case propagation in Gimli over 8 rounds.

How secure is Gimli?

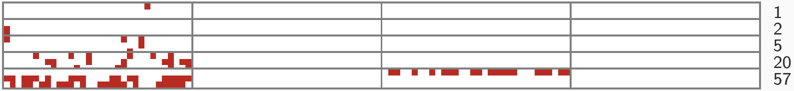
- Simple diffusion
  - avalanche effect shown after 10 rounds.
  - each bit influences the full state after 8 rounds.



Worse case propagation in Gimli over 8 rounds.

How secure is Gimli?

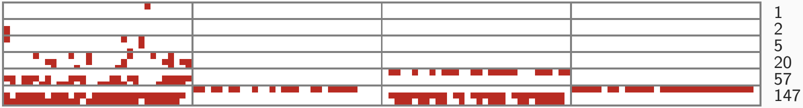
- Simple diffusion
  - avalanche effect shown after 10 rounds.
  - each bit influences the full state after 8 rounds.



Worse case propagation in Gimli over 8 rounds.

How secure is Gimli?

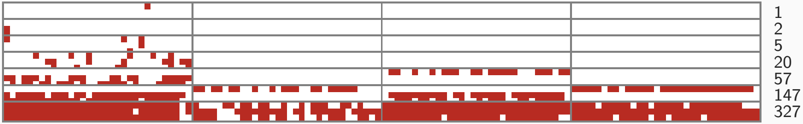
- Simple diffusion
  - avalanche effect shown after 10 rounds.
  - each bit influences the full state after 8 rounds.



Worse case propagation in Gimli over 8 rounds.

How secure is Gimli?

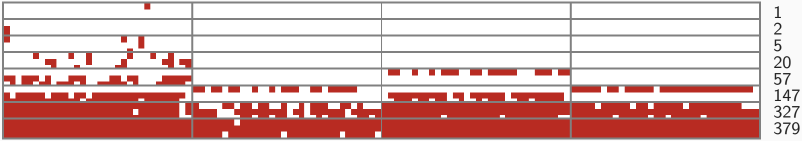
- Simple diffusion
  - avalanche effect shown after 10 rounds.
  - each bit influences the full state after 8 rounds.



Worse case propagation in Gimli over 8 rounds.

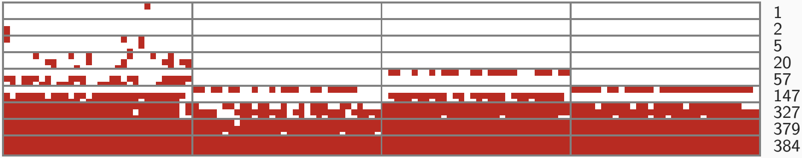
How secure is Gimli?

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



Worse case propagation in Gimli over 8 rounds.

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



Worse 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.

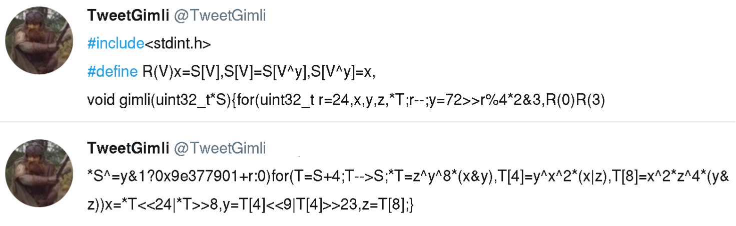
Image: *Wikipedia*, *Fair Use*

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*.

BACKUP SLIDES

How efficient is Gimli? (Hardware)

Permutation	Cycles	Resources	Period (ns)	Time (ns)	Res.×Time/state
FPGA – Xilinx Spartan 6 LX75					
Ascon	2	732 S(2700 L+325 F)	34.570	70	**158.2**
GIMLI 12*r*	2	1224 S(4398 L+389 F)	27.597	**56**	175.9
Keccak	2	1520 S(5555 L+405 F)	77.281	155	587.3
GIMLI 24*r*	1	2395 S(8769 L+385 F)	56.496	57	352.4
GIMLI 8*r*	3	831 S(2924 L+390 F)	24.531	74	159.3
GIMLI 6*r*	4	646 S(2398 L+390 F)	18.669	75	**125.6**
GIMLI 4*r*	6	415 S(1486 L+391 F)	8.565	**52**	**55.5**
GIMLI (Serial)	108	139 S(492 L+397 F)	3.996	432	156.2
28nm ASIC – ST 28nm FDSOI technology					
GIMLI 12*r*	2	35452 GE	2.2672	**5**	**418.6**
Ascon	2	32476 GE	2.8457	6	577.6
Keccak	2	55683 GE	5.6117	12	1562.4
GIMLI 24*r*	1	66205 GE	4.2870	5	739.1
GIMLI 8*r*	3	25224 GE	1.5921	**5**	313.7
GIMLI 4*r*	6	14999 GE	1.0549	7	**247.2**
GIMLI (Serial)	108	5843 GE	1.5352	166	2522.7
180nm ASIC – UMC L180					
GIMLI 12*r*	2	26685 GE	9.9500	**20**	**1382.9**
Ascon	2	23381 GE	11.4400	23	1671.7
Keccak	2	37102 GE	22.4300	45	4161.0
GIMLI 24*r*	1	53686 GE	17.4500	18	2439.6
GIMLI 8*r*	3	19393 GE	7.9100	24	**1198.4**
GIMLI 4*r*	6	11008 GE	10.1700	62	1749.1
GIMLI (Serial)	108	3846 GE	11.2300	1213	12146.0

Gates Equivalent(GE). Slice(S). LUT(L). Flip-Flop(F).

Bijectivity

$$f_0 = \begin{cases} x_0' \leftarrow x_0 \\ y_0' \leftarrow y_0 \oplus x_0 \\ z_0' \leftarrow z_0 \oplus y_0 \end{cases} \qquad f_0^{-1} = \begin{cases} x_0 \leftarrow x_0' & = x_0' \\ y_0 \leftarrow y_0' \oplus x_0 & = y_0' \oplus x_0' \\ z_0 \leftarrow z_0' \oplus y_0 & = z_0' \oplus y_0' \oplus x_0' \end{cases}$$

$$f_1 = \begin{cases} x_1' \leftarrow x_1 \oplus z_0 \\ y_1' \leftarrow y_1 \oplus x_1 \oplus (x_0 \vee z_0) \\ z_1' \leftarrow z_1 \oplus y_1 \end{cases} \qquad f_1^{-1} = \begin{cases} x_1 \leftarrow x_1' \oplus z_0 & = x_1' \oplus z_0 \\ y_1 \leftarrow y_1' \oplus x_1 \oplus (x_0 \vee z_0) & = y_1' \oplus x_1' \oplus z_0 \oplus (x_0 \vee z_0) \\ z_1 \leftarrow z_1' \oplus y_1 & = z_1' \oplus y_1' \oplus x_1' \oplus z_0 \oplus (x_0 \vee z_0) \end{cases}$$

$$f_2 = \begin{cases} x_2' \leftarrow x_2 \oplus z_1 \oplus (y_0 \wedge z_0) \\ y_2' \leftarrow y_2 \oplus x_2 \oplus (x_1 \vee z_1) \\ z_2' \leftarrow z_2 \oplus y_2 \end{cases} \qquad f_2^{-1} = \begin{cases} x_2 \leftarrow x_2' \oplus z_1 \oplus (y_0 \wedge z_0) & = x_2' \oplus z_1 \oplus (y_0 \wedge z_0) \\ y_2 \leftarrow y_2' \oplus x_2 \oplus (x_1 \vee z_1) & = y_2' \oplus x_2' \oplus z_1 \oplus (y_0 \wedge z_0) \oplus (x_1 \vee z_1) \\ z_2 \leftarrow z_2' \oplus y_2 & = z_2' \oplus y_2' \oplus x_2' \oplus z_1 \oplus (y_0 \wedge z_0) \oplus (x_1 \vee z_1) \end{cases}$$

and

$$f_n = \begin{cases} x_n' \leftarrow x_n \oplus z_{n-1} \oplus (y_{n-2} \wedge z_{n-2}) \\ y_n' \leftarrow y_n \oplus x_n \oplus (x_{n-1} \vee z_{n-1}) \\ z_n' \leftarrow z_n \oplus y_n \oplus (x_{n-3} \wedge z_{n-3}) \end{cases}$$

and

$$f_n^{-1} = \begin{cases} x_n \leftarrow x_n' \oplus z_{n-1} \oplus (y_{n-2} \wedge z_{n-2}) \\ y_n \leftarrow y_n' \oplus x_n' \oplus z_{n-1} \oplus (y_{n-2} \wedge z_{n-2}) \oplus (x_{n-1} \vee z_{n-1}) \\ z_n \leftarrow z_n' \oplus y_n' \oplus x_n' \oplus z_{n-1} \oplus (y_{n-2} \wedge z_{n-2}) \oplus (x_{n-1} \vee z_{n-1}) \oplus (x_{n-3} \wedge z_{n-3}) \end{cases}$$

SP^{-1} is fully defined by recurrence. SP is therefore bijective.

Gimli in C99 (268 chars)

```c
#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)*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];
  }
```

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