NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

Automated Analysis for Pushing Performance Limits in Symmetric-Key Cryptography

Thomas Peyrin
NTU Singapore

IWSEC 2024 - Kyoto
17th September 2024

Problem Statement

Cryptographic design is always a fight **performance** vs **security**

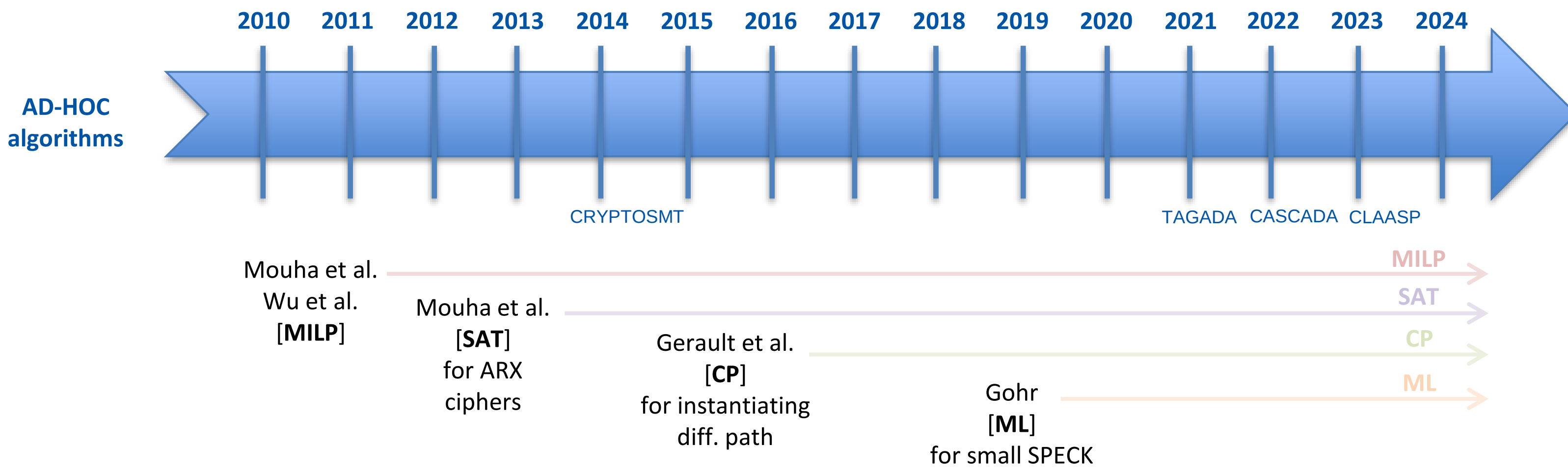
Performance is usually modeled according to some physical/technological model, and the community is now considering more and more exotic metrics (lightweight, low-latency, MPC-friendly, etc)

Security analysis was done by humans and now more and more assisted by automated tools.

Can automated tools be more integrated within the design process ?

Automated Cryptanalysis

Timeline of Automated Cryptanalysis



Today almost all cryptanalysis are assisted by solvers (FSE 2024: two entire sessions just for pure automated cryptanalysis)

Automated cryptanalysis using declarative frameworks (SAT/MILP/CP/etc.) is generally slower or at best same as ad-hoc tools, but so much **more convenient**

Mainly on **differential** and **linear cryptanalysis**, but now also on integral distinguishers, cube attacks, meet-in-the-middle attacks, etc.

Solving time is a crucial aspect and can be impacted by:

- the framework you use (SAT/MILP/CP/etc.)
- the strategy of modeling (many works on various modeling strategies)
- the solver (less contributions on that, different research field)
- the type of problem studied / scale

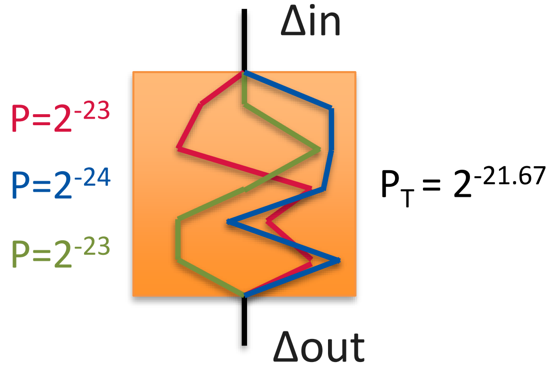
Automated Cryptanalysis for Differential Paths

Typically, for finding **differentials** or **differential trails**:

- Use **variables** to represent the various stages of the internal state bit differences during the round (and throughout the rounds)
- Use other variables to represent the **probability P** of the differential path (in -log2)
- Model a round of the cipher as a set of **declarative constraints** (Markov assumption !) to represent the difference propagation (either truncated or not). Use temporary variables if needed for certain components.
- Put all this into a system and use a **solver** on it.
- Can be combined with extra upper-level strategies (Matsui branch-and-bound, etc.)

One can:

- Find the best differential path / linear characteristic
- Enumerate the number of solutions
- Estimate the probability of a differential



Open Cryptanalysis Platform (Open-CP)

OPEN-CP: a new collaborative cryptanalysis platform

- In collaboration with many cryptanalysts
- **Free** and **open source**
- **Easy** to use / contribute
- Start simple (differential / linear)
- **Goal:** become the go-to platform for creating / testing / benchmarking cryptanalysis
- Need to establish governance to have proper development process into place, regular meetings, …

https://github.com/Open-CP/OCP

Easy and Fast cipher definition

```python
# The Speck internal permutation
class Speck_permutation(Permutation):
    def __init__(self, name, version, s_input, s_output, nbr_rounds=None, model_type=0):

        p_bitsize = version
        if nbr_rounds==None: nbr_rounds=22 if version==32 else 22 if version==48 else 26 if version==64 else 28 if version==96 else 32 if version==128 else None
        if model_type==0: nbr_layers, nbr_words, nbr_temp_words, word_bitsize = 4, 2, 0, p_bitsize>>1
        super().__init__(name, s_input, s_output, nbr_rounds, [nbr_layers, nbr_words, nbr_temp_words, word_bitsize])

        if version==32: rotr, rotl = 7, 2
        else: rotr, rotl = 8, 3

        # create constraints
        if model_type==0:
            for i in range(1,nbr_rounds+1):
                self.states["STATE"].RotationLayer("ROT1", i, 0, ['r', rotr], 0) # Rotation layer
                self.states["STATE"].SingleOperatorLayer("ADD", i, 1, op.ModAdd, [0,1], [0]) # Modular addition layer
                self.states["STATE"].RotationLayer("ROT2", i, 2, ['l', rotl], 1) # Rotation layer
                self.states["STATE"].SingleOperatorLayer("XOR", i, 3, op.bitwiseXOR, [0,1], [1]) # XOR layer
```

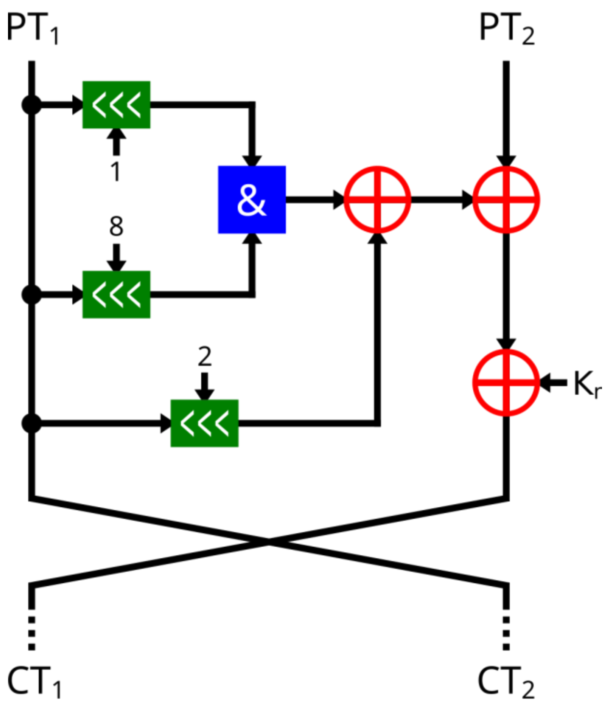
```python
# The Skinny internal permutation
class Skinny_permutation(Permutation):
    def __init__(self, name, version, s_input, s_output, nbr_rounds=None, model_type=0):

        p_bitsize = version
        if nbr_rounds==None: nbr_rounds=32 if version==64 else 64 if version==128 else None
        if model_type==0:  nbr_layers, nbr_words, nbr_temp_words, word_bitsize = 4, 16, 0, int(p_bitsize/16)
        super().__init__(name, s_input, s_output, nbr_rounds, [nbr_layers, nbr_words, nbr_temp_words, word_bitsize])

        # create constraints
        if model_type==0:
            for i in range(1,nbr_rounds+1):
                if word_bitsize==4: self.states["STATE"].SboxLayer("SB", i, 0, op.Skinny_4bit_Sbox)
                else: self.states["STATE"].SboxLayer("SB", i, 0, op.Skinny_8bit_Sbox)  # Sbox layer
                self.states["STATE"].AddConstantLayer("C", i, 1, "xor", [0,0,0,0, 0,0,0,0, 2,0,0,0, 0,0,0,0])  # Constant layer
                self.states["STATE"].PermutationLayer("SR", i, 2, [0,1,2,3, 7,4,5,6, 10,11,8,9, 13,14,15,12]) # Shiftrows layer
                self.states["STATE"].MatrixLayer("MC", i, 3, [[1,0,1,1], [1,0,0,0], [0,1,1,0], [1,0,1,0]], [[0,4,8,12], [1,5,9,13], [2,6,10,14], [3,7,11,15]])  #Mixcolumns layer
```

Modeling Example of OPEN-CP

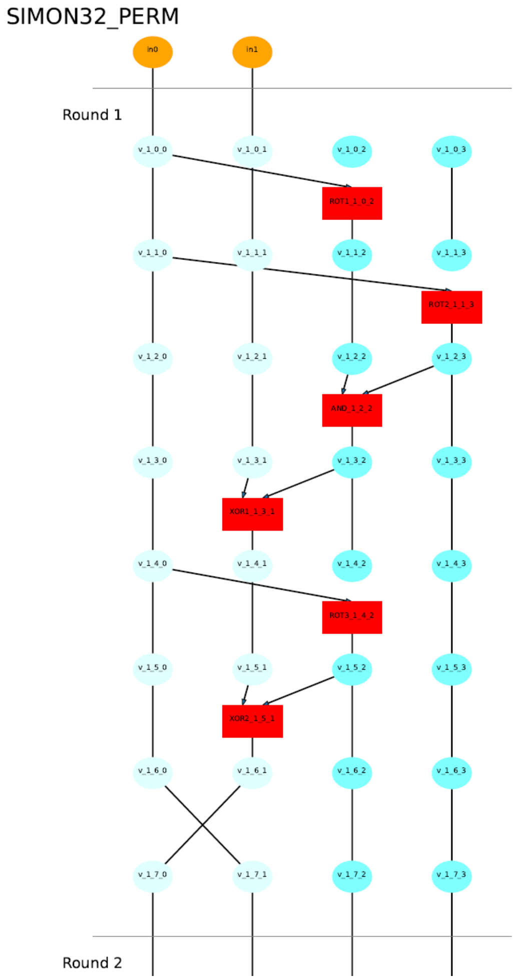
variables

operators

Example: SIMON-32 permutation



(image from Wikipedia)



Automatic Generation of C / Python code

```python
#Rotation Macros
def ROTL(n, d, bitsize): return ((n << d) | (n >> (bitsize - d))) & (2**bitsize - 1)
def ROTR(n, d, bitsize): return ((n >> d) | (n << (bitsize - d))) & (2**bitsize - 1)

# Function implementing the SIMON32_PERM function
# Input:
#   IN: a list of 2 words of 16 bits
# Output:
#   OUT: a list of 2 words of 16 bits
def SIMON32_PERM(IN, OUT):

    # Input
    v_0_0 = IN[0]
    v_0_1 = IN[1]
    v_0_2 = v_0_3 = 0

    # Round function
    for i in range(10):
        v_1_0 = v_0_0
        v_1_1 = v_0_1
        v_1_2 = ROTL(v_0_0, 1, 16)
        v_1_3 = v_0_3
        v_2_0 = v_1_0
        v_2_1 = v_1_1
        v_2_2 = v_1_2
        v_2_3 = ROTL(v_1_0, 8, 16)
        v_3_0 = v_2_0
        v_3_1 = v_2_1
        v_3_2 = v_2_2 & v_2_3
        v_3_3 = v_2_3
        v_4_0 = v_3_0
        v_4_1 = v_3_1 ^ v_3_2
        v_4_2 = v_3_2
        v_4_3 = v_3_3
        v_5_0 = v_4_0
        v_5_1 = v_4_1
        v_5_2 = ROTL(v_4_0, 2, 16)
        v_5_3 = v_4_3
        v_6_0 = v_5_0
        v_6_1 = v_5_1 ^ v_5_2
        v_6_2 = v_5_2
        v_6_3 = v_5_3
        v_7_0 = v_6_1
        v_7_1 = v_6_0
        v_7_2 = v_6_2
        v_7_3 = v_6_3
        v_0_0 = v_7_0
        v_0_1 = v_7_1
        v_0_2 = v_7_2
        v_0_3 = v_7_3

    # Output
    OUT[0] = v_7_0
    OUT[1] = v_7_1

# test implementation
IN = [0x0, 0x0]
OUT = [0x0, 0x0]
SIMON32_PERM(IN, OUT)
print('IN', str([hex(i) for i in IN]))
print('OUT', str([hex(i) for i in OUT]))
```

```c
#include <stdint.h>
#include <stdio.h>

//Rotation Macros
#define ROTL(n, d, bitsize) (((n << d) | (n >> (bitsize - d))) & ((1<<bitsize) - 1))
#define ROTR(n, d, bitsize) (((n >> d) | (n << (bitsize - d))) & ((1<<bitsize) - 1))

// Function implementing the SIMON32_PERM function
// Input:
//   IN: an array of 2 words of 16 bits
// Output:
//   OUT: an array of 2 words of 16 bits
void SIMON32_PERM(uint32_t* IN, uint32_t* OUT){
    uint32_t v_0_0, v_0_1, v_0_2, v_0_3, v_1_0, v_1_1, v_1_2, v_1_3, v_2_0, v_2_1, v_2_2, v_2_3,

    // Input
    v_0_0 = IN[0];
    v_0_1 = IN[1];

    // Round function
    for (int i=0; i<10; i++) {
        v_1_0 = v_0_0;
        v_1_1 = v_0_1;
        v_1_2 = ROTL(v_0_0, 1, 16);
        v_1_3 = v_0_3;
        v_2_0 = v_1_0;
        v_2_1 = v_1_1;
        v_2_2 = v_1_2;
        v_2_3 = ROTL(v_1_0, 8, 16);
        v_3_0 = v_2_0;
        v_3_1 = v_2_1;
        v_3_2 = v_2_2 & v_2_3;
        v_3_3 = v_2_3;
        v_4_0 = v_3_0;
        v_4_1 = v_3_1 ^ v_3_2;
        v_4_2 = v_3_2;
        v_4_3 = v_3_3;
        v_5_0 = v_4_0;
        v_5_1 = v_4_1;
        v_5_2 = ROTL(v_4_0, 2, 16);
        v_5_3 = v_4_3;
        v_6_0 = v_5_0;
        v_6_1 = v_5_1 ^ v_5_2;
        v_6_2 = v_5_2;
        v_6_3 = v_5_3;
        v_7_0 = v_6_1;
        v_7_1 = v_6_0;
        v_7_2 = v_6_2;
        v_7_3 = v_6_3;
        v_0_0 = v_7_0;
        v_0_1 = v_7_1;
        v_0_2 = v_7_2;
        v_0_3 = v_7_3;

    }

    // Output
    OUT[0] = v_7_0;
    OUT[1] = v_7_1;
}
```

Automatic Generation of SAT / MILP models

```
v_3_3_0_1 - v_3_4_0_1 = 0
v_3_3_0_2 - v_3_4_0_2 = 0
v_3_3_0_3 - v_3_4_0_3 = 0
v_3_3_0_4 - v_3_4_0_4 = 0
v_3_3_0_5 - v_3_4_0_5 = 0
v_3_3_0_6 - v_3_4_0_6 = 0
v_3_3_0_7 - v_3_4_0_7 = 0
v_3_3_0_8 - v_3_4_0_8 = 0
v_3_3_0_9 - v_3_4_0_9 = 0
v_3_3_0_10 - v_3_4_0_10 = 0
v_3_3_0_11 - v_3_4_0_11 = 0
v_3_3_0_12 - v_3_4_0_12 = 0
v_3_3_0_13 - v_3_4_0_13 = 0
v_3_3_0_14 - v_3_4_0_14 = 0
v_3_3_0_15 - v_3_4_0_15 = 0
v_3_3_0_0 + v_3_3_1_0 + v_3_4_1_0 - 2 XOR_3_3_1_d_0 >= 0
v_3_3_0_0 + v_3_3_1_0 + v_3_4_1_0 <= 2
XOR_3_3_1_d_0 - v_3_3_0_0 >= 0
XOR_3_3_1_d_0 - v_3_3_1_0 >= 0
XOR_3_3_1_d_0 - v_3_4_1_0 >= 0
v_3_3_0_1 + v_3_3_1_1 + v_3_4_1_1 - 2 XOR_3_3_1_d_1 >= 0
v_3_3_0_1 + v_3_3_1_1 + v_3_4_1_1 <= 2
XOR_3_3_1_d_1 - v_3_3_0_1 >= 0
XOR_3_3_1_d_1 - v_3_3_1_1 >= 0
XOR_3_3_1_d_1 - v_3_4_1_1 >= 0
v_3_3_0_2 + v_3_3_1_2 + v_3_4_1_2 - 2 XOR_3_3_1_d_2 >= 0
v_3_3_0_2 + v_3_3_1_2 + v_3_4_1_2 <= 2
XOR_3_3_1_d_2 - v_3_3_0_2 >= 0
XOR_3_3_1_d_2 - v_3_3_1_2 >= 0
XOR_3_3_1_d_2 - v_3_4_1_2 >= 0
v_3_3_0_3 + v_3_3_1_3 + v_3_4_1_3 - 2 XOR_3_3_1_d_3 >= 0
v_3_3_0_3 + v_3_3_1_3 + v_3_4_1_3 <= 2
XOR_3_3_1_d_3 - v_3_3_0_3 >= 0
XOR_3_3_1_d_3 - v_3_3_1_3 >= 0
XOR_3_3_1_d_3 - v_3_4_1_3 >= 0
v_3_3_0_4 + v_3_3_1_4 + v_3_4_1_4 - 2 XOR_3_3_1_d_4 >= 0
v_3_3_0_4 + v_3_3_1_4 + v_3_4_1_4 <= 2
XOR_3_3_1_d_4 - v_3_3_0_4 >= 0
XOR_3_3_1_d_4 - v_3_3_1_4 >= 0
XOR_3_3_1_d_4 - v_3_4_1_4 >= 0
v_3_3_0_5 + v_3_3_1_5 + v_3_4_1_5 - 2 XOR_3_3_1_d_5 >= 0
v_3_3_0_5 + v_3_3_1_5 + v_3_4_1_5 <= 2
XOR_3_3_1_d_5 - v_3_3_0_5 >= 0
XOR_3_3_1_d_5 - v_3_3_1_5 >= 0
XOR_3_3_1_d_5 - v_3_4_1_5 >= 0
v_3_3_0_6 + v_3_3_1_6 + v_3_4_1_6 - 2 XOR_3_3_1_d_6 >= 0
v_3_3_0_6 + v_3_3_1_6 + v_3_4_1_6 <= 2
XOR_3_3_1_d_6 - v_3_3_0_6 >= 0
XOR_3_3_1_d_6 - v_3_3_1_6 >= 0
XOR_3_3_1_d_6 - v_3_4_1_6 >= 0
v_3_3_0_7 + v_3_3_1_7 + v_3_4_1_7 - 2 XOR_3_3_1_d_7 >= 0
v_3_3_0_7 + v_3_3_1_7 + v_3_4_1_7 <= 2
XOR_3_3_1_d_7 - v_3_3_0_7 >= 0
XOR_3_3_1_d_7 - v_3_3_1_7 >= 0
```

Future of OPEN-CP

- **More attacks** ! (boomerang / impossible diff / division property / etc .)
- **Key recovery** phase
- **Graphical interface** for user interaction (cipher design / attack config.)
- Automatic generation of cipher **implementations**, test vectors, attacks
- Parallelization
- Testing on reduced rounds
- Pre-existing **library of ciphers and attacks**
- Differential path drawing, LaTeX/TikZ code generation
- Allow **modular combination of attacks/models**
- Optimized Sbox / Diffusion matrix implementations database

We want YOU !

If interested to participate / getting updates:

- contact me at thomas.peyrin@ntu.edu.sg
- or join the googlegroup

automated-cryptanalysis@googlegroups.com

- or clink on this link:

https://groups.google.com/g/automated-cryptanalysis

- GitHub:

https://github.com/Open-CP/OCP

Automated Cryptanalysis for Designers

Classical design process: cipher's structure is pre-established by the human. The computer will brute force some components (Sbox, diffusion matrix) or parameters (rotation constant, etc.) to select the best candidate.

However:

- There is no "search" per se, it is just brute force search and taking the best candidate
- Evaluation of the cipher's security and performance is done at the end (no insight to search in a smart way)

Can we give more freedom for the computer to create good ciphers ?

Can automated cryptanalysis help us searching for good ciphers ?

Fast AES-based MAC

LeMac - PetitMac

Fast AES-Based Universal Hash Functions and MACs (Featuring LeMac and PetitMac) – **ToSC 2024-2**

Joint work with A. Bariant, J. Baudrin, G. Leurent, C. Pernot and L. Perrin

Why Fast MAC ?

- AES has globally good performances, but it is **really fast in practice** because of **hardware acceleration** widely available (AES-NI).
- The granularity of AES-NI is on the **AES round**, so it has been used to build many fast primitives:
  - Hash functions (ECHO, LANE, SHAVITE-3, VORTEX, etc.),
  - AEAD schemes (AEGIS, TIAOXIN-346, DEOXYS, ROCCA(-S), etc.),
  - Permutations (AREION, SIMPIRA, HARAKA, PHOLKOS, etc.).
- Now, not so difficult to reach throughput < 1 c/B on typical processors

  Ex: 2 AES rounds in parallel each cycle, thus (10/2)/16 = 0.31 c/B
- But sixth-generation mobile comm. systems (6G) to deliver an amazing throughput of 100 Gbps to 1 Tbps (0.24 to 0.024 c/B on a 3GHz CPU) !

We need to create primitives with even much larger throughput !

AES-based UHF-based MACs

UHF-based MAC:

- GMAC, Poly1305 uses **Wegman-Carter-Shoup** with only $2^{n/2}$ / 0 security for nonce-respecting / misuse

$$\mathrm{WCS}[H,E]_{k_1,k_2}(M,N) = H_{k_1}(M) \oplus E_{k_2}(N)$$

- **EWCDM** gives 2^{n} / $2^{n/2}$ for nonce-respecting / misuse

$$\mathrm{EWCDM}[H,E]_{k_1,k_2,k_3}(M,N) = E_{k_3}\big(H_{k_1}(M) \oplus E_{k_2}(N) \oplus N\big)$$

AES-based UHFs: PC-MAC and EliMAC (rate of 4 AES rounds per block).

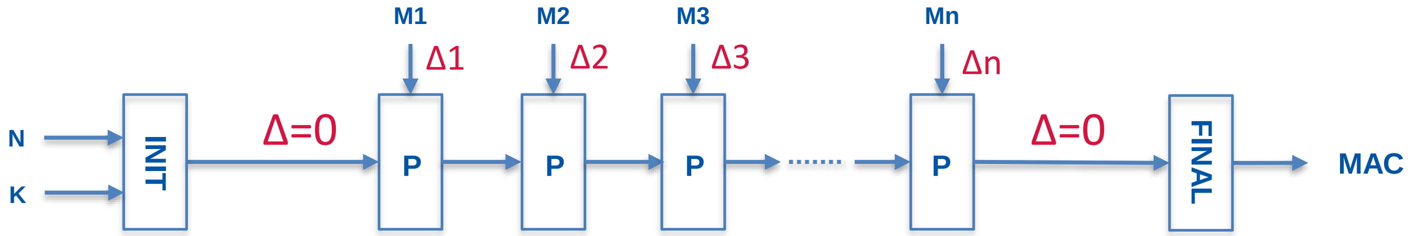
Our MACs (LeMac and PetitMac): 128-bit key, 128-bit tag

AES-based 2^{-128} UHF with rate 2 AES rounds/block in EWCDM.

State-of-the-art of Fast AES-based MAC

Many ultra-fast AES-based collision resistant permutations:

AEGIS, TIAOXIN-346, ROCCA-(S), Jean-Nikolić [JN16] and Nikolić [Nik17a] (fastest)



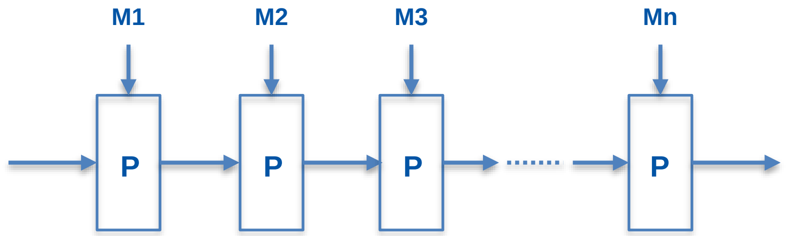
Goal: guarantee **no collision path** exist with good probability

ROCCA targets 256-bit key / 128-bit tag AEAD. Some security issues [HII+22].
ROCCA-S targets 256-bit key / 256-bit tag AEAD (under submission at IETF).

Sub-optimal throughput: optimal in ROCCA framework [TSI23] reaches 0.104 c/B on Tiger Lake, while theoretical max is 0.0625 c/B.

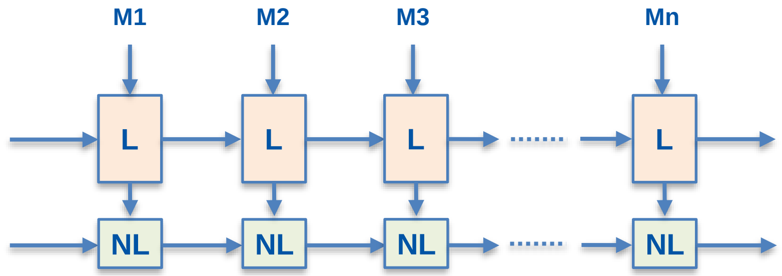
Designing a collision-resistant permutation

Classical: large state entirely updated non-linearly. Issue: costly for a large state.



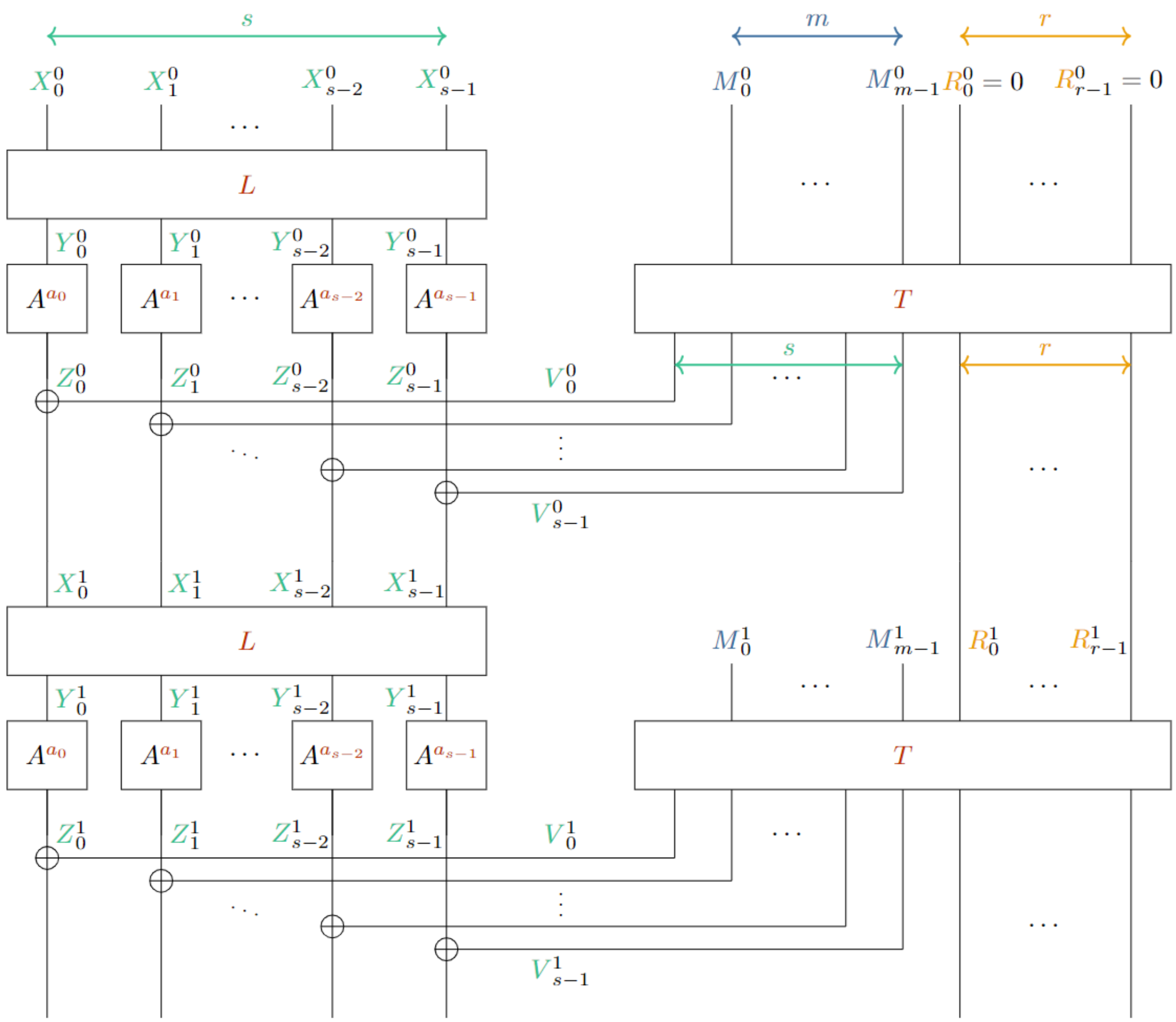
Better ?: large state separated in two parts (inspired from TBC or PANAMA hash):

- **one part updated with (expensive) non-linear components** (AES round in our case)
- **one part updated with linear components** (not influenced by the first one, reducing dependencies that complicate instructions scheduling and automated security analysis).



Our overall permutation structure

- **Framework** more general than previous ones
- **Goal:** no differential path with P > 2^{-128}
- initialization / finalization
- A is AES round, T and L are linear matrices
- AddRoundKey is free with AES-NI: we can use a free XOR after each AES round
- Increasing r and s generally improves performance, but we limit to s + r <16



Automatic security and performance analysis

Security analysis:

- a MILP model to evaluate diff. paths automatically without linear incompatibilities (cheap)
- another MILP model with linear incompatibilities (quite expensive)

Performance benchmark: an automatic implementation is produced for each candidate (quite cheap) to benchmark them.

- so performant that XOR becomes important (carefully consider AES-NI / XOR latency, throughput, ports). For x AES rounds, make x/2 XOR max (unlike Jean-Nikolic or Rocca).
- Dependency chains are also important: Rocca in decryption has long chains (reduced perf.)
- Many other complex things to consider, so the best way is to actually benchmark directly

Architecture	Instr	Latency	Throughput	P_0	P_1	P_2	P_3	P_4	P_5	P_6
Intel Haswell	XOR	1	0.33	x	x				x	
	AESENC	7	1						x	
Intel Skylake	XOR	1	0.33	x	x				x	
	AESENC	4	1	x						
Intel Ice Lake	XOR	1	0.33	x	x				x	
	AESENC	3	0.5	x	x					
Intel Tiger Lake	XOR	1	0.33	x	x				x	
	AESENC	3	0.5	x	x					
AMD Zen 1/2/3/4	XOR	1	0.25	x	x	x	x			
	AESENC	4	0.5	x	x					

Scheduling of AESENC and XOR instructions on modern processors

Handling a large search space

Extremely large search space, so we reduce it by:

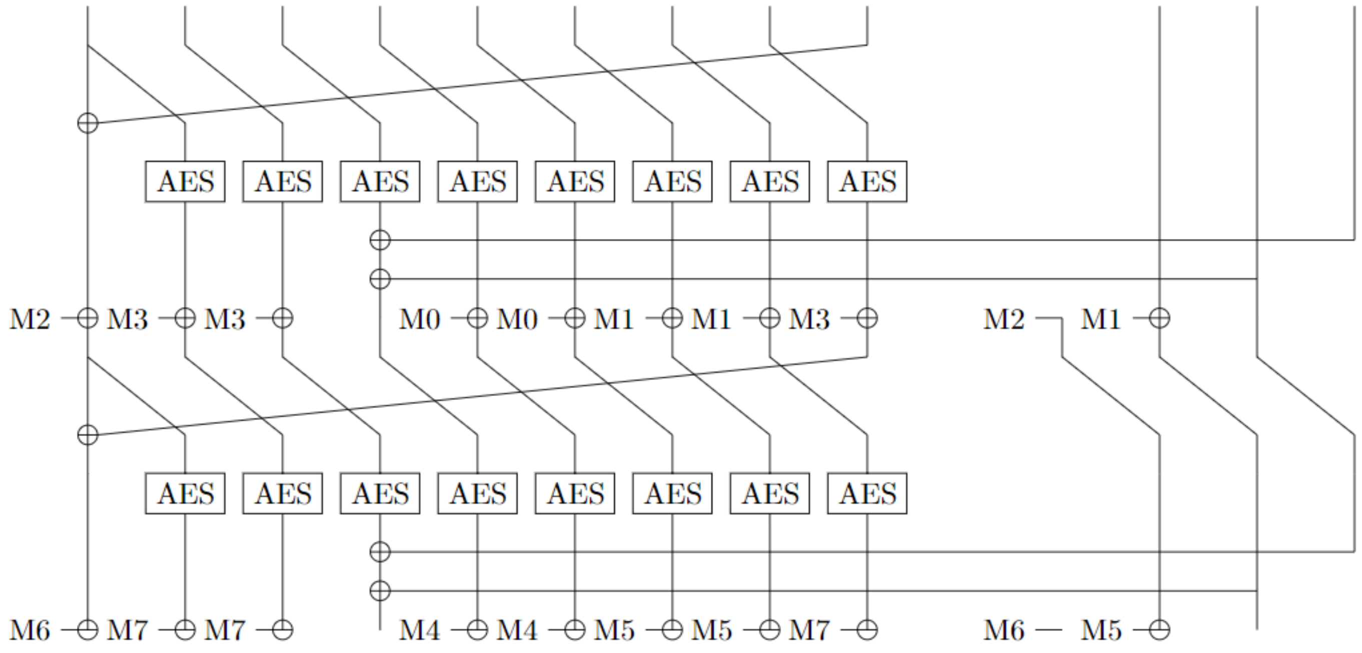
- leveraging symmetries
- select subparts that are interesting (limit #XORs, higher diffusion power of the matrices)

Our search strategy (NEW):

LeMac (128-bit key / 128-bit tag)

- The state is composed of **13 128-bit words** (9 in non-linear part, 4 in linear)
- 8 AES rounds for 4 message blocks (rate 2), only 4 extra XORs (**perfect ratio**)
- **Security:** at least **26 active Sboxes** (diff. path probability < $2^{-6*26} = 2^{-156}$)

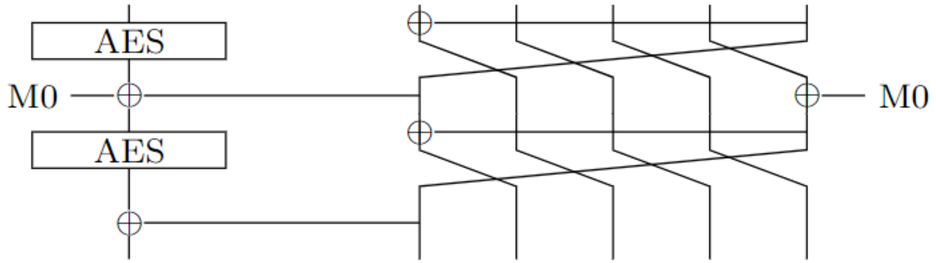
2 rounds of the UHF of LeMac



PetitMac (128-bit key / 128-bit tag)

- The state is composed of **6 128-bit words** (1 in non-linear part, 5 in linear)
- 2 AES rounds for 1 message block (rate 2), 3 extra XORs
- **Security:** at least **26 active Sboxes** (diff. path probability < $2^{-(26*6)} = 2^{-156}$)

1 round of the UHF of PetitMac



Performance results

< 0.1 c/B throughput for LeMac ! (Using only 128-bit instructions, not AVX-512).

The **fastest MAC** (by far) on medium/high-end processors.

PetitMAC aims for a better **tradeoff** on constrained devices: AES round-based MAC with rate 2, with acceptable memory footprint.

18.3 c/B on ARM Cortex-M4.

		Speed (c/B)		
CPU	Cipher	1kB	16kB	256kB
Intel Haswell (Xeon E5-2630 v3)	GCM (AD only)	1.138	0.700	0.605
	Rocca (AD only)	0.602	0.225	0.201
	Rocca-S (AD only)	0.660	0.290	0.269
	AEGIS128 (AD only)	0.809	0.578	0.564
	AEGIS128L (AD only)	0.542	0.299	0.285
	Tiaoxin-346 v2 (AD only)	0.489	0.207	0.190
	Jean-Nikolić	0.455	0.149	0.159
	LeMac	0.498	0.148	0.131
	PetitMac	1.116	0.890	0.876
Intel Skylake (Xeon Gold 6130)	GCM (AD only)	0.817	0.396	0.370
	Rocca (AD only)	0.573	0.190	0.167
	Rocca-S (AD only)	0.568	0.213	0.192
	AEGIS128 (AD only)	0.682	0.470	0.460
	AEGIS128L (AD only)	0.505	0.267	0.253
	Tiaoxin-346 v2 (AD only)	0.473	0.206	0.189
	Jean-Nikolić	0.389	0.142	0.130
	LeMac	0.422	0.144	0.126
	PetitMac	0.792	0.635	0.626
Intel Ice Lake (Xeon Gold 5320)	GCM (AD only)	0.699	0.311	0.286
	Rocca (AD only)	0.528	0.171	0.149
	Rocca-S (AD only)	0.478	0.172	0.151
	AEGIS128 (AD only)	0.619	0.401	0.389
	AEGIS128L (AD only)	0.416	0.208	0.195
	Tiaoxin-346 v2 (AD only)	0.328	0.131	0.121
	Jean-Nikolić	0.307	0.126	0.113
	LeMac	0.289	0.082	0.068
	PetitMac	0.521	0.384	0.376

Code: https://github.com/AugustinBariant/Implementations_LeMac_PetitMac

Future of LeMac / PetitMac

- What about **(Authenticated)-Encryption** ?
- What about 256-bit keys (mandated by 6G) and 256-bit tags ?
- Probably **difficult to do faster**:
  - we are at the performance theoretical limit for rate 2
  - we proposed candidates with rate < 2, but practical performance is not improved
- Consider using LeMac/PetitMac as building blocks for amazing speed ! (NIST “Accordion cipher” ?)

Low-Latency Cryptography

Under preparation

Joint work with K. Hu., M. Khairallah and Q. Q. Tan

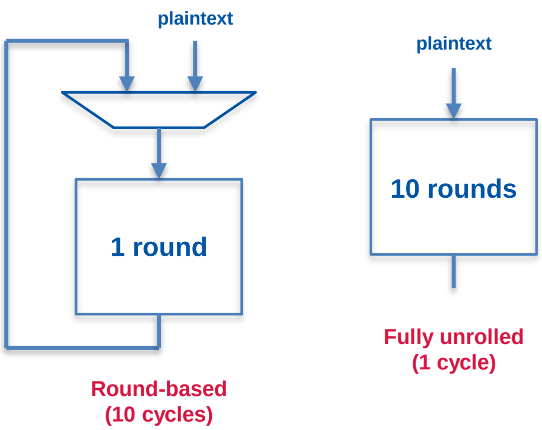
Why Low-latency

AES good for general usage, but lot of attention on lightweight cryptography in the past 15 years. NIST has standardized ASCON, **what's next ?**

In some applications, the **latency** (time it takes to produce the ciphertext byte/block of a corresponding plaintext byte/block) is very important:

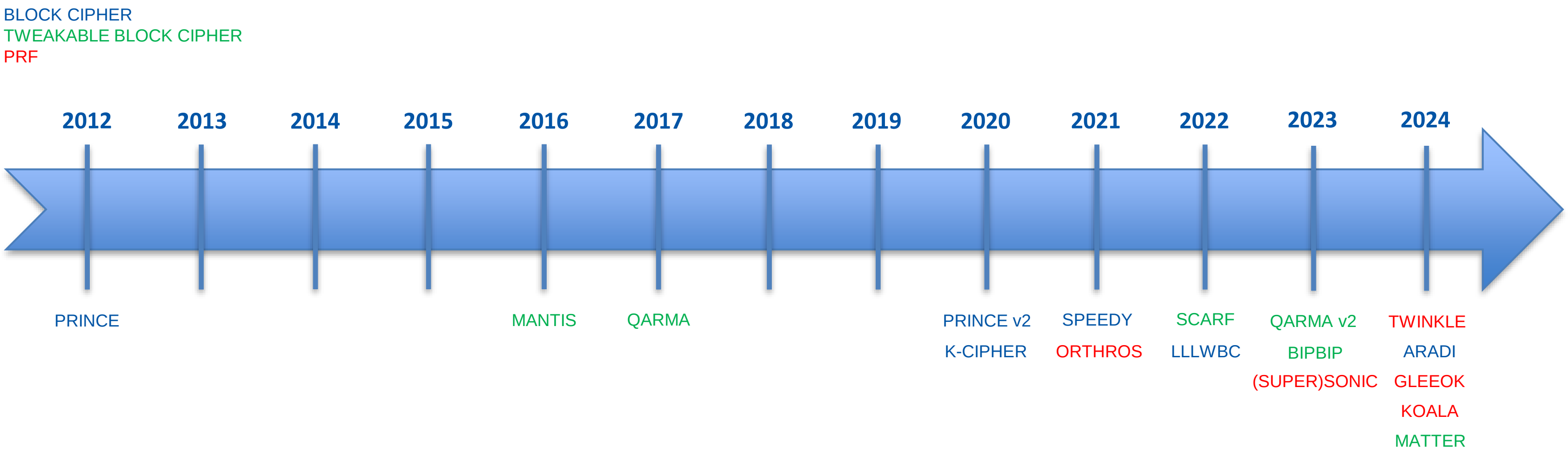
- RAM memory encryption/authentication (typically with a hardware memory encryption engine), especially with the rise of cloud computing,
- sensor data encryption/authentication (critical systems, automotive)
- system security (pointer authentication)

We talk about hardware (ASIC principally, or FPGA), with **fully unrolled implementations** (entire cipher in a single cycle, but lower freq.).



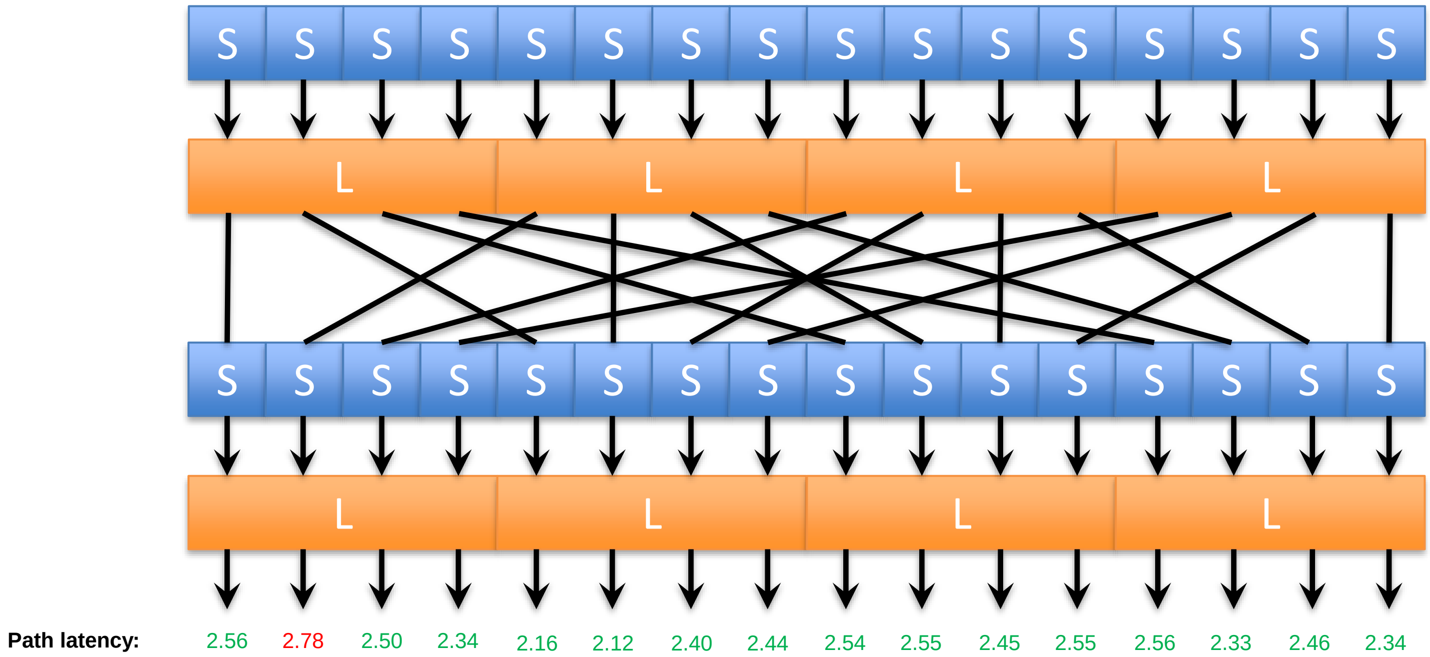
Here we consider the **internal primitive**, not the operating mode.

Low-latency cryptography timeline



- **PRINCE** was the first cipher to claim **latency** as main performance goal
- Low-latency trend is accelerating
- We now have BC, TBC, PRF candidates
- Design strategy is to use special Sboxes, linear layers, combinations of them, special structures, to reduce latency locally while maintaining security
- Special **operating modes** have also been proposed

Why Low-latency is difficult ?



In contrary to area/throughput, **it is difficult to predict the latency accurately in practice.**

It is also **difficult to know in advance the critical path** of the implementation and the impact that a change on one internal component might do to the latency.

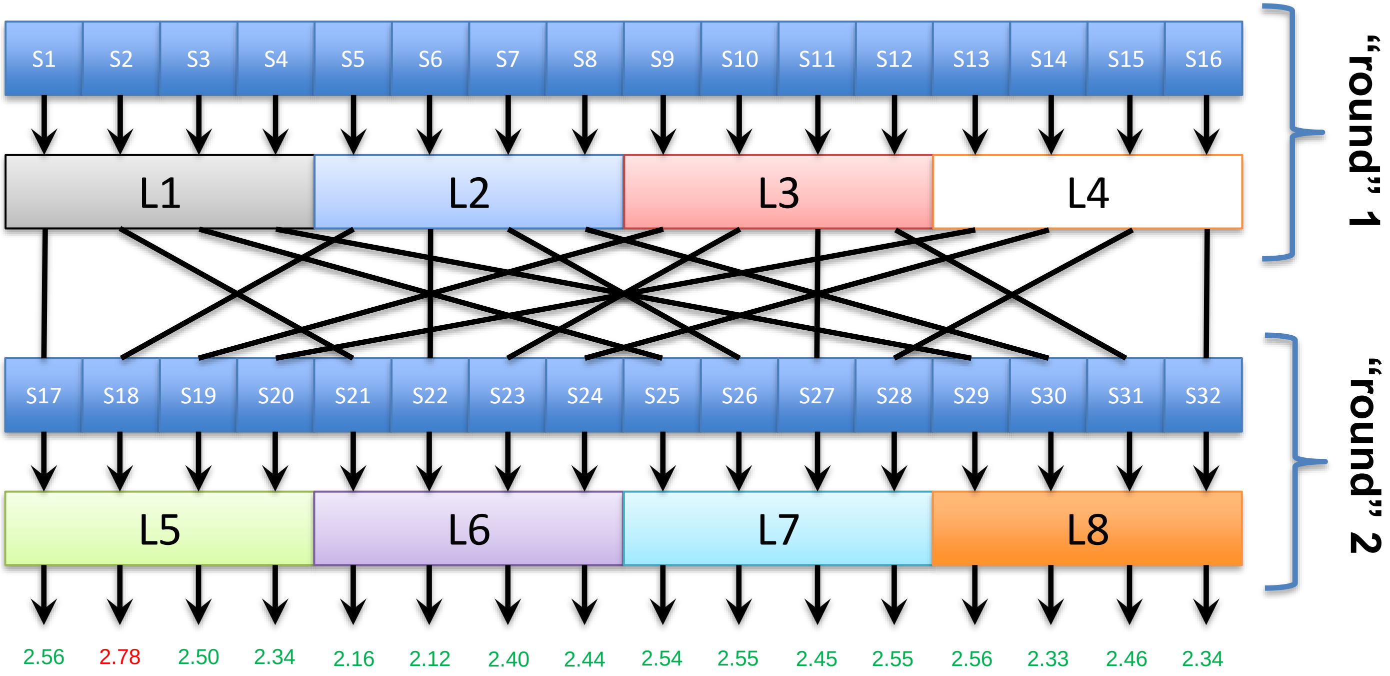
Breaking the iterative round paradigm

Low latency ciphers are used with **unrolled implementation**, so **no need to follow a classical round structure anymore** (**NEW**) !

Problem: the security analysis becomes difficult for humans
Solution (NEW): let automated cryptanalysis guide the design !

Two benefits:

- One can create the cipher **round per round**
- We can adapt each round (and each component within a round) separately to **minimize the max path latency**



Beyond auto cryptanalysis: auto implementations

Using the cipher's performance as a design target:

1st level: do not estimate the implementations performance during design phase, simply **make assumptions** on what makes a scheme performant and select building bricks accordingly.

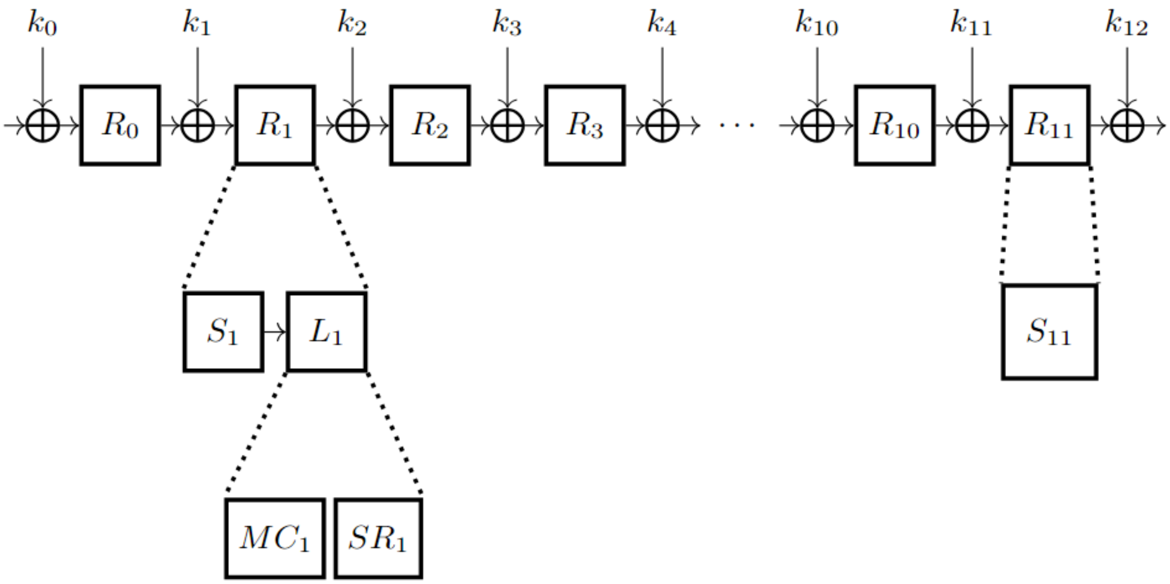
2nd level: while searching for which bricks to use or how to combine them, use a **model** to estimate the performance of the candidate design.

3rd level (NEW): while searching for which bricks to use or how to combine them, **generate automatically an actual implementation** of the design and estimate its performance. We used OpenLane (an Open-source VLSI flow) for estimating hardware performance.

The uKNIT Cipher

The **uKNIT extremely low-latency block cipher** structure:

- Classical **64-bit SPN,** with sixteen **4-bit low-latency Sboxes**, each can be different (bit-permuted variants of the MANTIS Sbox)
- Special **low-latency linear layers**
- **Each round can be different !**
- **Key Schedule: New** generalization of the STK construction



Building the cipher: Evolutionary Algorithm

Problem: the **search space is now VERY large** (sboxes, linear layers)

Solution: we use an **evolutionary algorithm** to search in that large space, optimizing for good latency/security tradeoff.

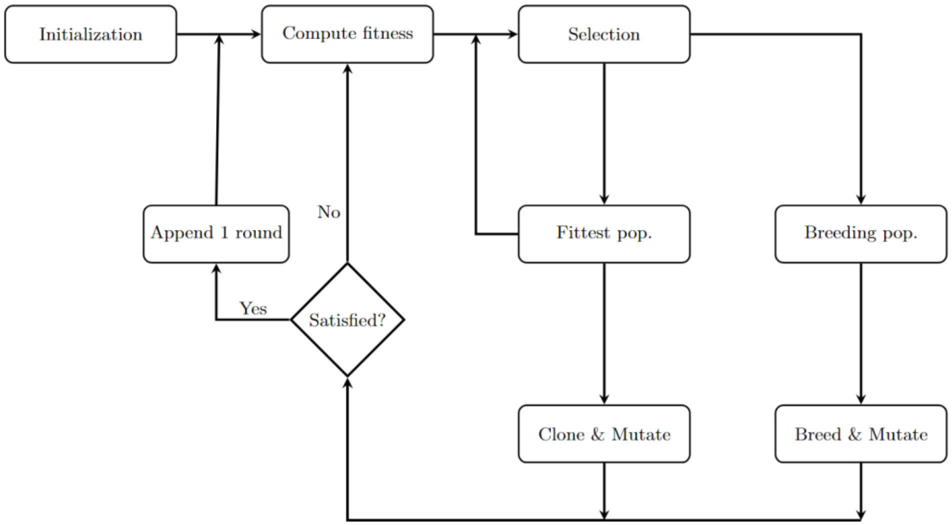
Importance of the **objective function**:

- If too latency oriented, not good
- If too security oriented, not good

$$\frac{\max[-\log_2(prob_d), -2 \cdot \log_2(bias_l)]^2}{lat}$$

We start from good candidates on 3 rounds. Then, we proceed **round per round** until reaching 12 rounds.

Our design is fully automated (almost **NEW** [Nikolić 2017])



Security of uKNIT

uKNIT has a **good resistance against differential and linear cryptanalysis**.

We also studied many other state-of-the-art cryptanalysis.

Stronger diff/linear resistance than PRINCE.

Differential probabilities for all windows of r-round

Win \ Rnd	0	1	2	3	4	5	6	7	8	9	10	11	PRINCE
1	2	2	2	2	2	2	2	2	2	2	2	2	–
2	8	8	8	8	8	6	8	8	8	8	8	–	–
3	14	14	14	13	12	14	14	14	14	14	–	–	–
4	32	29	25	19	23	22	32	32	32	–	–	–	32
5	43	40	33	31	41	45	41	40	–	–	–	–	39
6	55	48	45	50	54	53	49	–	–	–	–	–	44
7	61	59	58	62	65	63	–	–	–	–	–	–	56
8	67	71	73	71	74	–	–	–	–	–	–	–	66
9	85	88	83	83	–	–	–	–	–	–	–	–	74
10	101	96	93	–	–	–	–	–	–	–	–	–	80
11	110	104	–	–	–	–	–	–	–	–	–	–	89
12	121	–	–	–	–	–	–	–	–	–	–	–	99

Linear correlations for all windows of r-round

Win \ Rnd	0	1	2	3	4	5	6	7	8	9	10	11	PRINCE
1	1	1	1	1	1	1	1	1	1	1	1	1	–
2	4	4	4	4	3	4	4	4	4	4	4	–	–
3	7	7	7	6	6	7	7	7	7	7	–	–	–
4	16	14	12	10	11	16	16	16	16	–	–	–	16
5	20	20	16	16	20	22	19	19	–	–	–	–	19
6	24	23	20	22	27	25	23	–	–	–	–	–	22
7	27	26	25	30	31	29	–	–	–	–	–	–	27
8	32	34	34	34	34	–	–	–	–	–	–	–	32
9	40	42	39	39	–	–	–	–	–	–	–	–	34
10	48	45	45	–	–	–		–	–	–	–	–	38
11	51	49	–	–	–	–	–	–	–	–	–	–	41
12	55	–	–	–	–	–	–	–	–	–	–	–	49

Performance

uKNIT breaks new records for low-latency:

~ 10% **reduced latency** vs PRINCEv2

~ 20% **reduced area** vs PRINCEv2

~ 20% **increased security** (-$\log_2$ of differential probability) vs PRINCEv2

Cipher	Block Size	Latency (ns)	Area (μm^2)	Power (mW)
PRINCEv2	64	2.90	12,006.72	15.50
	64	1.65	27,564.12	26.87
SPEEDY 7	192	3.75	46,826.64	60.69
	192	1.79	88,331.04	84.53
Qarmav1 9	128	4.84	42,787.08	52.02
	128	2.74	94,944.23	87.95
uKNIT	64	2.50	9,793.08	12.09
	64	1.51	23,280.48	40.38

Hardware implementation benchmarks on TSMC 65nm

Future

- **uKNIT**: **lowest latency with good security**. Very competitive compared to the state-of-the-art
- More search can probably find a slightly better candidate, but probably not much
- Can be used as building block for larger primitives
- Our **design strategy** can be reused for other use-cases or primitives

Conclusion

Conclusion

- We will see **more automated cryptanalysis during design** phase
- Automation allows **design strategies that wouldn't be possible before**
- Performance gain is still possible in symmetric-key crypto design
- We tend to concentrate on complexity reduction to judge quality of automated cryptanalysis (i.e. $2^{20.5}$ is better than 2^{21}), but **the simplicity and ease-of-use of automated cryptanalysis is undervalued**

Thank You !

ご清聴
ありがとうございました。