On the origin of yet another channel

Intelligent brute-force with evolutionary circuit - statistical testing of output from cryptographic functions

Petr Švenda, Vashek Matyáš {svenda,matyas}@fi.muni.cz

CRoCS
Centre for Research on Cryptography and Security

Overview

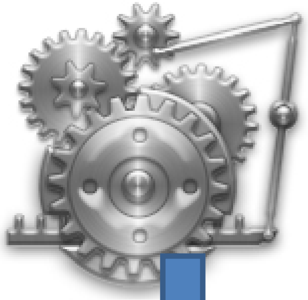
1. Randomness testing with STS NIST & Dieharder
2. Random distinguisher based on software circuit
3. Results for selected eStream/SHA-3 candidates
4. Discussion, interesting observations
5. Future extensions

Why to test randomness of function output?

1. Building block for pseudorandom generator
2. Requirement by third-party like NIST
   – AES, SHA-3 competition
3. Significant deviances from uniform distribution and unpredictability may reveal function defects
   – (but no proof otherwise)

- Manual approach: human cryptanalysis
- Automated approach: statistical testing

Workflow with STS NIST/Dieharder



1001110011…100

10^5-10^9 B

Tests

Count the 1s
Overlapping permutations
Runs tests
….

“null hypothesis” ⇒ p-values

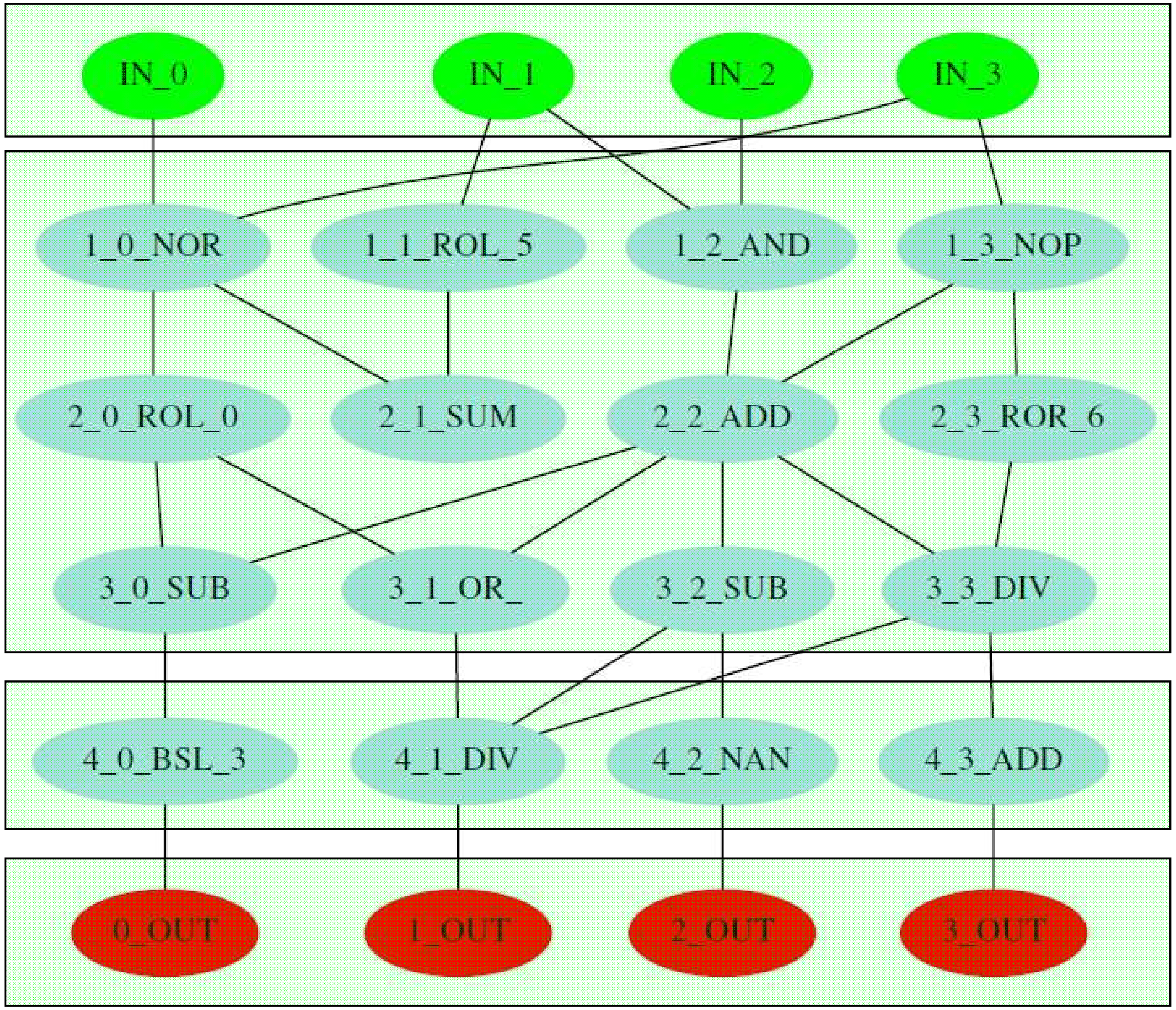
p-value < α ⇒ fail

Proposed idea – software circuit

- Design tests automatically
  - test is algorithm ⇒ hardware-like circuit
- Several issues:
  - Who will design the circuit? (genetic programming)
  - Who will define null hypothesis? (random distinguisher)
  - How to compare quality of candidates? (test vectors)

https://github.com/petrs/EACirc/

Software circuit (EACirc)



Input layer

Internal layers

Output layer

Outputs

Hypothesis: *If function output is somehow defective, circuit should be able to distinguish between the data produced by a function and truly random data.*

Quantum Random Bit Generator service

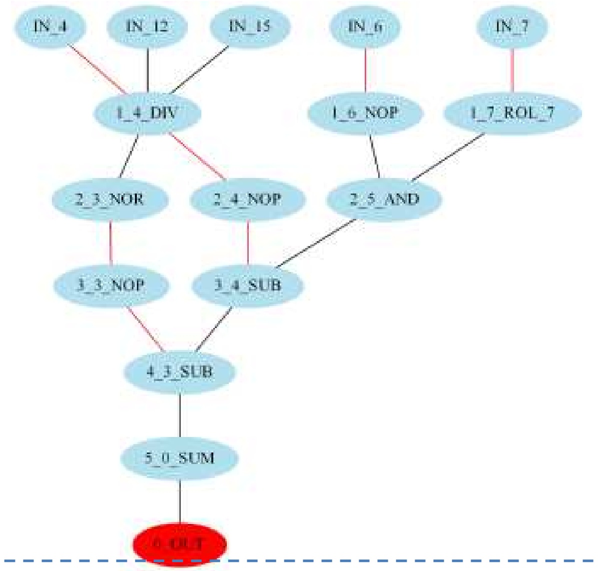
ECRYPT

500x 1011010100...101

500x 1001110011...100

Test vectors

1011010100...101



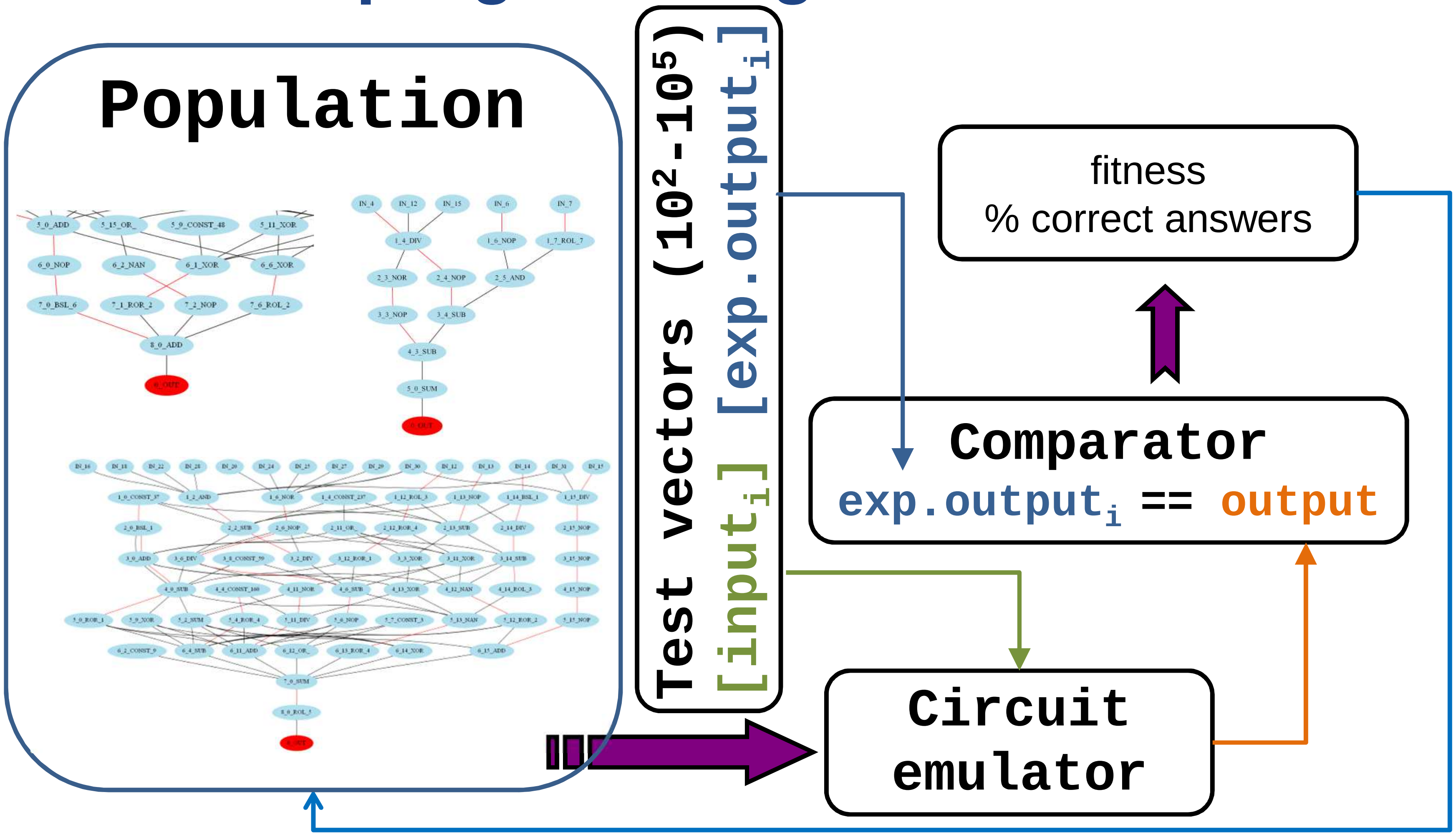
Circuit execution

10110111

`HW(10110111) > 4 => QRNG`

Fitness

Genetic programming of circuits



Circuit evaluation speed is critical

- Circuit evaluation necessary:
  - for every generation (>> 10000x)
  - for every individual in population (10-100x)
  - for every test vector (100-100000x)
  - ($6x10^8$ in our settings)
- CPU & GPU implementation developed
  - 10^6 test vectors evaluated in 3000ms (CPU@3GHz)
  - 10^6 test vectors evaluated in 150ms (CUDA@nVidia GF460)
- Used framework
  - up to 1000 CPUs @ 2.4GHz (Metacentrum grid)
  - 280 CPUs @ 3GHz (study rooms)

Methodology

- Limit number of rounds of algorithm
- Generate & run STS NIST and Dieharder tests
- Prepare input data for EACirc
  - generate ½ test vectors from function (key change freq.)
  - generate ½ test vectors from truly random source (QRBGS http://random.irb.hr/)
- Generate & test software circuits (repeat, EA)

Test vectors – key change frequency

1. Key fixed for whole run (all generations)
   - all test sets obtained from long stream generated with single key
2. Key fixed only for one test set (e.g., 500 test vectors)
3. Key per every test vector
   - one (random) key generates only one test vector with same length
   - some functions still cripple output (TSC-10, Decim-1, LEX-3)

- Test set change frequency (every 1st or 100th generation)
- Problem
  - periodicity in stream longer than NUM_VECTORS*VECT_LENGTH
  - 2.2MB / 7.8KB / 16B

Example results for Grain

#generations, 99% strong distinguisher

# of rounds	IV and key reinitialization								
	once for run			for each test set			for each test vector		
	Dieharder (x/20)	STS NIST (x/162)	EACirc	Dieharder (x/20)	STS NIST (x/162)	EACirc	Dieharder (x/20)	STS NIST (x/162)	EACirc
1	0.0	0	$n = 221$	0.0	0	(0.67)	18.5	162	(0.52)
2	0.0	0	$n = 471$	0.5	0	(0.66)	20.0	162	(0.52)
3	19.5	160	(0.52)	20.0	162	(0.52)	20.0	162	(0.52)
13	20.0	162	(0.52)	20.0	161	(0.52)	19.5	162	(0.52)

avg. success rate

Dieharder / STS NIST / EACirc (key per run)

#rounds	Decim			Grain			FUBUKI			Hermes			LEX			SALSA20			TSC		
min-max	1-8			1-5			1-32			1-2			1-13			1-20			1-32		
1	*	*	*	*	*	*	-	-	-	-	-	-	*	*	*	*	*	*	-	-	-
2	*	*	*	*	*	*	-	-	-	-	-	-	*	*	*	*	*	*	-	-	-
3	*	*	*	-	-	-	-	-	-	-	-	-	*	*	*	-	-	-	-	-	-
4	*	*	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
5	*	*	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
6	*	*	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
7	*	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
9	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*	*	*
10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*	*	*
11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*	*	-
12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*	-	-

Dieharder / STS NIST / EACirc (key per set)

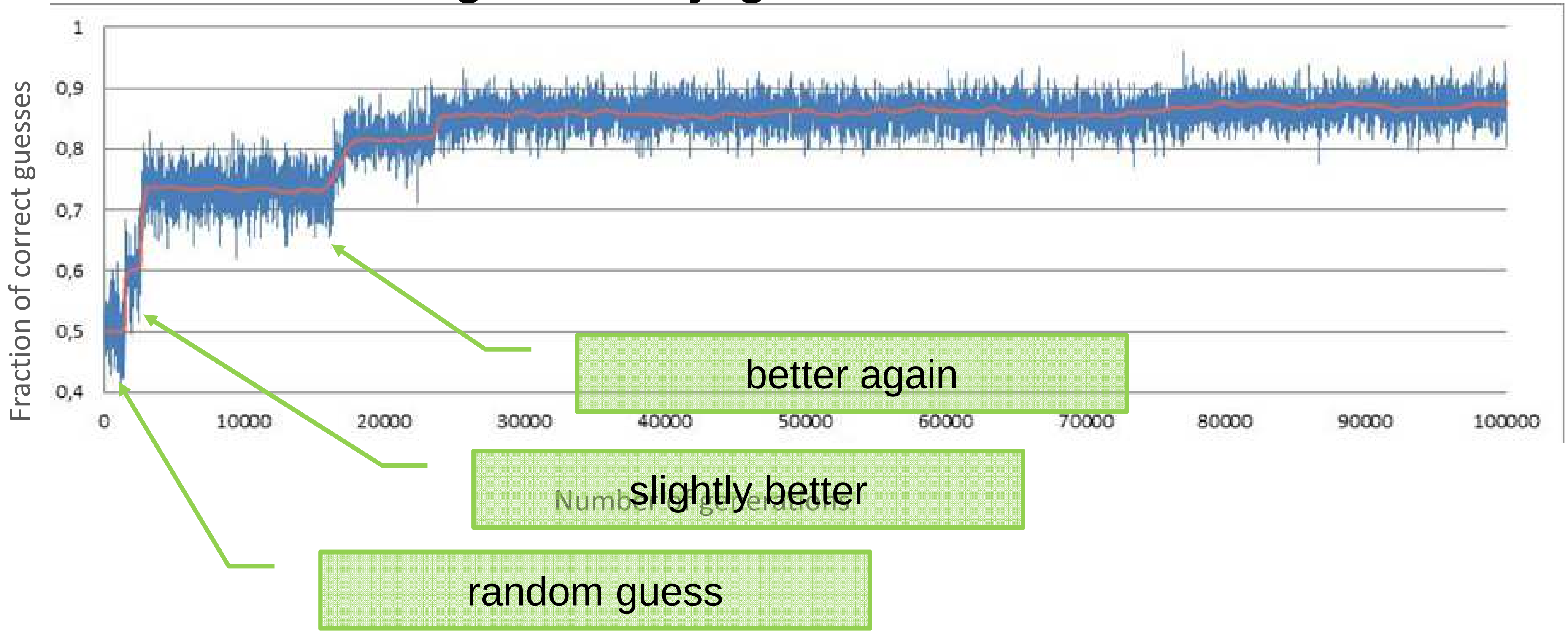
#rounds	Decim			Grain			FUBUKI			Hermes			LEX			SALSA20			TSC		
min-max	1-8			1-5			1-32			1-2			1-13			1-20			1-32		
1	*	*	*	*	*	*	-	-	-	-	-	-	*	*	*	*	*	*	-	-	-
2	*	*	*	*	*	*	-	-	-	-	-	-	*	*	*	*	*	*	-	-	-
3	*	*	*	-	-	-	-	-	-	-	-	-	*	*	*	-	-	-	-	-	-
4	*	*	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
5	*	*	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
6	*	*	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
7	*	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
9	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*	*	*
10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*	*	*
11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*	*	-
12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*	-	-

Dieharder/STS NIST/EACirc (key per vector)

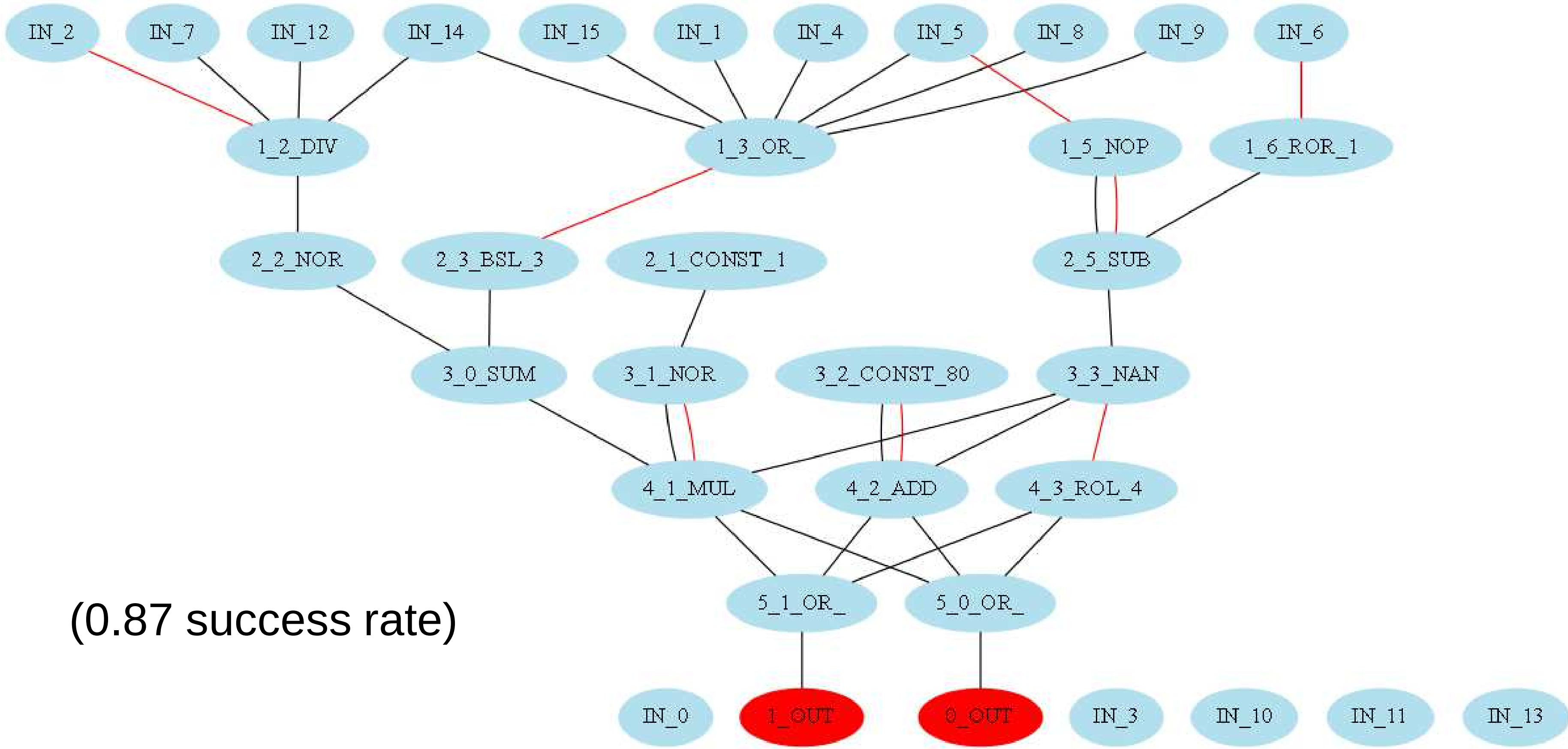
#rounds	Decim			Grain			FUBUKI			Hermes			LEX			SALSA20			TSC		
min-max	1-8			1-5			1-32			1-2			1-13			1-20			1-32		
1	*	*	*	*	*	*	-	-	-	-	-	-	*	*	*	*	*	*	-	-	-
2	*	*	*	*	*	*	-	-	-	-	-	-	*	*	*	*	*	*	-	-	-
3	*	*	*	-	-	-	-	-	-	-	-	-	*	*	*	-	-	-	-	-	-
4	*	*	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
5	*	*	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
6	*	*	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
7	*	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
9	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*	*	*
10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*	*	*
11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*	*	-
12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*	-	-

Learning on real structure (Dynamic-SHA)

- Test set changed every generation

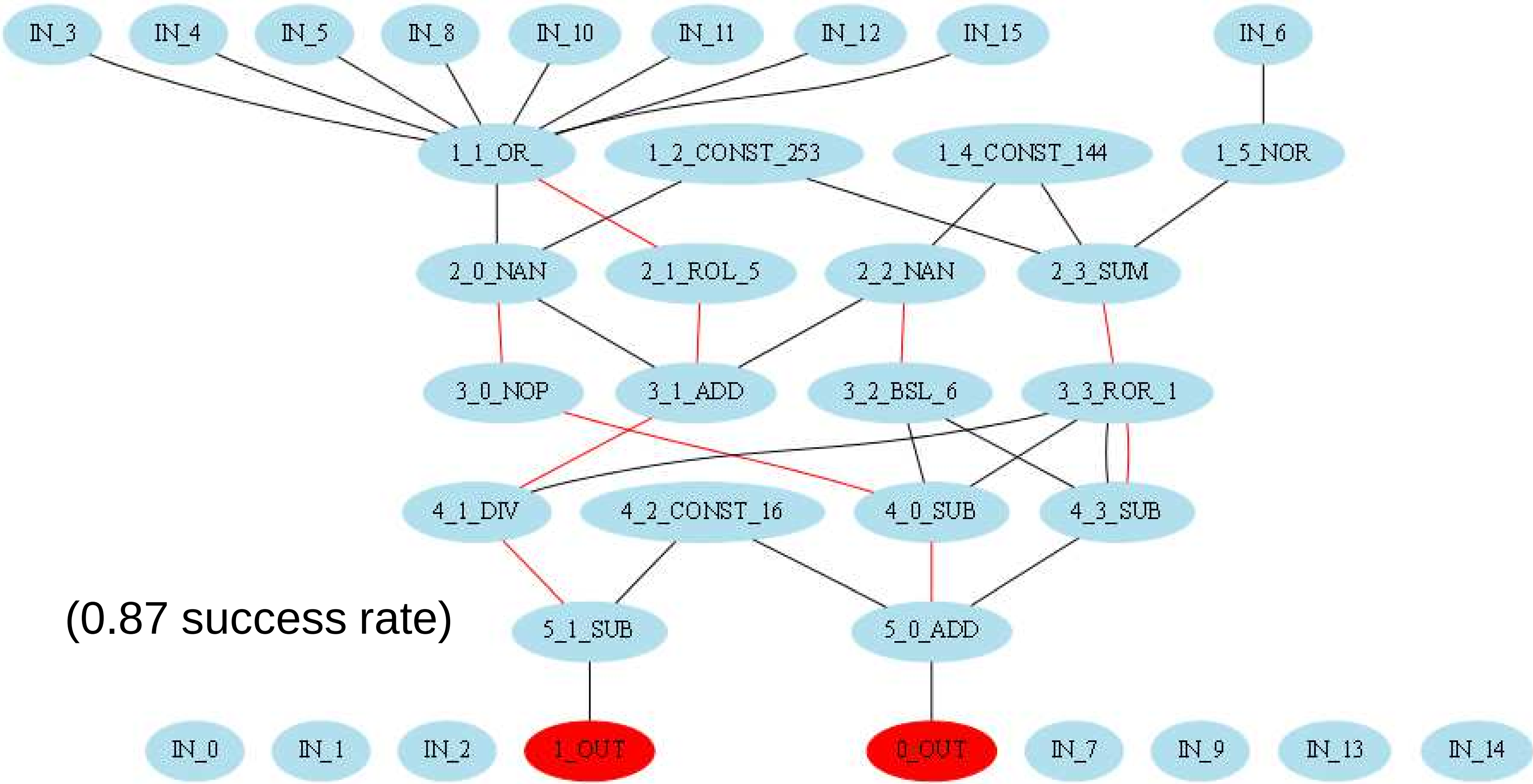


Salsa20 – limited to two rounds (case 1)



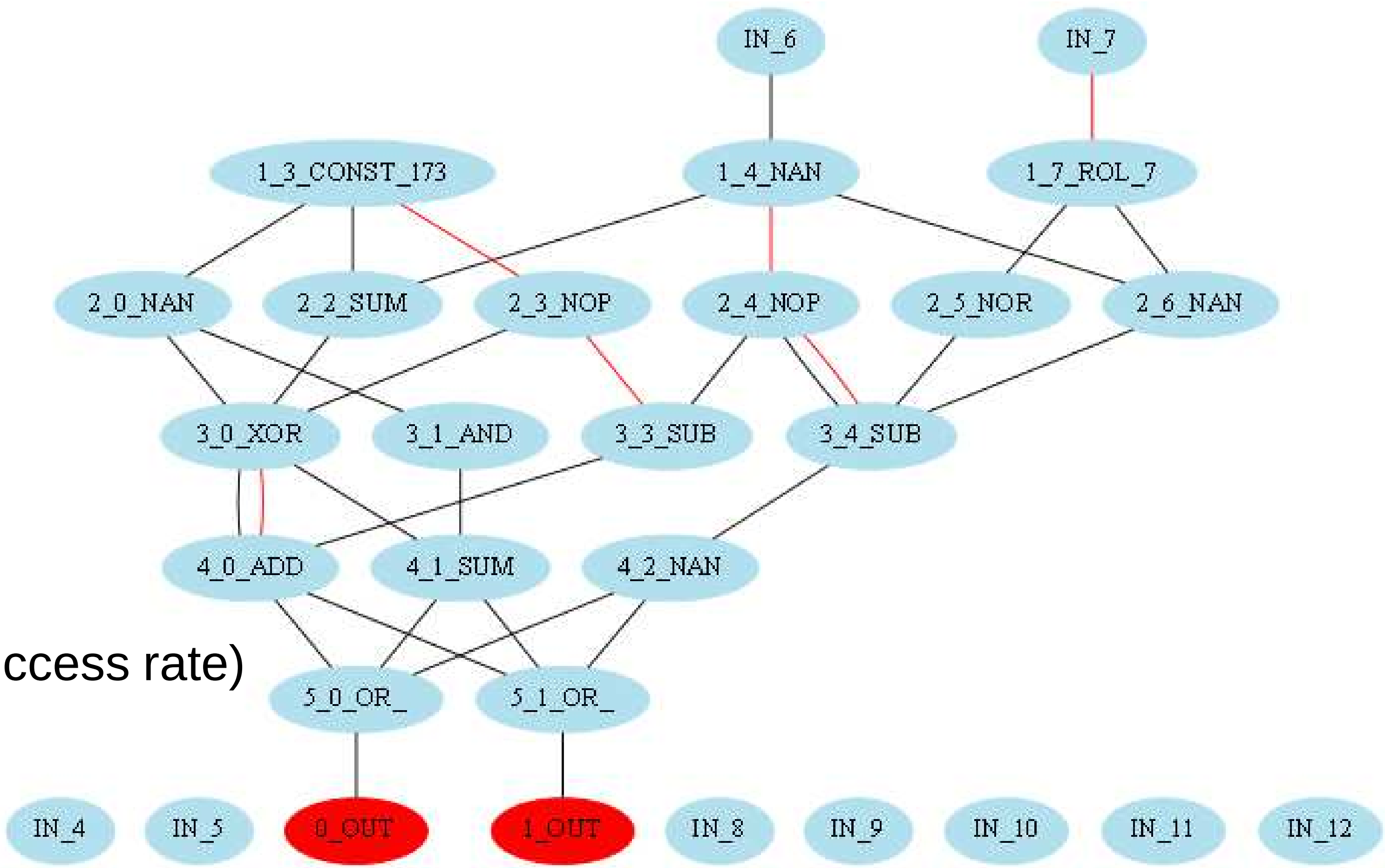
(0.87 success rate)

Salsa20 – limited to two rounds (case 2)



(0.87 success rate)

Salsa20 – limited to two rounds (case 3)



(0.87 success rate)

Salsa20 – limited to two rounds (case 2)

```
static void circuit(unsigned char inputs[MAX_INPUTS], unsigned char outputs[MAX_OUTPUTS]) {
    unsigned char VAR_IN_0 = inputs[0];
    unsigned char VAR_IN_1 = inputs[1];
    unsigned char VAR_IN_2 = inputs[2];
    unsigned char VAR_IN_3 = inputs[3];
    unsigned char VAR_IN_4 = inputs[4];
    unsigned char VAR_IN_5 = inputs[5];
    unsigned char VAR_IN_6 = inputs[6];
    unsigned char VAR_IN_7 = inputs[7];
    unsigned char VAR_IN_8 = inputs[8];
    unsigned char VAR_IN_9 = inputs[9];
    unsigned char VAR_IN_10 = inputs[10];
    unsigned char VAR_IN_11 = inputs[11];
    unsigned char VAR_IN_12 = inputs[12];
    unsigned char VAR_IN_13 = inputs[13];
    unsigned char VAR_IN_14 = inputs[14];
    unsigned char VAR_IN_15 = inputs[15];

    unsigned char VAR_1_1_OR_ = VAR_IN_3 | VAR_IN_4 | VAR_IN_5 | VAR_IN_8 | VAR_IN_10 | VAR_IN_11 | \
                                VAR_IN_12 | VAR_IN_15 | 0;
    unsigned char VAR_1_2_CONST_253 = 253 ;
    unsigned char VAR_1_4_CONST_144 = 144 ;
    unsigned char VAR_1_5_NOR = 0 | ~ VAR_IN_6 | ~ 0xff;
    unsigned char VAR_2_0_NAN = 0xff & ~  VAR_1_1_OR_ & ~ VAR_1_2_CONST_253 & ~ 0;
    unsigned char VAR_2_1_ROL_5 = VAR_1_1_OR_ << 5 ;
    unsigned char VAR_2_2_NAN = 0xff & ~  VAR_1_4_CONST_144 & ~ 0;
    unsigned char VAR_2_3_SUM = VAR_1_2_CONST_253 + VAR_1_4_CONST_144 + VAR_1_5_NOR + 0;
    unsigned char VAR_3_0_NOP = VAR_2_0_NAN ;
    unsigned char VAR_3_1_ADD = VAR_2_1_ROL_5 +  VAR_2_0_NAN + VAR_2_2_NAN + 0;
    unsigned char VAR_3_2_BSL_6 = VAR_2_2_NAN & 6 ;
    unsigned char VAR_3_3_ROR_1 = VAR_2_3_SUM >> 1 ;
    unsigned char VAR_4_0_SUB = VAR_3_0_NOP -  VAR_3_2_BSL_6 - VAR_3_3_ROR_1 - 0;
    unsigned char VAR_4_1_DIV = VAR_3_1_ADD /  ((VAR_3_3_ROR_1 != 0) ? VAR_3_3_ROR_1 : 1) / 1;
    unsigned char VAR_4_2_CONST_16 = 16 ;
    unsigned char VAR_4_3_SUB = VAR_3_3_ROR_1 -  VAR_3_2_BSL_6 - VAR_3_3_ROR_1 - 0;
    unsigned char VAR_5_0_ADD = VAR_4_0_SUB +  VAR_4_2_CONST_16 + VAR_4_3_SUB + 0;
    unsigned char VAR_5_1_SUB = VAR_4_1_DIV -  VAR_4_2_CONST_16 - 0;

    outputs[0] = VAR_5_0_ADD;
    outputs[1] = VAR_5_1_SUB;
}
```

Taking inputs (annotation for the VAR_IN_0 … VAR_IN_15 lines)

Inner layers (annotation for the VAR_1_1_OR_ … VAR_5_1_SUB lines)

Producing outputs (annotation for the outputs lines)

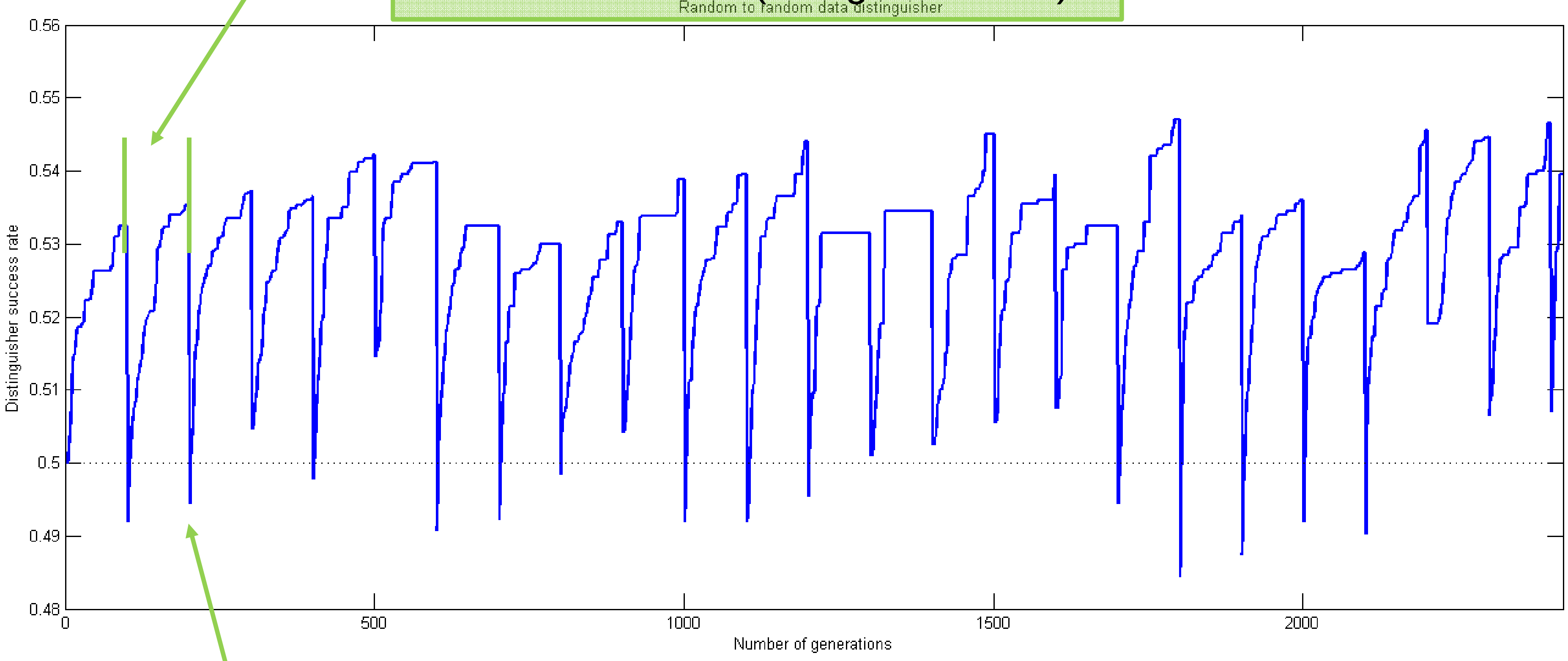
So what is the resulting test for battery?

- One particular circuit?
  - circuit was evolved for particular function and key
  - sometimes, circuit works even when key is changed
  - (most probably) not useful for different function

- Whole process with evolution of circuits is the test!
  - Is evolution able to design distinguisher in limited number of generations?
  - If yes, then function output is defect

Overlearning only (random vs. random)

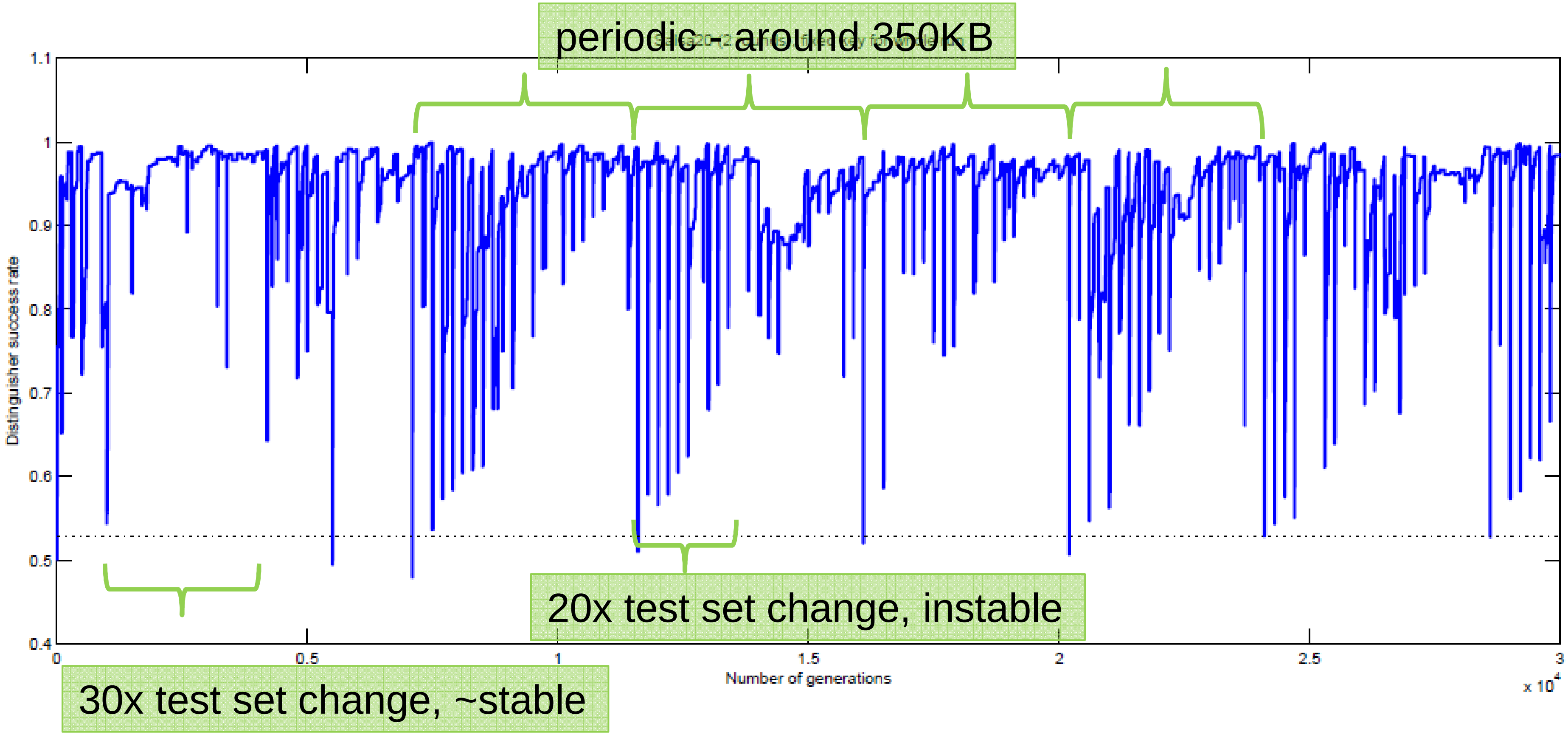
same test vector (100 generations)



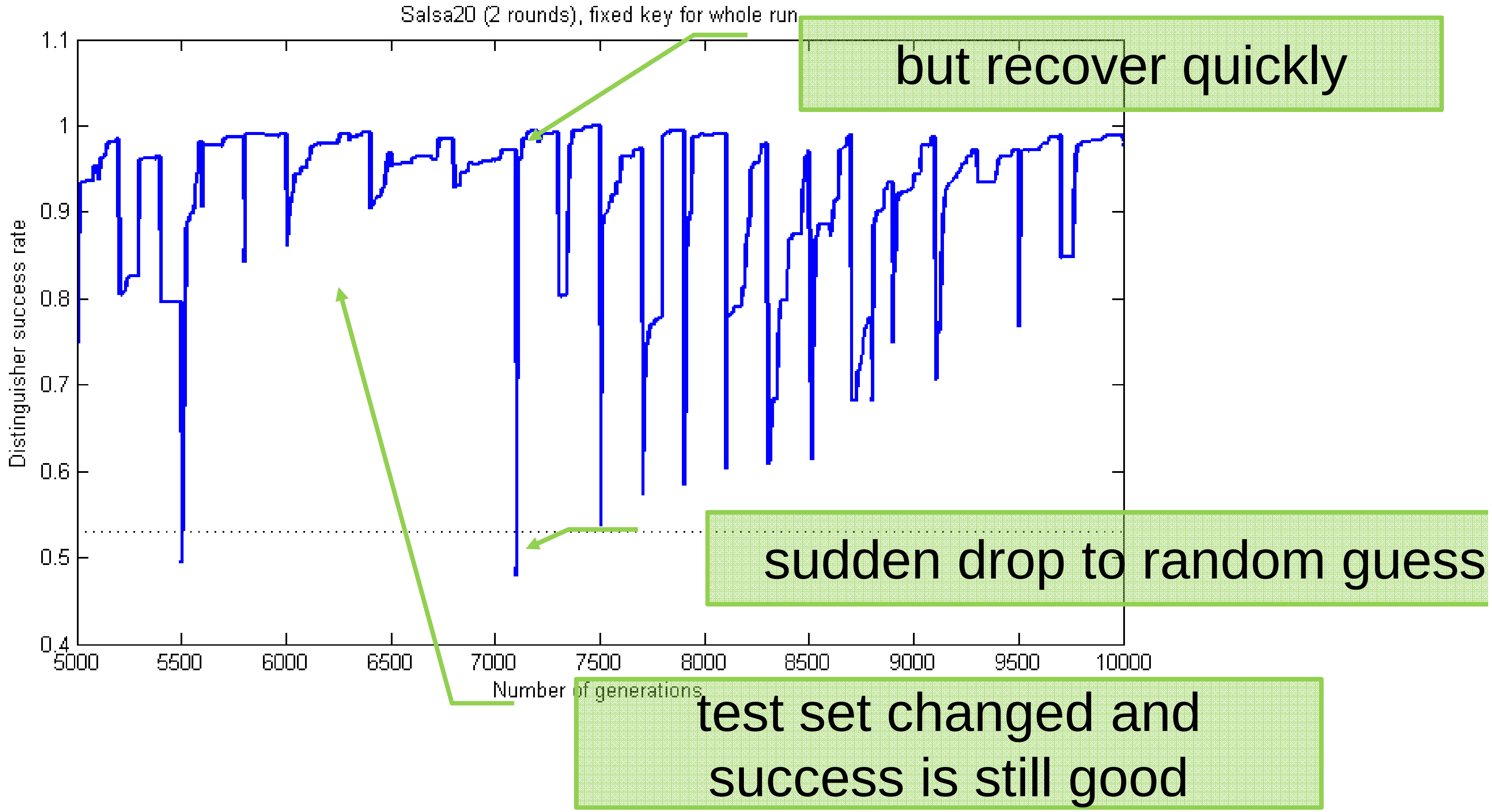
test vector change (sudden drop in fitness)

Learning speed as fitness (Salsa20-2)

- 30 000 generations ⇒ 300 changes of test set



Learning speed – zoom (Salsa20-2)



Comparison to statistical batteries

- Advantages
  - new approach, no need for predefined pattern
  - dynamic construction of test for particular function
  - works on very short sequences (16 bytes only)
- Disadvantages
  - no proof of test quality or coverage (random search)
  - possibly hard to analyze the result
  - possibly longer test run time (learning period)

On fairness of comparison

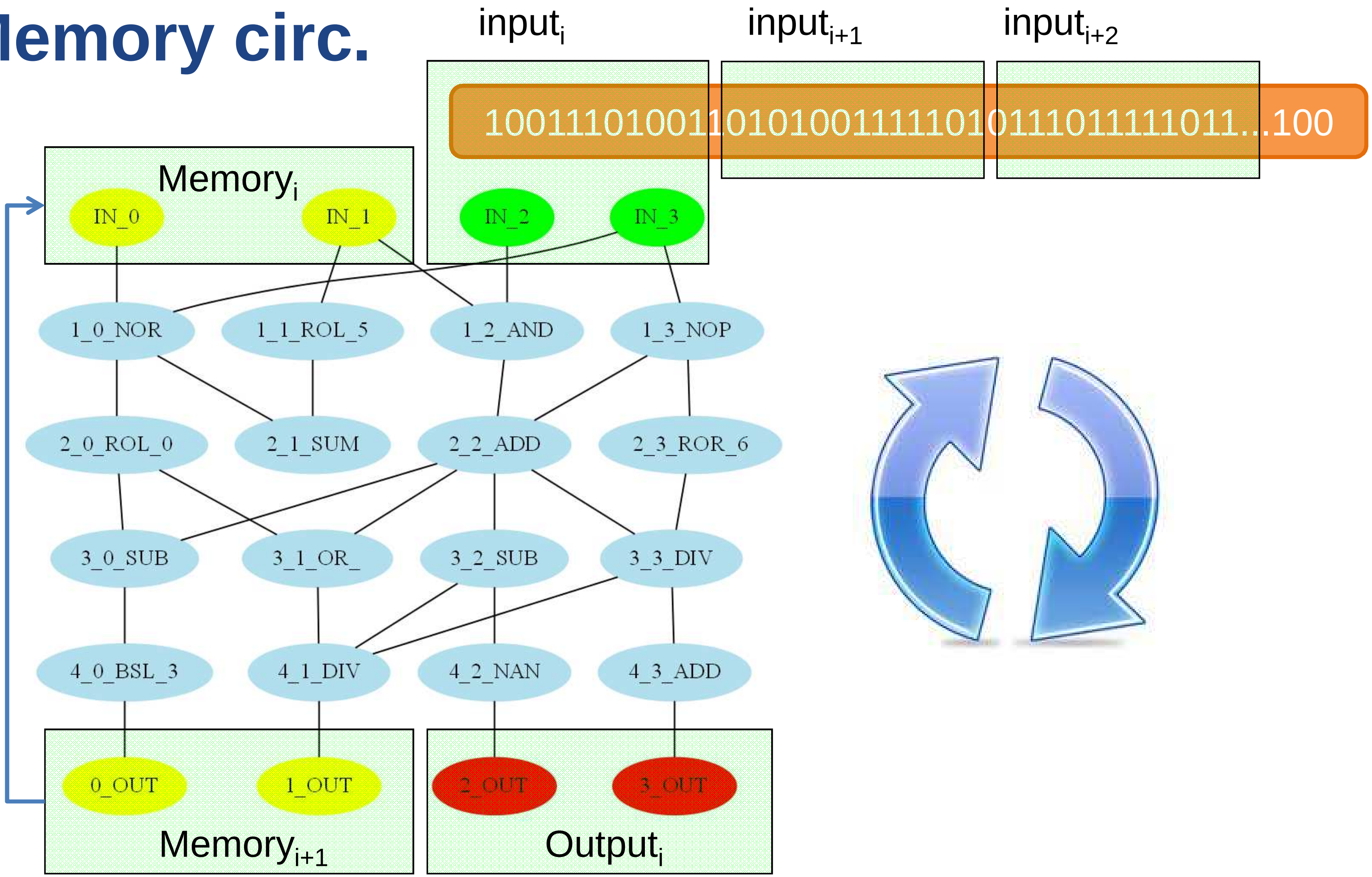
- STS NIST & Dieharder sometimes better so far
  - (key per run, for some functions only)
  - advantage decrease as key frequency exchange increases
  - shorter data produced with same key is available
- But...
  - Dieharder requires up to 200MB of data
  - STS NIST recommends 12MB (100x1000000bits)
- EACirc requires:
  - only 16B for testing
  - 2.2MB for learning (if 30k generations)
- Next step – How to supply more data to circuit?

Future work and extensions

- So far, we focused on broader rather than deeper testing
  – more functions, but less generations and optimizations
  – verification of results (static circuit instead of emulation)

1. Longer evolution, more layers run may help
   – higher number of generations, optimizations (diversity)
2. Make longer data available to circuit
   – circuit with memory (next slide)
3. Allow for more complex computation into node
   – linear genetic programming for every node, code fragments

Memory circ.



$input_i$
$input_{i+1}$
$input_{i+2}$
1001110100110101001111101011101111011...100
$Memory_i$
IN_0
IN_1
IN_2
IN_3
1_0_NOR
1_1_ROL_5
1_2_AND
1_3_NOP
2_0_ROL_0
2_1_SUM
2_2_ADD
2_3_ROR_6
3_0_SUB
3_1_OR_
3_2_SUB
3_3_DIV
4_0_BSL_3
4_1_DIV
4_2_NAN
4_3_ADD
0_OUT
1_OUT
2_OUT
3_OUT
$Memory_{i+1}$
$Output_i$

More instruction in single node

- So far, only simple operations used (SUM, DIV...)
- Small program can be executed inside node
  1. sequence of simple operations (non-branching), LGP
  2. code extracted from function’s (Java) implementation
     - emulation of disassembled bytecode
- Stack-based execution assumed
  - input argument given by connector(s) from previous layer
  - instructions and length set by evolution
  - top of the stack as node’s output value
- All still automatic (LGP, disassembling)

Other goals than random distinguisher

- Strict avalanche criterion
- Next bit predictor
- Application to only subpart of function
- ...

Conclusions

- Genetic programming for random distinguisher
- Comparable results to STS NIST
  - lacking with longer sequences
- More detailed analysis of results needed
  - comparison of multiple circuits for same settings
  - weakness detected
- Make more data available to circuit
  - circuit with memory

Questions

Thank you for your attention!

Questions