Evolutionary circuit

Petr Švenda
xsvenda@fi.muni.cz

Labak&BUSLab, FI MU Brno

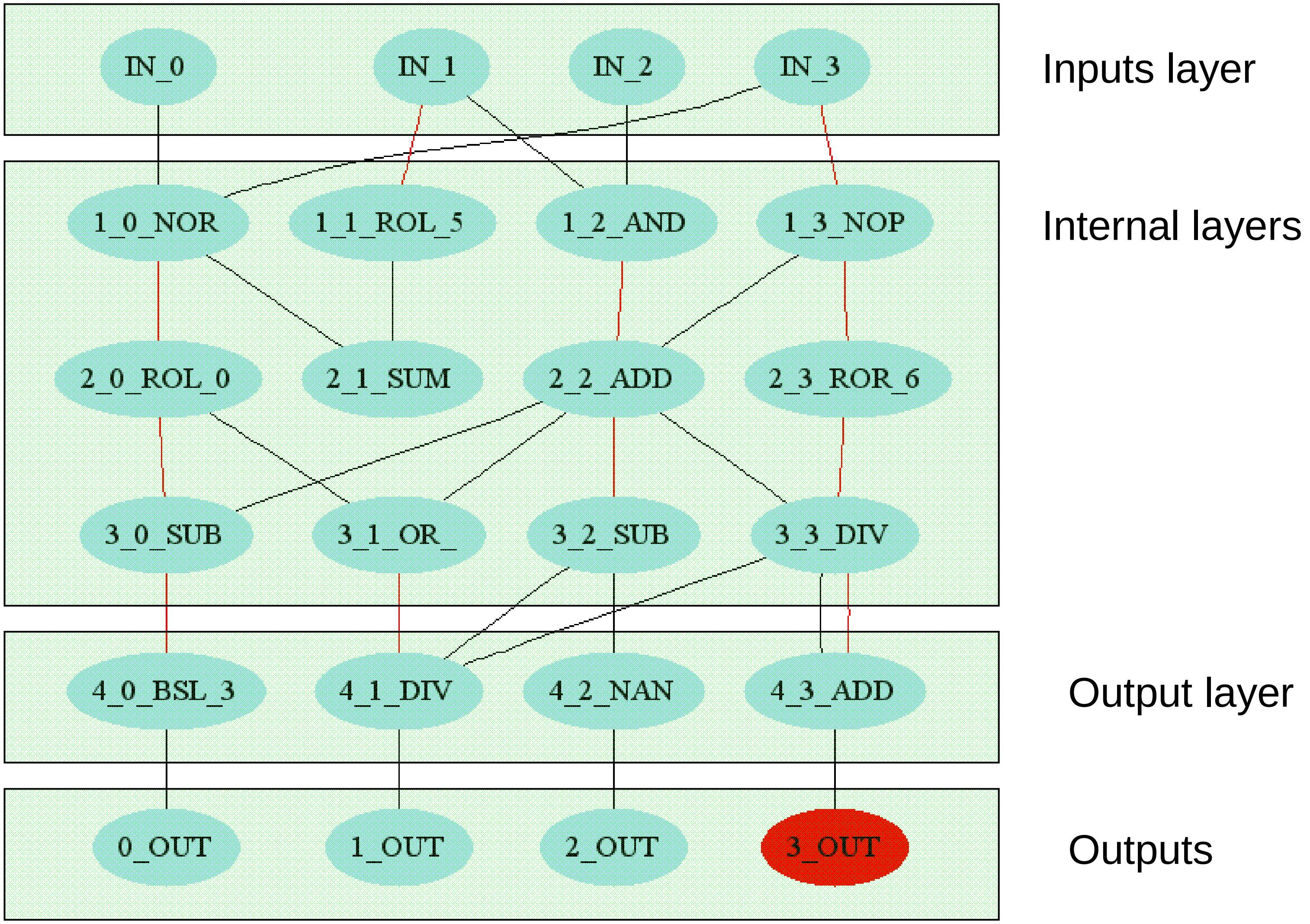
Overview

- Design of software circuits
- Evolutionary Algorithms
- Combination with Evolutionary Algorithms
- Visualization
- Practical applications
- Practical experience
- Open issues

Evolutionary circuits (EAC)

- Distinct layers of elementary functions
- Usually same number of functions in each (internal) layer
  - with exception of first “input” and last “output” layer
- Layers are interconnected by “wires”
  - typically 2 connectors in hardware
  - up to full interconnection with previous layer in software
    - each function can obtain whole input from previous layer
- Input data are set as input for first layer
- Output of internal layer is set as input data for next layer
- Output data from last internal layer are output of circuit

Circuit example (4 layers, 4 fncs in layer)



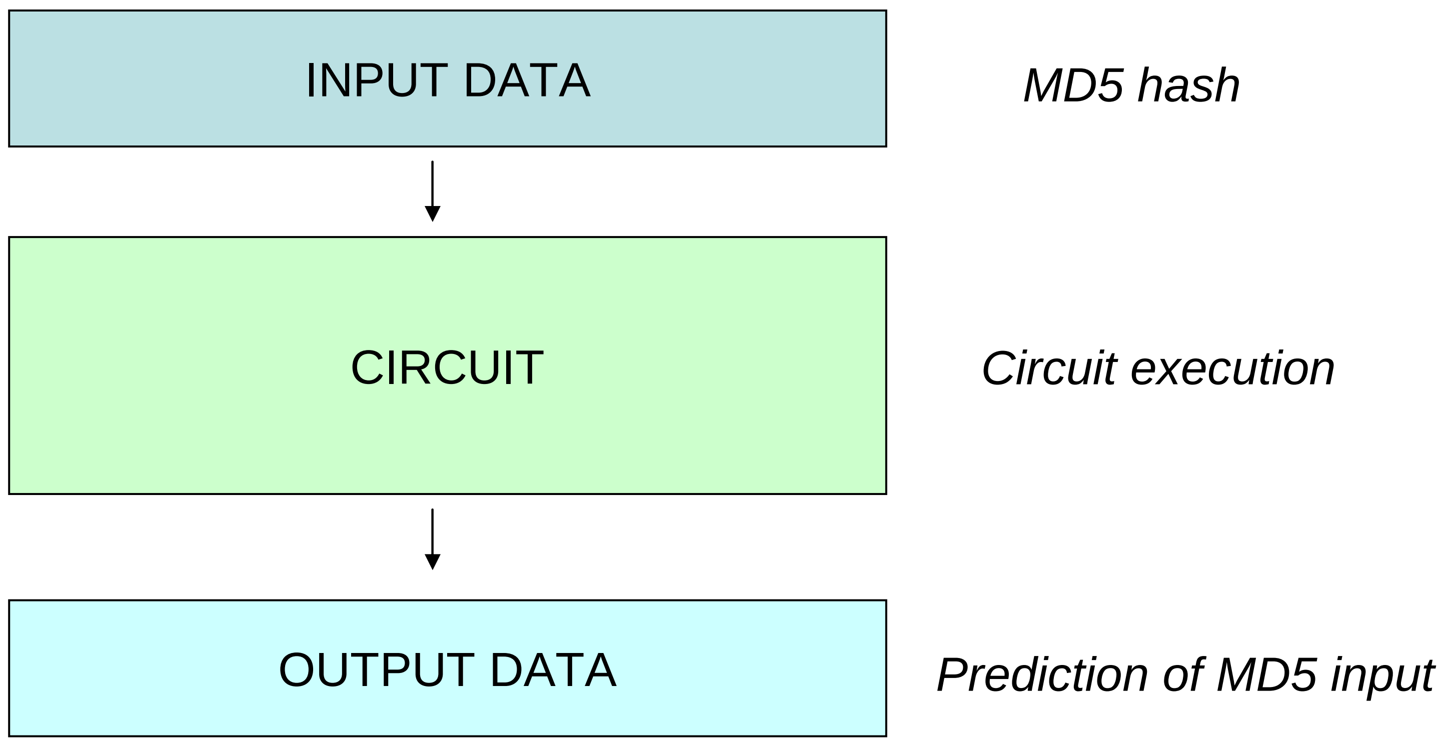
Node functions/connectors

- Special purpose functions
  - NOP, CONST
- Bit oriented functions
  - AND, OR, XOR, NOR, NAND, ROTL, ROTR, BITSELECTOR
- Byte/dword oriented functions
  - SUM, SUBS, ADD, MULT, DIV
- Input of the function is given by connector mask
  - 4 bytes mask => max. 32 inputs (can be limited by settings)
    - connections are independent from input data (static circuit)
  - special variable connection mask
    - final connection mask is computed from selection of actual inputs
    - (part of) connectors inside circuit change with input data (dynamic circuit)
- Some functions have one input fixed (same offset, previous layer)
  - BITSELECTOR – fixed input, used to mask out specific bits

Data processing details

- Base operation value
  - bit, byte, dword
  - size of operand passing through connection “wire”
- Suitable operand type depends on the problem
  - e.g., if the solution should be function working with bytes
    - using bits still may work but is harder to evolve (1B xor = 8x1b xor)
  - but sometimes we simply don’t know
    - is MD5 inverse function bit or byte oriented?
- So far, we were working only with byte-oriented circuit
  - but bit selection is in instruction set
    - theoretically possible for evolution to work on bit level anyway
    - but may be much more difficult

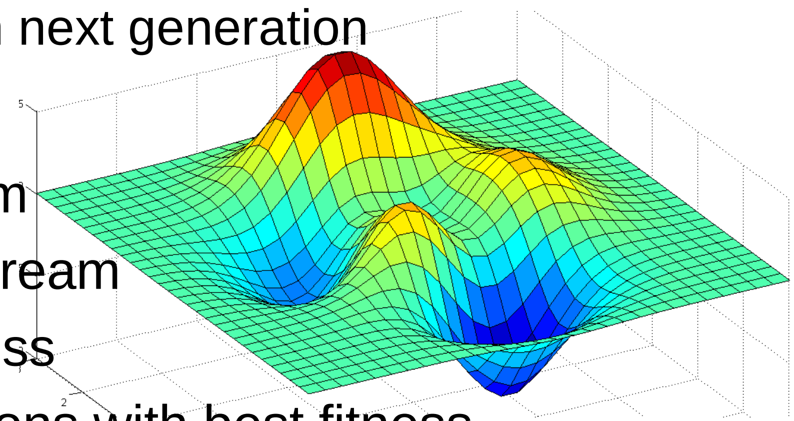
Basic circuit execution



- So we have the tool, how to find the proper circuit now?

Method solving inspired by evolution

- Charles Darwin - *On the Origin of Species* (1859)
- Necessary prerequisite for evolution to work:
  - elementary units – genes
  - possibility to reproduction - copy itself with reasonable quality
  - possibility to mutation - new information can be introduced
  - natural selection – “better” specimen has more offspring
    - and its genes will be more often in next generation
- Evolutionary algorithms
  - “clever” search for function maximum
  - candidate solution encoded as bit stream
  - algorithmic function to evaluate fitness
  - next generation selected from solutions with best fitness



Types of evolutionary algorithms

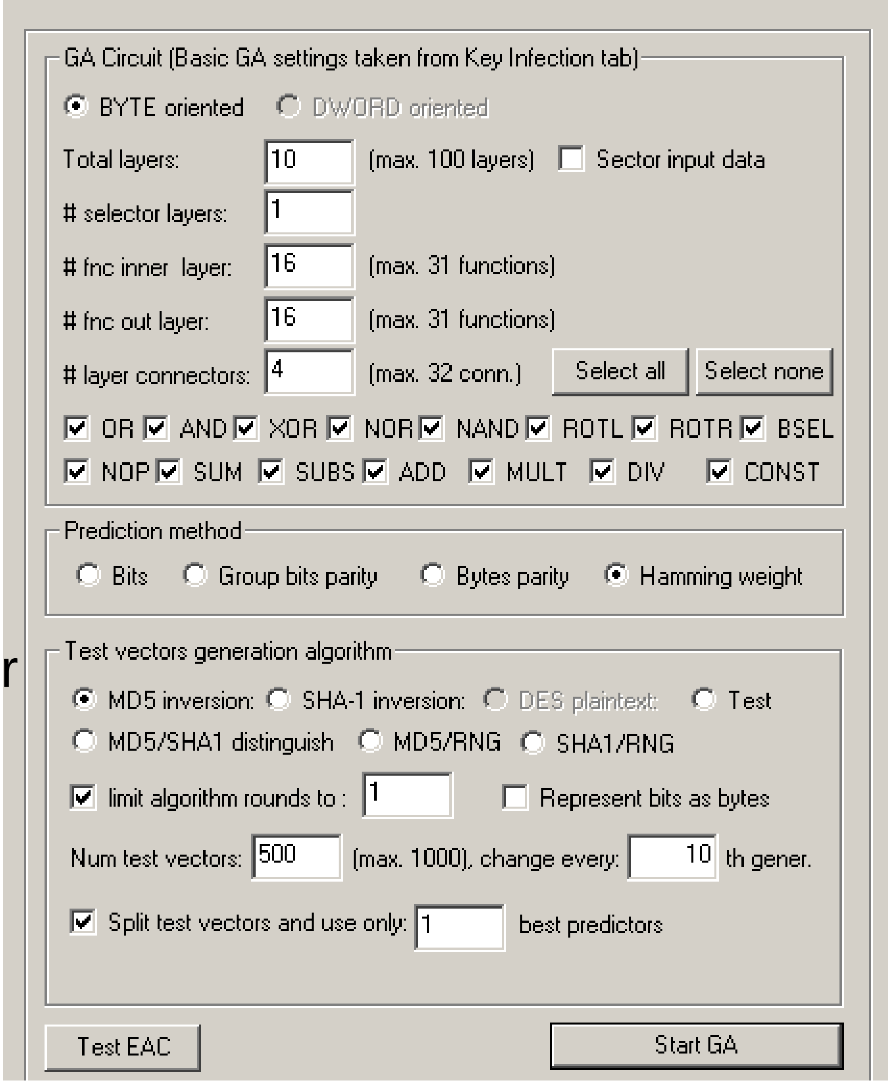
- Basic idea is still same mutation/crossing/fitness
- Representation of solution may differ
  - genetic algorithms: set of variables
  - genetic programming: LISP-like trees
  - linear genetic programming: actual instructions of “program”
  - evolutionary circuits: hardware-like circuits with basic functions
- Previous work with secrecy amplification protocols was LGP
  - sequence of instructions of the protocol
- Here we focus on software version of circuits
  - software emulation of circuit

Combination with Evolutionary algorithms

1. Create “population” of several (e.g., 20) circuits
2. Initialize functions and connections at random
3. Generate random test vectors {input, correct_output}
   - e.g., {MD5($data_X$), $data_X$}, … {MD5($data_Y$), $data_Y$},
4. Evaluate population
   - compute degree of match between circuit output and correct_output
   - different match metrics methods
     - bit, parity, hamming weight, selected group, …
5. Select best performing circuits from actual generation and form new
   - mutation (flip of single connection, change of function)
   - crossover (half of layers from first and second parent)
6. Repeat again from step 4.
   - sometimes (e.g., each 10^{th} generation) from step 3. – new test vectors

EAC implementation, C++

- Circuit parameters
  - num layers/fncs/connectors
  - allowed functions
- Prediction methods
  - bits, parity, hamming weight
- Test vectors
  - MD5/SHA1 inversion
  - MD5/SHA1 random distinguisher
  - change frequency
- Limit algorithm rounds
  - weakened algorithms

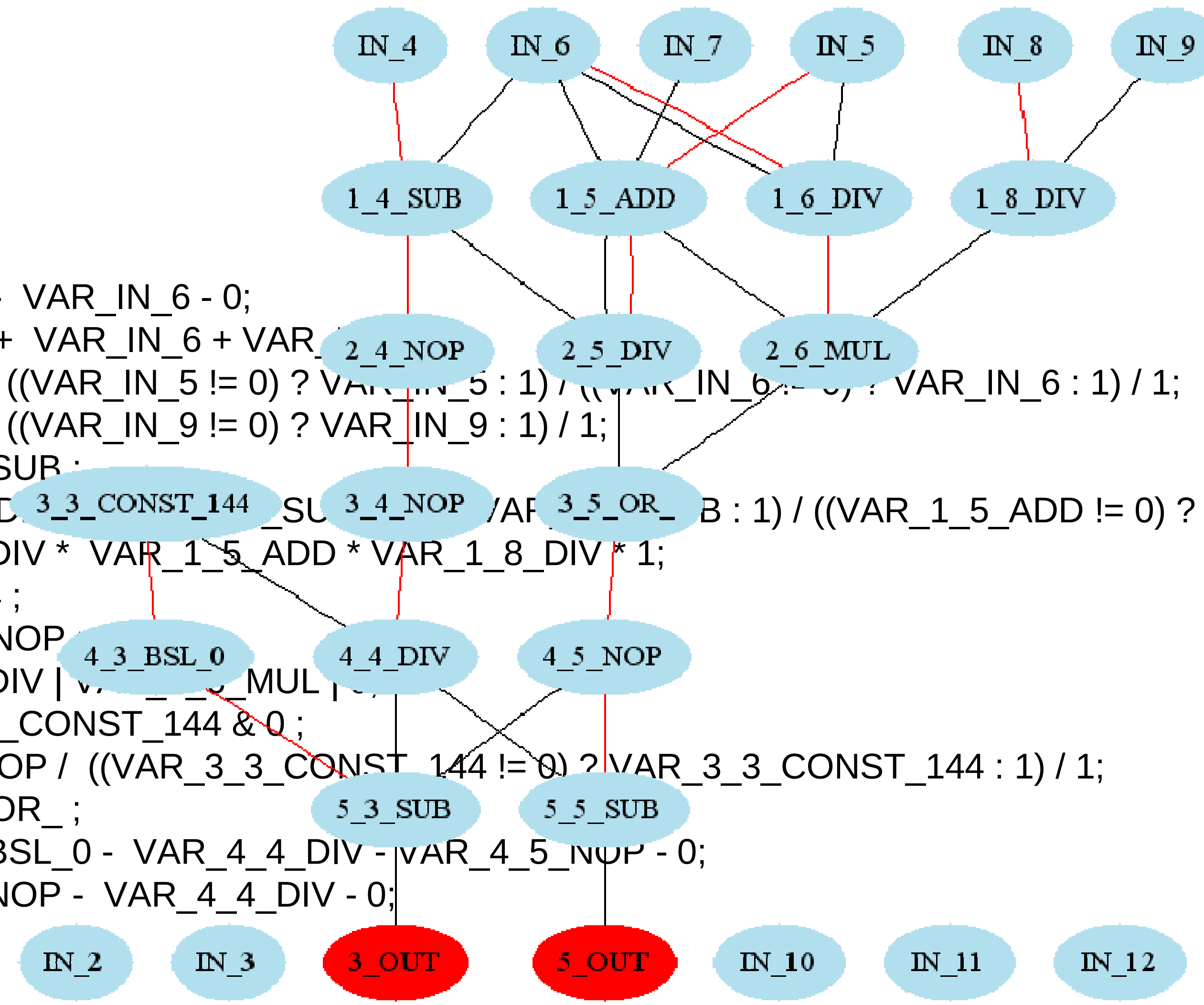


Visualization/code export

- Very important part of the process!
  - human readable form of circuit
  - allow to human check of correctness
  - helps to discover program bugs
- Circuit pruning
  - temporarily disable connection or function
  - if fitness decrease then connection/function was important
  - display only important parts of circuit (usually only around 10%)
- Visualization
  - Graphviz package
  - source script generated automatically during evolution
- Source code generation
  - automated export of compile-able circuit C source code

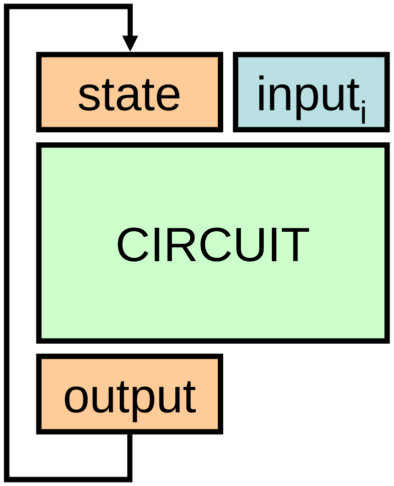
Visualization / source code export

```
BYTE VAR_IN_5 = inputs[5];
BYTE VAR_IN_6 = inputs[6];
BYTE VAR_IN_7 = inputs[7];
BYTE VAR_IN_8 = inputs[8];
BYTE VAR_IN_9 = inputs[9];

BYTE VAR_1_4_SUB = VAR_IN_4 -  VAR_IN_6 - 0;
BYTE VAR_1_5_ADD = VAR_IN_5 +  VAR_IN_6 + VAR_…
BYTE VAR_1_6_DIV = VAR_IN_6 /  ((VAR_IN_5 != 0) ? VAR_IN_5 : 1) / ((VAR_IN_6 != 0) ? VAR_IN_6 : 1) / 1;
BYTE VAR_1_8_DIV = VAR_IN_8 /  ((VAR_IN_9 != 0) ? VAR_IN_9 : 1) / 1;
BYTE VAR_2_4_NOP = VAR_1_4_SUB ;
BYTE VAR_2_5_DIV = VAR_1_5_AD… B : 1) / ((VAR_1_5_ADD != 0) ? V
BYTE VAR_2_6_MUL = VAR_1_6_DIV *  VAR_1_5_ADD * VAR_1_8_DIV * 1;
BYTE VAR_3_3_CONST_144 = 144 ;
BYTE VAR_3_4_NOP = VAR_2_4_NOP ;
BYTE VAR_3_5_OR_ = VAR_2_5_DIV | VAR_2_6_MUL | 0;
BYTE VAR_4_3_BSL_0 = VAR_3_3_CONST_144 & 0 ;
BYTE VAR_4_4_DIV = VAR_3_4_NOP /  ((VAR_3_3_CONST_144 != 0) ? VAR_3_3_CONST_144 : 1) / 1;
BYTE VAR_4_5_NOP = VAR_3_5_OR_ ;
BYTE VAR_5_3_SUB = VAR_4_3_BSL_0 -  VAR_4_4_DIV - VAR_4_5_NOP - 0;
BYTE VAR_5_5_SUB = VAR_4_5_NOP -  VAR_4_4_DIV - 0;

outputs[3] = VAR_5_3_SUB;
outputs[5] = VAR_5_5_SUB;
```



EAC variations

- Static circuits
  - all connections between layers are independent from input data
  - mask of connections is fixed during evaluation
- Dynamic circuits
  - special type of connection mask
  - actual connection mask is taken from output of previous layer (input data dependent)
- Circuits with state
  - suitable for problems with large input data
  - multiple passes of EAC
  - not all input data are given to circuit at the same time
  - circuit output is “state”
  - state is passed as part of circuit input for next pass



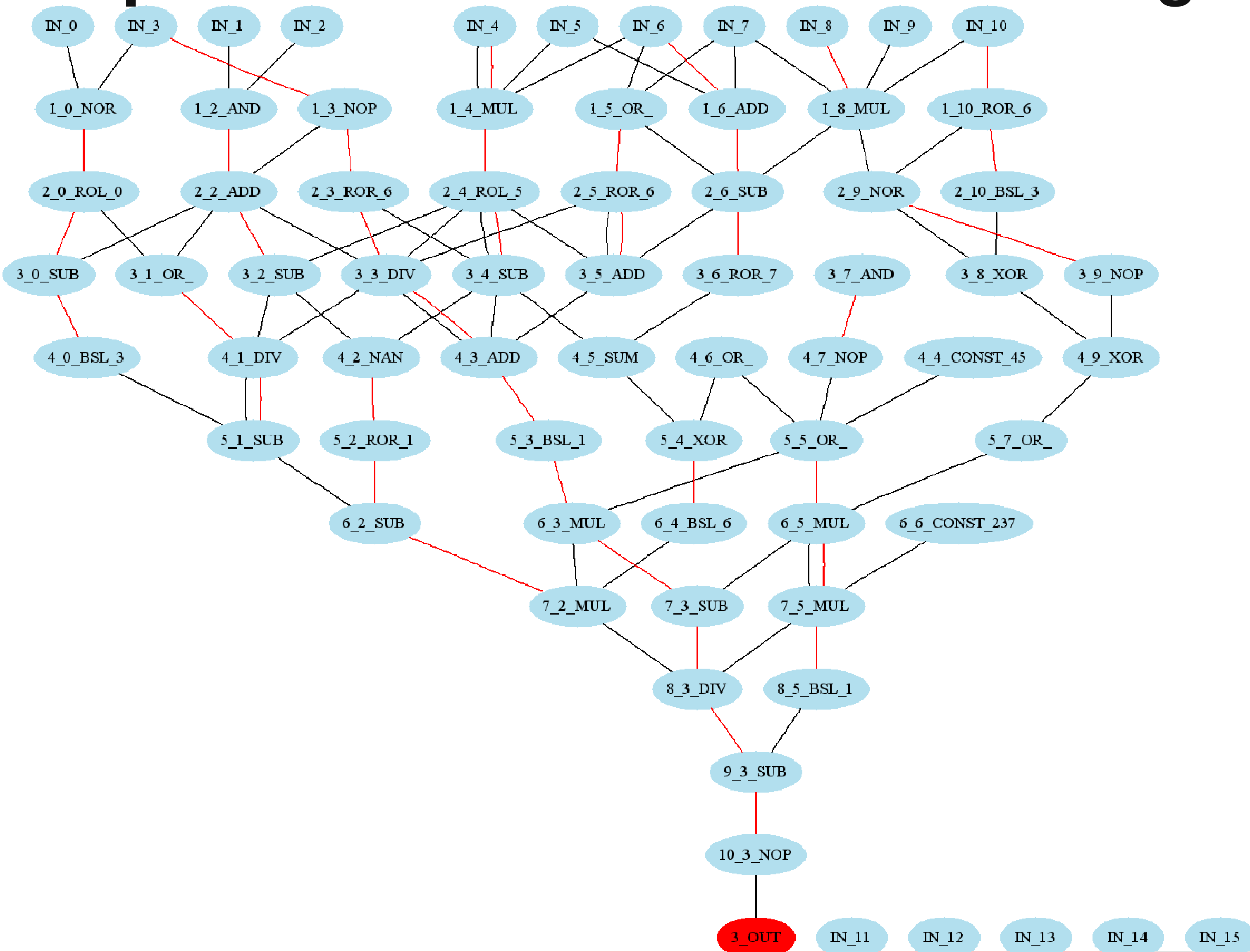
Practical results - hash function inversion

- We are at very early stage
  - most work on circuit development so far
- Hash function inversion
  - prediction of bits / hamming weight of input
  - a = x bytes random input, b = hash(a)
  - circuit obtains b and outputs prediction of a
  - tested on MD5
- Best results so far
  - hamming weight of input for 4-round MD5 (from 64)
  - circuit: 10 layers, 4 connectors

Practical results – random distinguisher

- Random stream distinguisher
  - circuit try to differentiate between completely random stream and stream generated by target function with unknown input
  - QRGBS http://random.irb.hr/index.php
  - input data are either random stream or hash of structured data
    - two random bytes repeated to form 16B input
  - output data is 0x00 for hash function, 0xff for random stream
  - tested on MD5 and SHA1
- Best results so far
  - around 68% success of distinguishing for 10-round MD5 (from 64)
  - around 70% success of distinguishing for 8-round SHA1 (from 80)
  - circuit: 10 layers, 4 connectors

Example: 10 rounds MD5/RNG distinguisher



Future work

- Sequence prediction
  - prediction of (some) next bits from given sequence
  - unpredictability import for pseudo-random generators
  - similar usage also for stream cipher sequence prediction
    - ECRYPT eSTREAM candidates
- Internal secrets prediction
  - function with known input and output and secret internal state
  - symmetric cryptography ciphers with unknown key
  - information about key is predicted
- Results better than random guessing (not 100%)
  - even for weakened algorithms (smaller number of rounds)
- Any other suggestions?

Practical experience

- Better to start with few layers (e.g., 5) and increase later
  - evolution is much faster
- Better to use only few connectors instead full interconnection
  - propagation of single value is limited
  - single mutation should generally have only limited impact
- Algorithms with limited rounds are useful
  - starting with few rounds only gives insight into how complexity of problems increase and parameters of circuit can increased accordingly
- Test vectors should be changed either too rarely nor too often
  - EAC needs time for learning, but can over-learn on particular data

Open issues/future work

- How to efficiently probe most suitable settings of EAC
  - number of layers, number of internal functions, connectors, …
- Portability to hardware
  - circuit is evaluated directly in FPGA (programmable hw)
  - usually speedup in order of 10^3
- Test of idea with circuit internal state
  - state enables to process long data or repeat same several times
  - but much complex circuit evolve, harder to human analysis
- Circuit with modifiable connectors based on data
  - dynamic circuit, probably harder to evolve and analyze
- Good candidate problem for EAC from other areas?

Thank you for the attention!

- ***for*** *i* ***from*** *0* ***to*** *63*
- ***if*** *0 ≤ i ≤ 15* ***then***
- *f := (b* ***and*** *c)* ***or*** *((****not*** *b)* ***and*** *d)*
- *…*
- *g := i*
- *temp := d*
- *d := c*
- *c := b*
- *b := b +* ***leftrotate****((a + f + k[i] + w[g]) , r[i])*
- *a := temp*