Genetic Algorithm-based Electromagnetic Fault Injection

Antun Maldini

Niels Samwel

Stjepan Picek

Lejla Batina

Institute for Computing and Information Sciences – Digital Security
Radboud University Nijmegen

FDTC 2018
2018-09-13

Outline

Introduction

- Fault Injection (FI) – supply voltage glitching, clock glitching, *EM pulse*, laser pulse
- on SHA-3 (Keccak) – but generic
- which parameters to use? – optimization algorithm

Idea

What we set out to do

- make an algorithm for parameter optimization
- use it on SHA-3 (Keccak)
- make it better than what's previously been done

Contribution

What we did

- made an EA for parameter optimization!
- attacked SHA-3
- it's better than the baseline! (and previous results)

What are we optimizing?

Parameters

X, Y – the two spatial dimensions

offset – w.r.t. the trigger

intensity – power of the EM pulse

No. of repetitions – a primitive form of pulse shape

These are the ones we can control with the equipment we have.

Why are we optimizing?

- most parameter settings don't result in FI
- exhaustive search impractical

Exhaustive search

- *really* exhaustive – 10^{12} points, 30 years
- even just 100×100 spatial, 20 intensity, 100 offset – 37 days

Related work

- very little work on FI parameter optimization

Madau & al.

- EMFI susceptibility criterion
- all surface points ranked by this criterion, reject worst α%
- reject 50% of chip surface, with 80% faults kept
- by *fault* they mean any abnormal behavior

Carpi & al.

- supply voltage glitching
- two stages: a 2D search, followed by a 1D grid search
- genetic, later memetic algorithm

Experimental setup

Device tested: Cortex-M4F on STMicroelectronics board

Experimental setup

Device tested:	Cortex-M4F on STMicroelectronics board
Code running:	SHA3-512 (WolfSSL implementation, in C)

Experimental setup

Device tested:	Cortex-M4F on STMicroelectronics board
Code running:	SHA3-512 (WolfSSL implementation, in C)
Fault injection by:	Riscure EM probe, VCGlitcher

Experimental setup

Device tested:	Cortex-M4F on STMicroelectronics board
Code running:	SHA3-512 (WolfSSL implementation, in C)
Fault injection by:	Riscure EM probe, VCGlitcher
All controlled by:	Python code on PC

Measuring different behaviours

Some definitions

point: a tuple of $(X, Y, intensity, offset, \#rep.)$

measurement: a single sampling of a point

Measuring different behaviours

Some definitions

point: a tuple of $(X, Y, intensity, offset, \#rep.)$

measurement: a single sampling of a point

Several classes of behaviour:

- NORMAL – nothing happens
- RESET – target locks up
- SUCCESS – we get a faulty output of the right length

Measuring different behaviours

Some definitions

point: a tuple of $(X, Y, \mathit{intensity}, \mathit{offset}, \#\mathit{rep}.)$

measurement: a single sampling of a point

Several classes of behaviour:

- NORMAL – nothing happens
- RESET – target locks up
- SUCCESS – we get a faulty output of the right length

Behaviour is not completely determined by the point!

- do multiple (5) measurements per point
- behaviour changes $\rightarrow$ CHANGING class

Objectives & assumptions

Objectives

- good coverage of the parameter space – we know nothing in advance!
- speed

Objectives & assumptions

Objectives

- good coverage of the parameter space – we know nothing in advance!
- speed

Assumptions

- EM pulse too weak – NORMAL class
- EM pulse too strong – RESET class
- desired behaviour is somewhere in between

Evolutionary algorithms

- population-based metaheuristic
- used for general, non-convex optimization problems
- exploration vs. exploitation

Evolutionary algorithms

A general outline:

$$\begin{aligned}&\mathit{Input}: \ \mathit{Parameters\ of\ the\ algorithm}\\&\mathit{Output}: \ \mathit{Optimal\ solution\ set}\end{aligned}$$

$t \leftarrow 0$
$P(0) \leftarrow \mathit{CreateInitialPopulation}$
while $\mathit{TerminationCriterion}$ not satisfied **do**
  $t \leftarrow t + 1$
  $P'(t) \leftarrow \mathit{SelectMechanism}\ (P(t-1))$
  $P(t) \leftarrow \mathit{VariationOperators}(P'(t))$
end while
return $\mathit{OptimalSolutionSet}(P)$

Evolutionary algorithms

A general outline:

$$\begin{aligned} &Input: \; Parameters\ of\ the\ algorithm \\ &Output: \; Optimal\ solution\ set \end{aligned}$$

$t \leftarrow 0$
$P(0) \leftarrow CreateInitialPopulation$
while $TerminationCriterion$ not satisfied **do**
  $t \leftarrow t + 1$
  $P'(t) \leftarrow SelectMechanism\ (P(t-1))$
  $P(t) \leftarrow VariationOperators(P'(t))$
end while
return $OptimalSolutionSet(P)$

Genetic algorithms

A general outline:

$Input$: $Parameters\ of\ the\ algorithm$
$Output$: $Optimal\ solution\ set$

$t \leftarrow 0$
$P(0) \leftarrow CreateInitialPopulation$
while $TerminationCriterion$ not satisfied **do**
  $t \leftarrow t + 1$
  $P'(t) \leftarrow SelectMechanism\ (P(t-1))$
  $Ch(t) \leftarrow Mutate(Combine(P'(t)))$
  $P(t) \leftarrow$ Pick $sizeof(P(t))$ from $(Ch(t) \cup P(t))$
end while
return $OptimalSolutionSet(P)$

Our algorithm

Two phases: GA and local search

Our algorithm

Two phases: GA and local search

GA

- 20 generations of 50 units each
- roulette-wheel selection
- non-standard crossover
- elitism (with 1 elite individual)

Our algorithm

Two phases: GA and local search

GA

- 20 generations of 50 units each
- roulette-wheel selection
- non-standard crossover
- elitism (with 1 elite individual)

LS

- run after the GA is done
- further exploit the area around the SUCCESSful points found

Selection

- 3-tournament resulted in overly fast convergence
- roulette-wheel is slower, especially with large population
- keeping the best individual – useful when good points are rare

Crossover

Standard crossover

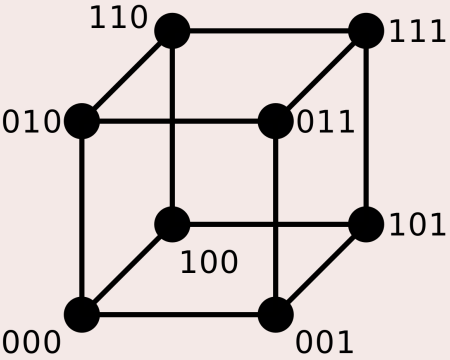
for *each parameter p* **do**
  $child.p \leftarrow random\_choice(parent_1.p, parent_2.p)$
end for

Our crossover

for *each parameter p* **do**
  $child.p \leftarrow random\ value\ in\ range\ [parent_1.p, parent_2.p]$
end for

Crossover

Illustrated on a 3-cube



Fitness function

- NORMAL – 2
- RESET – 5
- SUCCESS – 10
- CHANGING – ???

Fitness function

- NORMAL – 2
- RESET – 5
- SUCCESS – 10
- CHANGING – we look at the 5 measurements of a point

$$fitness_{\text{CHANGING}} = 4 + 0.2 \cdot N_{\text{NORMAL}} + 0.5 \cdot N_{\text{RESET}} + 1.2 \cdot N_{\text{SUCCESS}}$$

Local search

When we're done exploring...

for each SUCCESSful point P **do**
  for i from 1 to 10 **do**
    neighbour $\leftarrow$ random point from *neighbourhood*(P)
    scan neighbour
  end for
end for

Neighbourhood: cube centered on P, edge length 0.02

Results

- all statistics are averages over 5 runs
- average run length of 3301.6 points

TL;DR

	Random	GA	improvement
faulty msmts.	1.3%	58.8%	42.5 times
distinct faulty msmts.	1.0%	19.9%	20.5 times

...as % of all individual measurements

Results – details

	whole run		**first 500 points**	
	Random	GA	Random	GA
NORMAL	2955.8 (90.7%)	662.8 (18.9%)	452.6 (90.5%)	315.2 (63.0%)
RESET	65.0 (2.0%)	496.4 (15.0%)	9.8 (2.0%)	73.4 (14.7%)
CHANGING	232.4 (7.0%)	375.2 (11.4%)	36.0 (7.2%)	79.0 (15.8%)
SUCCESS	8.8 (0.3%)	1 807.2 (54.7%)	1.6 (0.3%)	32.4 (6.5%)
#faulty m.	228.2 (1.3%)	9700.4 (58.8%)	33.4 (1.3%)	260.8 (10.4%)
#distinct m.	160.8 (1.0%)	3288.4 (19.9%)	22.6 (0.9%)	158.8 (6.3%)

Exploiting faults?

Exploiting faults?

- Can we actually use the faulty outputs we have?

Exploiting faults?

- Can we actually use the faulty outputs we have?
- How?

Exploiting faults?

- Can we actually use the faulty outputs we have?
- How?
- Is it practical?

Exploiting faults?

- Can we actually use the faulty outputs we have? Yes.
- How?
- Is it practical?

Exploiting faults?

- Can we actually use the faulty outputs we have? Yes.
- How? Use DFA or AFA.
- Is it practical?

Exploiting faults?

- Can we actually use the faulty outputs we have? Yes.
- How? Use DFA or AFA.
- Is it practical? Mostly.

Algebraic Fault Analysis

- Luo & alii, 2018. (for SHA-3)
- Idea: let a SAT solver do the hard work
  1. represent internal state by boolean vars
  2. formulate algorithm & fault model as boolean statements (this provides the propagation constraints)
  3. obtain a (correct, faulty) output pair (these provide concrete constraints)
- enough implicit information to deduce part of state

Algebraic Fault Analysis

Recovering the state

load into SAT solver: $(correct, faulty)$
while more solutions exist **do**
  $solution \leftarrow SAT$.get_solution()
  SAT.add_constraint($\neg solution$)
end while

Solver eventually runs out of satisfiable solutions.

Bits which are same in all solutions are recoverable.

Algebraic Fault Analysis, specifics

- Luo & al. provide 3 fault models (8-bit, 16-bit, 32-bit)
- In *n*-bit fault model, faults are *n*-bit aligned
- also, three methods: single-fault, two-fault, two-fault with partially recovered state at χ_i^{23}
- we use Method III (the last one)

Results

GA

- 106 exploitable faults
- out of 14979 distinct faults (0.71%)
- out of 82540 measurements (0.141%)

Random

- 110 exploitable faults
- out of 947 distinct faults (11.61%)
- out of 100000 measurements (0.113%)

A bit more efficient – 24.6%.

Why the loss?

- the GA phase is "blind" (no exploitability knowledge)
- the LS phase searches around all SUCCESS points equally

To do:

Integrate exploitability checks in fitness function

Local search – neighbourhood?

- The share of unique faults looks lower than baseline (34% vs 70%)
- Not a fair comparison!
- Still, can we improve?

To do:

Figure out a better range & number of points to scan in neighbourhood

Questions?