Genetic Algorithm-based Electromagnetic Fault Injection

Antun Maldini[1], Niels Samwel[2], Stjepan Picek[3], and Lejla Batina[2]

[1] University of Zagreb, Zagreb Croatia
antun.maldini@fer.hr
[2] Radboud University, Nijmegen, The Netherlands
{n.samwel, lejla}@cs.ru.nl
[3] Delft University of Technology, Delft, The Netherlands
s.picek@tudelft.nl *

Abstract. Electromagnetic fault injection (EMFI) is a powerful active attack, requiring minimal modifications of the device under attack while having excellent penetration capabilities. The number of possible parameter combinations when characterizing an attack is usually huge, rendering exhaustive search impossible. In this work we present a novel evolutionary algorithm for optimizing the parameters for EM fault injection, which outperforms previous search methods for EMFI. The cryptographic device under attack is treated as a black box, with only a few very general assumptions on its inner workings. We test our evolutionary algorithm by attacking SHA-3 where we are able to obtain 40 times more faulty measurements and 20 times more distinct fault measurements than the random search. When coupled with the algebraic fault attack, we get 25% more exploitable faults per individual measurement.

Keywords: Electromagnetic Fault Injection, Genetic Algorithm, Local Search, Parameter optimization, SHA-3, Algebraic Fault Attack

1 Introduction

Implementation attacks do not target algorithm's security but the weaknesses in its implementation instead. Two well-known kinds of implementation attacks are side-channel attacks (SCAs) and fault injection (FI) attacks. Side-channel attacks are passive, non-invasive attacks where the device under attack operates within specified conditions and the attacker simply observes the physical leakages produced. Fault injection attacks are, on the other hand, active, more invasive attacks where the attacker inserts faults (glitches) in order to disrupt the normal behaviour of the algorithm.

* This work has been supported in part by Croatian Science Foundation under the project IP-2014-09-4882 and by the Technology Foundation TTW (Project 13499 TYPHOON), from the Dutch government.

Side-channel attacks received a lot of attention in the last few decades where we saw successful exploitation of several side channels like timing [1], power consumption [2], and EM emanation [3]. To use that information and deliver as powerful as possible attacks, researchers devised various strategies. Not surprisingly many of those strategies in the last few years are based on machine learning [4, 5] and deep learning [6].

When considering fault injection attacks, the situation is somewhat different in terms of analysis techniques. There are several sources of glitches such as laser pulses, electrical glitches, and electromagnetic radiation. A fault injection attack is successful if after exposing the device under attack to a specially crafted external interference, the device shows an unexpected behaviour i.e. a fault, which can be exploited by an attacker. Here, the challenge lies in selecting the appropriate parameters for a fault to succeed. If those parameters are not well chosen, the target will respond in a way that does not permit an actual fault analysis attack. When considering various sources of faults, there are different number of parameters and corresponding ranges. In general, the search space size of possible parameter values is large and relatively few points in the search space result in faults. Consequently, an interesting question is: how to find suitable parameter values or intervals, or more precisely, how to efficiently find the correct values in the search space? Surprisingly, the main options today are to use either random search or some sort of exhaustive search (since full exhaustive search is usually not possible the attacker will concentrate only in some regions with a certain precision, which we call here grid search). Analogously, if it is possible to make SCA more powerful by using machine learning (and more generally, artificial intelligence) one would expect the same to be possible with the fault injection.

In this paper, we discuss how to use a special type of metaheuristics called genetic algorithms in order to find parameter values resulting in faults for electromagnetic fault injection. A somewhat similar research direction is followed in several papers [7–9] but there the authors consider power glitching, which is a much simpler case from the search space size perspective, and they attack the PIN mechanism in a smartcard. Here, we use the faults obtained via our technique to mount an algebraic fault attack on a SHA-3 implementation where we consider only pulsed EMFI and we have a total of 5 parameters. We emphasize that our version of search algorithm differs significantly from previous works as detailed in the rest of the paper. Finally, our code is available as an open source implementation.[4]

2 Related work

Although a vast amount work has been done on fault injection itself, see e.g., [10–15], only a small fraction of it concerns parameter optimization.

In [16], the authors develop an EMFI susceptibility criterion, which they use to rank the points of the chip surface depending on how susceptible they are

[4] Github: `https://github.com/geneticemfaults/geneticemfaults`

to fault injection. The underlying assumption for the criterion is the Sampling Fault Model, described in [17]. The criterion itself is a combination of Power Spectral Density (measuring emitted power at the clock frequency) and Magnitude Squared Incoherence (measuring how linked the emitted signal is to the data being processed). They use a grid scan (in 2 spatial dimensions) to measure all the points and rank them according to the criterion; a share α of the highest-ranking points are kept for further scanning; the rest is thrown away. They are able to reject over 50% of the chip surface (75% in their best case), while keeping 80% of the points causing faults. However, by *fault*, they mean any perturbation of the normal behavior of the algorithm.

In [7], the authors apply several different methods to the problem of parameter optimization for supply voltage (VCC) glitching. They reduce the dimensionality of the problem by splitting the search in two stages. In the first stage, they look for the best (glitch voltage, glitch length) combination. In the second stage, 10 most promising (voltage, length) combinations are tried at each point in the time range (which is discretized into 100 instants). All parameters not explicitly specified are set as random. The methods are compared at the first stage – random search, FastBoxing and Adaptive zoom&bound algorithms, and a genetic algorithm. This approach is a "smart" search in 2D with a grid search in 1D.

That work is extended in [9] where the authors use a combination of genetic algorithm and local search (called memetic algorithm) in order to find faults even more efficiently. The authors consider power glitching with 3 parameters and are interested in fast characterization of the search space.

3 Preliminaries

3.1 Genetic Algorithms

Evolutionary algorithms represent population-based metaheuristic optimization techniques inspired by biological evolution and phenomena like mutation, recombination, and selection [18–20]. The solutions in the population compete and in that process improve their goodness as evaluated by a fitness function. Evolutionary algorithms often perform well in many types of problems because they ideally do not make assumptions about the underlying solutions' landscape. Today, there are many types of evolutionary algorithms, but probably the best known ones are genetic algorithms (GA). An instance of a genetic algorithm maps a real optimization problem to the natural concepts as follows:

1. The objective function (which we are optimizing) becomes the fitness function.
2. A solution (a point in the solution space) becomes an individual in the population.

The general pseudocode of an evolutionary algorithm is given below. Note, this is general enough to cover any type of evolutionary algorithm, including genetic algorithms.

1: $Input:$ $Parameters\ of\ the\ algorithm$
2: $Output:$ $Optimal\ solution\ set$
3: $t \leftarrow 0$
4: $P(0) \leftarrow CreateInitialPopulation$
5: **while** $TerminationCriterion$ **do**
6: $t \leftarrow t + 1$
7: $P'(t) \leftarrow SelectMechanism\ (P(t-1))$
8: $P(t) \leftarrow VariationOperators(P'(t))$
9: **end while**
10: $Return\ OptimalSolutionSet(P)$

3.2 Keccak/SHA-3

Keccak [21] is a cryptographic hashing algorithm that is the winner of the competition for Secure Hashing Algorithm 3 (SHA-3) held by NIST. It uses the sponge construction with a permutation at its core. Keccak is a versatile cryptographic primitive that can be used in different sizes with different security strengths. It can also be used in different modes, such as keyed mode to compute a MAC or encrypt and decrypt data or without a key to compute a hash value.

The Keccak-$f[b]$ permutation is defined by its width b, in our case $b = 1600$. Keccak-$f[b]$ is described as a sequence of operations on a state a that is a three-dimensional array of elements of $GF(2)$, namely $a[5, 5, 64]$. Coordinates x and y should be taken modulo 5 and coordinate z should be taken modulo 64. Sometimes an index is omitted implying the statement is valid for all values of the omitted indices. Keccak-$f[b]$ is an iterated permutation over 24 rounds of R.

$$R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$$

$$\begin{aligned}
\theta : C_{(x)} &= \sum_{y=0}^{4} A_{(x,y)} \\
D_{(x)} &= C_{(x-1)} + rot(C_{(x+1)}, 1) \\
A_{(x,y)} &= A_{(x,y)} + D_{(x)} \\
\pi \text{ and } \rho : B_{(y,2x+3y)} &= \mathrm{rot}(A_{(x,y)}, r(x,y)) \\
\chi : A_{(x,y)} &= B_{(x,y)} + (B_{(x+1,y)} \times B_{(x+2,y)} + 1) \\
\iota : A_{(0,0)} &= A_{(0,0)} + RC
\end{aligned}$$

All operations are carried out in $GF(2)$. Function $\mathrm{rot}(W, i)$ is a bitwise cyclic shift operation. The constants $r(x, y)$ are rotation offsets. RC is the round constant. For more details, see [21]. We refer to the linear steps as $\lambda = \pi \circ \rho \circ \theta$.

4 Experimental Setup

For the target, we use a Cortex-M4 STM32F407IG (Riscure "Piñata") board running a C implementation of SHA-3. This implementation is taken from the WolfSSL library.[5] The board communicates to a PC by a serial interface and

[5] WolfSSL, an embedded SSL/TLS library. Available at: `https://www.wolfssl.com/`

is powered by an external power supply. We use a Riscure EM probe and the VCGlitcher device that controls it. The whole setup is controlled by the code that is written in Python 2.x. For interfacing the Riscure equipment, we use Python bindings for the VCGlitcher C API (which is a 32-bit DLL).

The on-board code provides the trigger which signals that the cryptographic operation is in progress; this is used as a reference point for injecting the fault. In the case that the board gets stuck in an illegal state after a glitch, it needs to be reset. The only way to reliably reset this particular board is by cutting its power, which can take a significant fraction of a second, depending on the capacitors. We used 100 ms for this.

The physical dimensions of the chip package are 24×24 mm. Repositioning error of our XYZ table is 0.05 mm, which gives us a spatial grid of at most 480×480. Note that the limiting factor in our case is likely the size of the probe tip, which is much larger.

4.1 Parameters

There are multiple parameters one can vary to affect fault probability: position of the probe tip (X, Y, and Z), pulse intensity, time offset of the pulse, pulse duration, shape and angle of the probe tip, and pulse shape. In our experiments, we consider only a subset of these:

- two parameters (X and Y) for position, since the distance from the board (Z) can be compensated by a change in intensity. These parameters are real values in $[0, 1]$ range.
- glitch intensity, which regulates the voltage of the pulse. The SDK manual suggests that it is a percentage of power used. We use real values in the range $[0, 1]$.
- time offset, between 367 and 375 µs, because that is where the injection point must be, for the code we are running. We encode the offset at integer value (number of 2 ns ticks).
- number of repetitions of the pulse, as a primitive form of pulse shape. We set this parameter to be in the range $[1, 10]$.

We do not vary pulse duration, and we leave it to a fixed value of 40 ns. Similarly, we do not vary shape and angle of the probe tip, since changing those cannot be easily automated.

4.2 Search Space Size

As mentioned above, it is not possible to conduct exhaustive search when considering fault injection with realistic targets. Hence, the question that remains is as follows: what is the search space size and how could we efficiently sample it? When considering X and Y position, there is a 0.05 mm repositioning error and 24×24 mm chip size, which gives max resolution of 480×480. For the time offset, with a 2 ns resolution and the range between 367 and 375 microseconds there are in total 4 000 different values. We have no good rule for determining the smallest meaningful increment for the glitch intensity, but if we use a 1%

increment, that gives us a range of 100 values. The repetitions are selected to be a random integer value in range $[1, 10]$, which gives us 10 values.

The total size is therefore $480 * 480 * 4\,000 * 100 * 10 \approx 10^{12}$. At ≈ 0.16 seconds per point, this results in 29 203 years to conduct an exhaustive search. Since trying the same parameters multiple times does not necessarily always yield the same response, we conduct 5 measurements for each point. Even if we would completely ignore everything but X, Y, and offset, we would still need 29.2 years to conduct an exhaustive search.

5 Search Algorithm

5.1 Assumptions

From the viewpoint of the algorithm, we consider the device to be a black-box. It assumes that the objective function is not a "golf-course-like" (i.e., to be too flat), in which case the lack of a significant gradient in the fitness landscape means that there is no driving bias toward fitness optimum. The reasoning behind is that a very weak EM pulse will not affect the target at all, and we will observe the normal behaviour. Conversely, a very strong EM pulse will completely dishevel its operation and even potentially damage it. Consequently, we expect the faulty behaviour to occur somewhere between those two extremes, i.e., along the class border. Additionally, offset ranges (min to max offset) are set by the user, based on a rough expectation of the duration of the cryptographic algorithm.

Usually the objective function guides the optimization algorithm towards better solutions, and the algorithm ends when it finds the best one. Here, we do not want just a single "best" solution since not every fault we find will also be exploitable, and there are situations where more than one are required, so we aim to obtain multiple solutions.

5.2 GA Objectives

We require our algorithm to have the following characteristics:

1. Good coverage of the parameter space – since we do not know where the exploitable faults are located, we need to explore the search space efficiently.
2. Speed – we require the algorithm to be fast in finding the faults, otherwise there is no advantage of using it when compared to random search, for instance.

These two requirements are somewhat conflicting with each other. Because most of the parameter space is useless (i.e., has no faults), covering enough space to be reasonably secure we did not miss anything important means potentially wasting a lot of measurements.

Next, we introduce the terminology we use when discussing the search and possible outputs of the algorithm. A point is a distinct set of parameters, i.e., a point in the parameter space. A measurement is the result of a single attempt at glitching the target with those parameters.

When counting the faulty measurements, we distinguish between:

1. the total number of faulty measurements,
2. the number of distinct faulty responses (i.e., "unique faulty measurements").

The difference is that if a measurement results in a before-seen faulty output, the second one will not count again. As an example, we find a set of parameters resulting in a specific fault. Later, we find some other set of parameters that result in the same fault but we do not count that new faulty measurement into distinct faulty measurements. For the exploitability purposes, the second number is more interesting as detailed in the following sections.

We classify the board response in one of the following classes:

- NORMAL: for normal behavior, meaning the board performs as if we did not do anything.
- RESET: the board did not reply at all, requiring a reset to restore to normal operation.
- SUCCESS: the board produces an output/ciphertext/signature/hash different than the correct one.
- CHANGING: for each point we investigate, we perform 5 measurements. If all measurements are in the same class, the point is put into one of the first three classes; otherwise it goes into the CHANGING class.

Fitness values are set according to the class: SUCCESS has the highest fitness (10), followed by the CHANGING, then RESET (5), and finally, NORMAL (2). CHANGING points' fitness depends on its underlying measurements: a mix of NORMAL and RESET measurements is somewhat better than an all-RESET or all-NORMAL point, but having SUCCESSful individual measurements moves the fitness closer to an all-SUCCESS point. We calculate the fitness of a CHANGING point in the following way:

$$fitness_{\text{CHANGING}} = 4 + 1.2 * N\_SUCCESS + 0.2 * N\_NORMAL + 0.5 * N\_RESET. \quad (1)$$

Here, factors 0.2 and 0.5 are chosen in analogy to the values for NORMAL and RESET (which are 2 and 5) while the rest of the factors are selected for the scaling reasons. For example, 4 NORMAL and 1 RESET measurements give fitness 5.3, which is higher than the fitness of a RESET point (with all 5 RESET measurements). Similarly, 4 SUCCESS and 1 RESET measurements give fitness 9.3, which is lower than the fitness of a SUCCESS point (with all 5 SUCCESS measurements).

5.3 Algorithm Definition

Despite the fact that we use genetic algorithms like some previous works [7, 9], our custom made algorithm is quite different. We discuss the specifics of our design in the following paragraphs. Our algorithm has several parameters to be determined. We selected those values on the basis of our tuning experiments and recommendations from [7, 9].

More in detail, our algorithm consists of two separate phases as follows:

1. The first phase is a genetic algorithm where we run it for 20 generations with population size 50.
2. Only when the genetic algorithm is done, we start with the local search, which takes 10 randomly chosen points in neighbourhood of each SUCCESS point. Note that this is a significant difference from related works where both GA and local search worked at the same time. We opted first to concentrate on exploration aspect – GA (to explore various regions of the target) and only after that on exploitability aspect – local search (to concentrate on more promising regions). Naturally, GA itself has also exploitability component that is especially manifested in the crossover operator but we also designed a custom made crossover operator that promotes exploration perspective.

GA Phase Our algorithm begins with a genetic algorithm that runs for N generations and has a population of M individuals. The initial population is selected uniformly at random within parameter ranges. We aim to maximize the fitness value, which corresponds to having as much as possible SUCCESS points.

The first phase of GA is selection and we use roulette wheel selection. There, the fitness function assigns a fitness to possible solutions (in this case, points). This fitness level is used to associate a probability of selection with each solution. Those individuals that have better fitness are also proportionally more likely to be selected. We also use elitism where the elite size equals 1. This means that the best individual cannot be replaced by some other individual (e.g., mutated offspring). We also experimented with 3-tournament selection as used in [7,9] but we found it to be too restrictive, i.e., to promote too fast convergence resulting in obtaining solutions from only a small part of the search space.

After the selection phase, we run crossover. Note, this version of a crossover promotes explorability and it enables GA to traverse large parts of the search space. The pseudocode for crossover is given below:

```
1: for each parameter p do
2:     child.p = random value in range [parent1.p, parent2.p]
3: end for
```

Finally, the last step is the mutation operator that works as follows:

The mutation rate $p\_mutation$ is set to 5%. In the case the parameter gets out of its ranges, it is clipped to the edges of its range.

Local Search Phase After GA is done, we focus local search on the promising parts of the explored search space: the space around the intersection points (i.e., places where class values change), and the space around any faults we've already found. We define the neighbourhood of a point as a cube centred in it, with edge length equal to 0.02. By length of 0.02 in parameter space, we mean 2% of the range of that parameter. Parameters x, y, and intensity are all within the

```
for param in [x, y, intensity] do
    random = random value ∈ [0, 1]
    if random > p_mutation then
        r = uniform random from interval [−0.5, 0.5]
        param = param + r
    end if
end for
random = random value
if random > p_mutation then
    r = uniform random from interval [−0.5, 0.5]
    offset < −offset + r ∗ OFFSET_RANGE
end if
random = random value
if random > p_mutation then
    repetitions = random integer from [1, 10]
end if
```

range $[0, 1]$. For offset, it's 2% of OFFSET_RANGE (which is OFFSET_MAX - OFFSET_MIN). To determine the distance of values, we use the Euclidean distance.

5.4 Practical Considerations

Commonly, optimization algorithms (and nature-inspired metaheuristics in particular) rely on a large number of iterations. Another assumption usually made is that the evaluation of possible solution points is uniform. Here, we have expensive measurements, where the cost of evaluation depends not only on the property of the point itself, but also the context of its evaluation. Although our algorithm consists of genetic algorithm and local search, we denote it often as genetic algorithm but we always consider it to have also local search phase. We do not call our technique a memetic algorithm since the GA and local search phases are separated.

When considering EMFI, there's the probe tip which has to physically move to a different point. To do this with a sufficient precision requires a non-negligible amount of time – the exact amount varies depending on the setup, but it can be up to several seconds per measurement. In comparison, a reset requires just a fraction of a second (for our board, ≈100 ms to do it reliably). The measurement part itself is even faster – 30 ms or less. Thus, the order in which points are evaluated matters. Even with an optimal routing for any batch of N points, more batches mean more time wasted. For population-based algorithms, this translates to small population sizes not being as efficient as large ones.

Additionally, we may want to get a glimpse of the results even before the scan is finished, especially for long-running scans. In the case of a random or grid scan, this means splitting the scan into batches where each covers more or less the whole parameter space, since scanning points in the optimal order results in uneven coverage.

6 Results

In this section, we present our results when attacking SHA-3. First, we investigate how well is GA able to find faults (i.e., force the algorithm to output the wrong ciphertext) and then, whether such points can be used in order to obtain the state of the algorithm. There, we use algebraic fault analysis (AFA), as described in [22]. AFA eliminates the need for analysis of fault propagation as is needed in differential fault analysis; instead it uses a SAT solver to recover the state bits from a (clean output, faulty output) pair.

6.1 Finding Faults

The duration of GA is determined by the number of faults it finds. We conducted 5 independent runs with 2 074, 2 343, 3 353, 3 606, and 5 132 points, respectively. Each individual run will be different due to the stochastic nature of GA as well as the target response. To obtain statistically meaningful results, we report values averaged over all runs. On average, in each run there are 3 301.6 points, of which:

- 662.8 (18.9%) NORMAL
- 496.4 (15.0%) RESET
- 375.2 (11.4%) CHANGING
- 1 807.2 (54.7%) SUCCESS

This also means we conduct 16 508 individual measurements on average. Out of these, 9 700.4 (58.8%) are faulty, and 3 288.4 (19.9%) are unique/distinct. Comparing this with random search with 3 302 points, we get:

- 2 995.8 (90.7%) NORMAL
- 65.0 (2.0%) RESET
- 232.4 (7.0%) CHANGING
- 8.8 (0.3%) SUCCESS

Again, this would represent 16 510 individual measurements. Out of these, 228.2 (1.3%) are faulty, and 160.8 (1.0%) are unique/distinct. To conclude, when averaged over 5 runs, our GA algorithm gives 42.5 times more faulty measurements, and 20.5 times more distinct faulty ones. The somewhat lower share of distinct measurements for the GA algorithm can be explained by many SUCCESS points being close to each other due to the local search, thus being more likely to cause the same response.

In Tables 1 until 3, we give results for Random search and genetic algorithms when considering 500, 1 000, and 2 000 points, respectively. Observe how the results for random search do not change significantly with more measurements. At the same time, we observe that GA is very successful already for the smallest case where we use only 500 points and as we add more points, the percentage of SUCCESS points increases.

Finally, we depict the search space after random search or GA in Figures 1a until 1f. Figures 1a and 1b give results for X and Y parameters. Figures 1c until 1f also depict intensity as a parameter. We depict both cases with and without NORMAL points to improve the readability. The number of points for

Table 1: 500 points.

	Random	GA
NORMAL	452.6 (90.5%)	315.2 (63.0%)
RESET	9.8 (2.0%)	73.4 (14.7%)
CHANGING	36.0 (7.2%)	79.0 (15.8%)
SUCCESS	1.6 (0.3%)	32.4 (6.5%)
#faults	33.4 (1.3%)	260.8 (10.4%)
#distinct	22.6 (0.9%)	158.8 (6.3%)

Table 2: 1 000 points.

	Random	GA
NORMAL	910.4 (91.0%)	381.8 (38.2%)
RESET	19.6 (2.0%)	198.0 (19.8%)
CHANGING	67.2 (6.7%)	169.2 (16.9%)
SUCCESS	2.8 (0.3%)	251.0 (25.1%)
#faults	58.8 (1.2%)	1 530.4 (30.6%)
#distinct	40.4 (0.8%)	956.6 (19.1%)

Table 3: 2 000 points.

	Random	GA
NORMAL	1 814.6 (90.7%)	541.6 (27.1%)
RESET	36.6 (1.8%)	351.2 (17.6%)
CHANGING	144.2 (7.2%)	285.0 (14.2%)
SUCCESS	4.6 (0.2%)	822.2 (41.1%)
#faults	130.6 (1.3%)	4 606.4 (46.0%)
#distinct	93.4 (0.9%)	2 030.4 (20.3%)

each figure is 3 300 (naturally, figures that do not depict NORMAL points display less points).

6.2 Attacking SHA-3 in Practice

To the best of our knowledge, SHA-3 implementation has not yet been attacked in practice. Attacks do exist, but only simulated ones: [23] show that differential fault analysis (DFA) can be used to recover the complete state in around 80 faults on average, if the attacker is able to inject single-bit faults in the input of the penultimate round (i.e., θ_i^{22}), though they rely on brute-forcing the last few bits. According to [24] (itself an extension of [25]), this is around 500 single-bit random faults for the whole state. [24] generalize the attack to a single-byte fault model, recovering the state in around 120 random faults.

Algebraic fault analysis (AFA) seems more promising: in the progress through [25], [24], and [22], the authors manage to bring down the number of faults needed to recover the internal state with SHA3-512 down to under 10 with AFA and the 32-bit fault model.

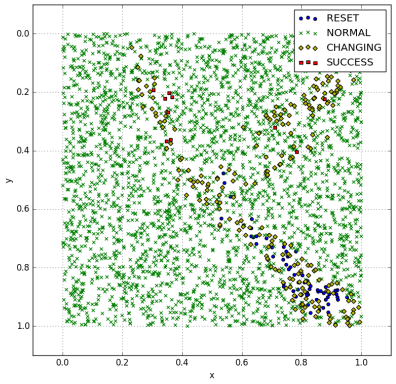
AFA has several advantages, besides being more efficient at recovering state, as follows:

- It does not require analysis of fault propagation through the algorithm, making it much easier to abstract the internal details.
- We can easily change the fault model, by just changing the appropriate constraints.
- Perhaps most importantly, it works for more relaxed fault models.

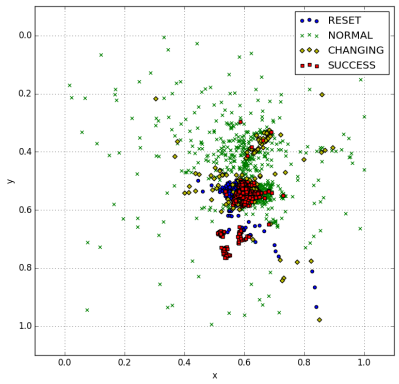
The attack in [22] allows the attacker to retrieve the state by injecting multiple faults in the input of the round 22 of Keccak. The faults are allowed to affect up to 1 unit of the state, where units are sized 8b, 16b, or 32b. As in previous work, we use θ_i^{22} as the fault injection point, and χ_i^{22} as the target state to recover. We reused their C++ retrieval code for this purpose.

The general idea behind AFA on SHA-3/Keccak is simple: use a SAT solver to do the work for us: we just need to provide appropriate constraints for it. We start with 1 600 Boolean variables representing the state (θ_i^{22}) and then provide constraints:

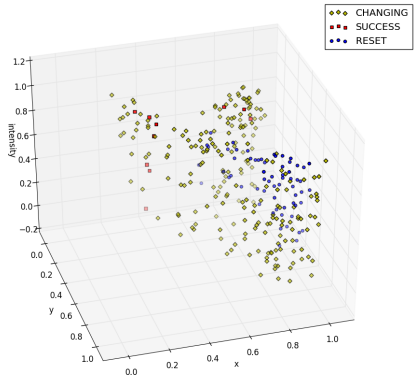
1. Fault Model – what kind of a fault do we cause? There's a separate set of (up to) 1 600 Boolean variables ($\Delta\theta_i^{22}$) representing the induced fault. $\theta_i^{22} \oplus \Delta\theta_i^{22}$ is the faulted state, before propagating through the final two rounds of the algorithm. Depending on what the fault model is, we add constraints such as "exactly one bit of $\Delta\theta_i^{22}$ is non-zero", corresponding to a one-bit fault model, or slightly more verbose ones for specifying things such as "we faulted a word-aligned 32-bit word", which would correspond to a 32-bit fault model in [22].
2. Keccak – how the (faulted) internal state propagates? For Keccak, the internal transformations can be relatively simply encoded as Boolean expressions. This implicitly tells the solver everything it needs to know about fault propagation, regardless of the fault model constraints. There are two cases we



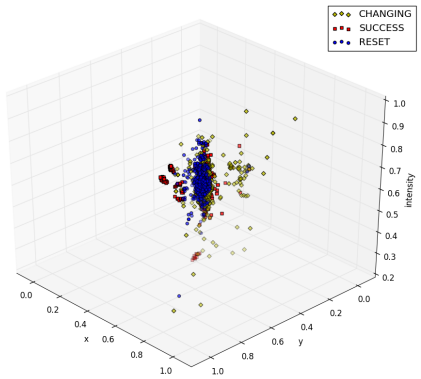
(a) Random search in 2D



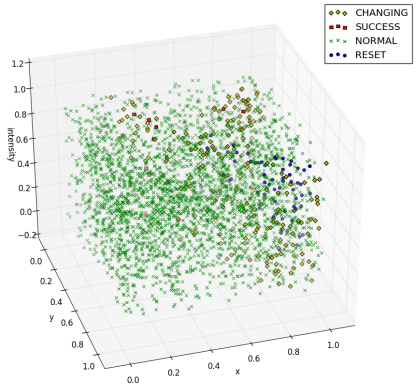
(b) GA and local search in 2D



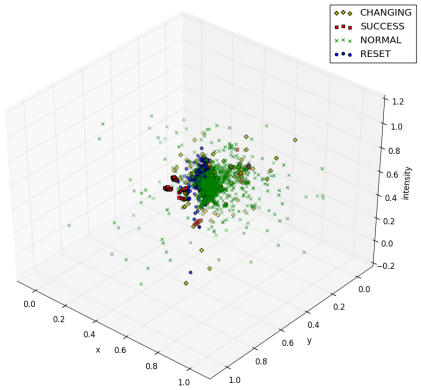
(c) Random search without NORMAL points



(d) GA and local search without NORMAL points



(e) Random search



(f) GA and local search

Fig. 1: Results for GA (with local search) and random search.

consider:

$$H = \iota^{23} \circ \chi \circ \pi \circ \rho \circ \theta \circ \iota^{22} \circ \chi \circ \pi \circ \rho \circ \theta(\theta_i^{22})$$

where H is the correct hash output, and

$$H' = \iota^{23} \circ \chi \circ \pi \circ \rho \circ \theta \circ \iota^{22} \circ \chi \circ \pi \circ \rho \circ \theta(\theta_i^{22} \oplus \Delta\theta_i^{22})$$

where H' is the faulty hash output.

3. Outputs – which are the concrete outputs? We give the SAT solver the actual values of H and H'. After so constraining the SAT solver, we can let it find a solution – an internal state satisfying all the constraints. Once it finds the first such solution, we ban this newly-found solution by adding it as an additional constraint, and let the SAT solver find another one. This process is repeated until no new solutions can be found.
   The bits of the state which are the same in all solutions are the ones we can recover: as for those which take different values in different solutions, their values are not entailed by the combined constraints of the fault model, the algorithm, and the outputs (i.e., the "real" constraints).
   Depending on the fault model and the version of SHA-3 (SHA3-512, SHA3-224, etc.), these constraints may or may not be enough to recover part of the state. In this case, additional constraints can be introduced, such as using two faulty hashes H'_1 and H'_2 at a time with a cost of extra Boolean variables and making it harder for the SAT solver (Method II in [22]), or first recovering part of the χ_i^{23} bits (Method III in [22]).

We applied the 32-bit fault model from [22] and Method III. The reason for this is a large number of potential faults to check while a short time for checking the exploitability of the induced faults is often an important factor. We tested the exploitability of all distinct faulty hashes obtained by our evolutionary algorithm, as well as of all those obtained by a random scan. While the share of distinct/unique faulty hashes depends on the size of the scan, the exploitability of a faulty hash does not. For this reason, we calculated the share of exploitable individual faults on all the samples we obtained (with the same hyperparameters).

The results are as follows: GA generated a total of 14 979 distinct faults (out of 82 540 individual measurements); 106 of these were exploitable 32-bit faults, for a share of 0.71%. Random search generated 947 distinct faults (out of 100 000 individual measurements); 110 of these were exploitable 32-bit faults, for a share of 11.61%. When translated into exploitable faults per individual measurement, this gives about 1.41×10^{-3} and 1.13×10^{-3} for GA and random search, respectively – an improvement of 24.6%.

Despite the fact that GA is still significantly more successful than the random search, we observe that actually most of the faults obtained with GA cannot be translated into exploitable faults. This results in a decrease between the performance difference of GA and random search. Still, such results are to be expected: since we never added the constraint of exploitability of faults into GA, it is hard to expect that GA will produce only such faults. Still, this could

be addressed by having a fitness function that integrates an analysis of fault exploitability.

7 Conclusions and Future Work

In this paper, we investigate how genetic algorithms can be used to facilitate faster and more powerful EMFI. When considering the search space size to investigate, it is evident that both random search and exhaustive search should not be the methods of choice. Indeed, our custom-made algorithm is able to find more than 40 times more faults than random search. Those results enable us almost 25% more exploitable faults per individual measurements when considering SHA-3 and algebraic fault attack. To the best of our knowledge, our algorithm is the most powerful currently available technique for finding parameters for EMFI.

Since there are only a few works considering how to find parameters leading to faults, that opens a number of potential research directions. We believe these two to be most interesting: exploring laser fault injection, and adding the notion of exploitability to the fitness function. The latter means that, instead of running local search on every SUCCESS point, we can first try to check whether it is exploitable, and only if it is, consider its neighborhood. Naturally, this also opens a question what is a good neighborhood to consider, or to state it differently: what is the best resolution for our search? Indeed, if all points within a certain neighborhood would result in no extra information we can use for exploitation, then there is no need for our algorithm to search within that region. Next, in this paper we concentrated on AFA Method III, but it would be interesting to investigate how our technique fares when used with Method II. Besides that, we plan to investigate different targets and improve the performance of our algorithm.

References

1. Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Proceedings of CRYPTO'96. Volume 1109 of LNCS., Springer-Verlag (1996) 104–113
2. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology. CRYPTO '99, London, UK, UK, Springer-Verlag (1999) 388–397
3. Quisquater, J.J., Samyde, D.: Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In Attali, I., Jensen, T., eds.: Smart Card Programming and Security, Berlin, Heidelberg, Springer Berlin Heidelberg (2001) 200–210
4. Lerman, L., Bontempi, G., Markowitch, O.: Side Channel Attack: an Approach Based on Machine Learning. In: Second International Workshop on Constructive SideChannel Analysis and Secure Design, Center for Advanced Security Research Darmstadt (2011) 29–41
5. Picek, S., Heuser, A., Jovic, A., Ludwig, S.A., Guilley, S., Jakobovic, D., Mentens, N.: Side-channel analysis and machine learning: A practical perspective. In: 2017

International Joint Conference on Neural Networks, IJCNN 2017, Anchorage, AK, USA, May 14-19, 2017. (2017) 4095–4102
6. Cagli, E., Dumas, C., Prouff, E.: Convolutional neural networks with data augmentation against jitter-based countermeasures - profiling attacks without preprocessing. In: Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings. (2017) 45–68
7. Carpi, R.B., Picek, S., Batina, L., Menarini, F., Jakobovic, D., Golub, M.: Glitch it if you can: Parameter search strategies for successful fault injection. In Francillon, A., Rohatgi, P., eds.: Smart Card Research and Advanced Applications, Cham, Springer International Publishing (2014) 236–252
8. Picek, S., Batina, L., Jakobovic, D., Carpi, R.B.: Evolving genetic algorithms for fault injection attacks. In: 2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). (May 2014) 1106–1111
9. Picek, S., Batina, L., Buzing, P., Jakobovic, D.: Fault injection with a new flavor: Memetic algorithms make a difference. In Mangard, S., Poschmann, A.Y., eds.: Constructive Side-Channel Analysis and Secure Design, Cham, Springer International Publishing (2015) 159–173
10. Kömmerling, O., Kuhn, M.G.: Design principles for tamper-resistant smartcard processors. In: Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology, Berkeley, CA, USA, USENIX Association (1999) 2–2
11. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In Fumy, W., ed.: Advances in Cryptology — EUROCRYPT '97, Berlin, Heidelberg, Springer Berlin Heidelberg (1997) 37–51
12. Samwel, N., Batina, L.: Practical fault injection on deterministic signatures: The case of eddsa. In Joux, A., Nitaj, A., Rachidi, T., eds.: Progress in Cryptology – AFRICACRYPT 2018, Cham, Springer International Publishing (2018) 306–321
13. Aghaie, A., Moradi, A., Rasoolzadeh, S., Schellenberg, F., Schneider, T.: Impeccable circuits. Cryptology ePrint Archive, Report 2018/203 (2018) `https://eprint.iacr.org/2018/203`.
14. O'Flynn, C.: Fault injection using crowbars on embedded systems. Cryptology ePrint Archive, Report 2016/810 (2016) `https://eprint.iacr.org/2016/810`.
15. Martín, H., Korak, T., Millán, E.S., Hutter, M.: Fault attacks on strngs: Impact of glitches, temperature, and underpowering on randomness. IEEE Transactions on Information Forensics and Security **10**(2) (2015) 266–277
16. Madau, M., Agoyan, M., Maurine, P.: An EM fault injection susceptibility criterion and its application to the localization of hotspots. In: International Conference on Smart Card Research and Advanced Applications, Springer (2017) 180–195
17. Ordas, S., Guillaume-Sage, L., Maurine, P.: EM injection: Fault model and locality. In: Fault Diagnosis and Tolerance in Cryptography (FDTC), 2015 Workshop on, IEEE (2015) 3–13
18. Holland, J.H.: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. The MIT Press, Cambridge, USA (1992)
19. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin Heidelberg New York, USA (2003)
20. Bäck, T., Fogel, D., Michalewicz, Z., eds.: Evolutionary Computation 1: Basic Algorithms and Operators. Institute of Physics Publishing, Bristol (2000)

21. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: The Keccak reference (January 2011) http://keccak.noekeon.org/.
22. Luo, P., Athanasiou, K., Fei, Y., Wahl, T.: Algebraic fault analysis of SHA-3 under relaxed fault models. IEEE Transactions on Information Forensics and Security (2018)
23. Bagheri, N., Ghaedi, N., Sanadhya, S.K.: Differential fault analysis of SHA-3. In Biryukov, A., Goyal, V., eds.: Progress in Cryptology – INDOCRYPT 2015, Cham, Springer International Publishing (2015) 253–269
24. Luo, P., Fei, Y., Zhang, L., Ding, A.A.: Differential fault analysis of SHA-3 under relaxed fault models. Journal of Hardware and Systems Security **1**(2) (Jun 2017) 156–172
25. Luo, P., Fei, Y., Zhang, L., Ding, A.A.: Differential fault analysis of SHA3-224 and SHA3-256. In: 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC). (Aug 2016) 4–15