Article 25fa pilot End User Agreement

This publication is distributed under the terms of Article 25fa of the Dutch Copyright Act (Auteurswet) with explicit consent by the author. Dutch law entitles the maker of a short scientific work funded either wholly or partially by Dutch public funds to make that work publicly available for no consideration following a reasonable period of time after the work was first published, provided that clear reference is made to the source of the first publication of the work.

This publication is distributed under The Association of Universities in the Netherlands (VSNU) 'Article 25fa implementation' pilot project. In this pilot research outputs of researchers employed by Dutch Universities that comply with the legal requirements of Article 25fa of the Dutch Copyright Act are distributed online and free of cost or other barriers in institutional repositories. Research outputs are distributed six months after their first online publication in the original published version and with proper attribution to the source of the original publication.

You are permitted to download and use the publication for personal purposes. All rights remain with the author(s) and/or copyrights owner(s) of this work. Any use of the publication other than authorised under this licence or copyright law is prohibited.

If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please contact the Library through email: copyright@ubn.ru.nl, or send a letter to:

University Library
Radboud University
Copyright Information Point
PO Box 9100
6500 HA Nijmegen

You will be contacted as soon as possible.

Chapter 13

Optimizing Electromagnetic Fault Injection with Genetic Algorithms

Antun Maldini, Niels Samwel, Stjepan Picek, and Lejla Batina

13.1 Introduction

Embedded security devices such as ID or bank cards, key immobilizers, and mobile phones are omnipresent in our lives and the threats to them directly affect the security and privacy of our data. The attackers often target the weaknesses of implementations rather than the algorithms running on those chips. Basically, those so-called implementation attacks do not focus on the algorithm itself but rather exploit some physical effects. Those effects, i.e. physical leakages become available due to the actual implementation of the algorithms on a platform that is typically constrained in terms of area, power, energy, etc. Two well-known types 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 (e.g. by glitching some parameters like voltage, power, clock, etc.) in order to disrupt the normal behavior of the algorithm.

A. Maldini (✉)

Faculty of Electrical Engineering and Computing, University of Zagreb, Zagreb, Croatia

N. Samwel · L. Batina

Digital Security Group, Radboud University, Nijmegen, The Netherlands

e-mail: nsamwel@cs.ru.nl; lejla@cs.ru.nl

S. Picek

Cyber Security Group, Delft University of Technology, Delft, The Netherlands

e-mail: s.picek@tudelft.nl

Side-channel attacks received a lot of attention in the last few decades where we saw successful exploitation of several side channels like timing [11], power consumption [12], and EM emanation [25]. To use that information and deliver powerful attacks that are even capable of recovering the secret keys, researchers devised various strategies. Not surprisingly, many of those strategies in the last few years are based on machine learning [14, 24] and deep learning [7].

When considering fault injection attacks, the situation is somewhat different in terms of attack methodology and analysis techniques. In certain scenarios, just one fault can lead to a total break of a cryptosystem as described in the first paper of Boneh et al. [6]. While it was clear that one fault can have devastating consequences on the security of a system, it was not immediately clear how complicated can be to insert the faults of interest. The first work that illustrates a practical fault injection attack on an actual RSA cryptosystem was done by Aumüller et al. [2]. They describe the engineering efforts behind a successful fault injection and possible countermeasures. Fault injection is often possible through glitching techniques. There are several sources of glitches possible 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 behavior, 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, we encounter 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 follow-up 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 available so far are to use either random search or some sort of exhaustive search. This is mainly due to the fact that a complete exhaustive search is usually not feasible so the attacker should focus on specific regions with a certain precision, and perform a so-called grid search.

In our work, we start from this observation and turn to some well-known machine learning strategies to deal with what is basically an optimization problem with a large number of parameters. Another motivation comes from relevant experiences with side-channel analysis and machine learning. In principle, if it is possible to make SCA more powerful by using machine learning (and more generally, artificial intelligence) one would expect the same for fault injection.

In this chapter, we shed some light on the above-mentioned questions and observations. We discuss how one could 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 previous works [8, 22, 23] but there the authors consider power glitching,

which is only a subset of the search space we have to deal with for EMFI. In addition, they attack a PIN checking mechanism on a smartcard. In our research, we use the faults obtained via a novel technique to mount an algebraic fault attack on a SHA-3 implementation where we consider pulsed EMFI only resulting in a total of five parameters. We emphasize that our version of search algorithm differs significantly from previous works as detailed in the rest of this chapter. Finally, our code is available as an open source implementation.¹

13.2 Related Work

Although a vast amount of work has been done on fault injection itself, see, e.g., [1, 6, 13, 19, 20, 26], only a small fraction of it concerns parameter optimization.

In [18], the authors develop an EMFI susceptibility criterion, which they use to rank the points of the chip surface depending on how sensitive they are to fault injection. The underlying assumption for the criterion is the sampling fault model, described in [21]. 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 two 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 [8], the authors apply several different methods to the problem of parameter optimization for the supply voltage (VCC) glitching. They reduce the dimensionality of the problem by splitting the search into two stages. In the first stage, they look for the best (glitch voltage, glitch length) combination. In the second stage, ten 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 [23] where the authors use a combination of a genetic algorithm and local search (called memetic algorithm) in order to find faults even more efficiently. The authors consider power glitching with three parameters and are interested in a fast characterization of the search space.

¹Github: <https://github.com/geneticemfaults/geneticemfaults>.

13.3 Preliminaries

13.3.1 Genetic Algorithms

Evolutionary algorithms represent population-based metaheuristic optimization techniques inspired by biological evolution and phenomena like mutation, recombination, and selection [3, 9, 10]. 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 in Algorithm 1. Note that this is general enough to cover any type of evolutionary algorithm, including genetic algorithms.

13.3.2 Keccak/SHA-3

In this work, we apply our attack on a cryptographic hash function, which is the new SHA-3 standard [5], and also known as Keccak. The Keccak main function is a sponge construction with a permutation as its core operation. Keccak is a cryptographic primitive that can be used in different modes (such as keyed and unkeyed) to compute hash values, MACs or to encrypt/decrypt data.

The core permutation named Keccak- $f[b]$ is defined by its width b and in our case, we use the full width where $b = 1600$. The permutation is described as a sequence of operations on a state a . The state is a three-dimensional array

Algorithm 1: Evolutionary algorithm pseudocode

Input : Parameters of the algorithm
Output : Optimal solution set

```

1  $t \leftarrow 0$ ;
2  $P(0) \leftarrow \text{CreateInitialPopulation}$ ;
3 while  $\text{TerminationCriterion}$  do
4    $t \leftarrow t + 1$ ;
5    $P'(t) \leftarrow \text{SelectMechanism}(P(t - 1))$ ;
6    $P(t) \leftarrow \text{VariationOperators}(P'(t))$ ;
7 return:  $\text{OptimalSolutionSet}(P)$ 
```

of elements in $GF(2)$. There are 5 rows and 5 columns, each of length 64, i.e. $a[5, 5, 64]$. Keccak- $f[b]$ is an iterated permutation over 24 rounds, the round function R is defined as follows:

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

In total there are five different steps that modify the state. An omitted index implies that the statement is valid for all values of that index.

$$\theta : \quad a(x, y, z) \leftarrow a(x, y, z) + \sum_{y'=0}^4 a(x-1, y', z) + \sum_{y'=0}^4 a(x+1, y', z-1),$$

$$\pi \text{ and } \rho : a(y, 2x + 3y) \leftarrow \text{rot}(a(x, y), r(x, y)),$$

$$\chi : \quad a(x) \leftarrow a(x) + (a(x+1) + 1) \cdot a(x+2),$$

$$\iota : \quad a(0, 0) \leftarrow a(0, 0) + RC$$

All operations are carried out in $GF(2)$. The function $\text{rot}(W, i)$ is a bitwise cyclic shift operation, where the constants $r(x, y)$ are rotation offsets. The value RC is the round constant, there is a different value for each round. For more details, see [5].

13.4 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.² The board communicates with a PC by a serial interface and 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). Figure 13.1 shows a photo of the setup.

The on-board code provides the trigger which signals that the cryptographic operation is in progress. This trigger is used as a reference point for injecting the fault. In 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 event.

²WolfSSL, an embedded SSL/TLS library. Available at: <https://www.wolfssl.com/>.

Fig. 13.1 The photo of the setup

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.

13.4.1 Parameters

There are multiple parameters one can vary to affect fault probability as follows:

- the spatial position of the probe tip (there are three degrees of freedom; X , Y , Z);
- the moment when the EM pulse fires;
- the pulse intensity;
- the shape of EM probe and the angle w.r.t. the target;
- the shape of EM pulse w.r.t. time

In our experiments, we consider only a subset of those as listed below:

- We consider a position, as a pair of parameters (X , Y). We do not vary the distance from the board (Z), since this can be largely compensated for by a change in intensity. These parameters are real values in $[0, 1]$ range.
- The glitch intensity regulates the voltage of the pulse. The SDK manual suggests it to be set to a percentage of power used. We use real values in the range $[0, 1]$.
- We consider also time offset to be between 367 and 375 μ s, 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).
- A number of repetitions of the pulse is a primitive form of the 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 automatically is (for us) hard enough to make them unsuitable for automatic optimization.

13.4.2 Search Space Size

As mentioned above, it is not possible to conduct an exhaustive search when considering fault injection with realistic targets. Hence, the question that remains is the following: 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 μ s there are in total 4000 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 the range [1, 10], which gives us ten values.

The total parameter set size is therefore $480 * 480 * 4000 * 100 * 10 \approx 10^{12}$. At ≈ 0.16 s 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 five measurements for each point. Even if we would completely ignore everything except X , Y , and offset, we would still need 29.2 years to conduct an exhaustive search.

13.5 Search Algorithm

13.5.1 Assumptions

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 behavior. Conversely, a very strong EM pulse will completely dishevel its operation and even potentially damage it. Consequently, we expect the faulty behavior 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 solution is required, so we aim to obtain multiple solutions.

13.5.2 *GA Objectives*

We require our algorithm to have the following two 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 requirements are somewhat conflicting with each other. Basically, as most of the parameter space is useless (i.e., has no faults), covering enough space to make sure we did not miss anything important implies 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 in the following: if a measurement results in a before-seen faulty output, the second one will not count again. As an example, assume 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 five 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 individual measurements belonging to the class SUCCESS moves the fitness closer to an all-SUCCESS point. We calculate the fitness of a CHANGING point in the following way:

$$\text{fitness}_C = 4 + 1.2 * N_S + 0.2 * N_N + 0.5 * N_R. \quad (13.1)$$

Here, N_S represents the number of SUCCESS points, N_R the number of RESET points, and N_N the number of NORMAL points. Finally, 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).

13.5.3 Algorithm Definition

Despite the fact that we use genetic algorithms like similar to some previous works [8, 23], 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 [8, 23].

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

1. The first phase is a genetic algorithm that we run for 20 generations with population size 50.
2. Only when the genetic algorithm is done, we start with the local search, which takes ten randomly chosen points in the neighborhood 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 many as possible SUCCESS points.

The first phase of a GA is selection; we use roulette-wheel selection. In roulette-wheel selection, the fitness function assigns a fitness to possible solutions (in this case, points in parameter-space). This fitness level is used to associate a probability of being selected for each solution: the probability that an individual will be selected is directly proportional to its fitness. (More precisely, it is equal to its share in the overall fitness: for the i -th individual, $P_i = \frac{\text{fitness}_i}{\sum_k \text{fitness}_k}$.)

Note that, since the selection is done randomly, it is possible that low-quality individuals are picked. This is not troubling since a GA would not function otherwise. However, it allows for the possibility of an excellent, hard-found solution being accidentally lost. A common countermeasure is *elitism*: with elitism, a number of best-ranking individuals (called the *elite*) are always chosen. We use elitism, with the elite size equal to 1, so only the single best individual gets carried over.

We also experimented with a 3-tournament selection as used in [8, 23] but found it too restrictive, since it promotes a convergence too quickly, resulting in solutions being obtained from only a small part of the search space. This went directly against the objective of a good coverage of the search space.

After the selection phase finishes, the crossover can start. The purpose of crossover is to combine the existing solutions to produce better ones. There are many ways to do the crossover. When we imagine our individuals as points—or rather, vectors—in the 5-dimensional parameter space, then each of the parameters corresponds to one element of this vector. When combining two such individuals, a traditional crossover might take some elements from one parent, and the rest from the other parent. The version of crossover that we used instead picks a point in-between the respective parents' elements. This promotes explorability and enables GA to traverse large parts of the search space.³ The pseudocode for this crossover is given in Algorithm 2, and the pseudocode for the mutation in Algorithm 3. 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.

³The parent points define an axis-aligned parallelepiped in parameter-space; the parents are placed on the diagonally opposite vertices. In a Hamming cube, these would be the all-zeros and all-ones vertices. The first crossover variant corresponds to picking one of its vertices, whereas the second crossover variant corresponds to picking a point within it.

Algorithm 2: Crossover operator

```

1 for each parameter  $p$  do
2    $child.p = \text{random value in range } [parent1.p, parent2.p];$ 

```

Algorithm 3: Mutation operator

```

1  $r_1, r_2 = \text{uniformly random from } [-0.5, 0.5];$ 
2 for param in  $[x, y, \text{intensity}]$  do
3   with probability  $p\_mutation$ :
4      $param = param + r_1;$ 
5 with probability  $p\_mutation$ :
6    $offset = offset + r_2 * OFFSET\_RANGE;$ 
7 with probability  $p\_mutation$ :
8    $repetitions = \text{random integer from } [1, 10];$ 

```

Local Search Phase

After the GA is done, we use local search to focus on the promising parts of the explored search space as follows: the space around the intersection points (i.e., places where class values change), and the space around any faults that were already found. We define the neighborhood of a point as a cube centered 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 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.

13.5.4 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 on the context of its evaluation. Although our algorithm consists of a genetic algorithm and local search, we denote it often as a 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 is 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 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.

13.6 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. For this purpose, we use algebraic fault analysis (AFA), as described in [17]. AFA eliminates the need for analysis of fault propagation as it is needed in differential fault analysis; instead, it uses a SAT solver to recover the state bits from a (clean output, faulty output) pair.

13.6.1 Finding Faults

The duration of GA is determined by the number of faults it finds. We conducted five independent runs with 2074, 2343, 3353, 3606, and 5132 points, respectively. Each individual run is different due to the stochastic nature of GA as well as the target response. To obtain statistically meaningful results, we report averaged values over all runs and report results in Table 13.1. For GA, on average, in each run, there are 3301.6 points, which means we conduct 16,508 individual measurements on average. Out of these, 9700.4 (58.8%) are faulty, and 3288.4 (19.9%) are unique/distinct. To compare, we use random search with 3302 points, which represents 16,510 individual measurements. Out of these, 228.2 (1.3%) are faulty, and 160.8 (1.0%) are unique/distinct.

Table 13.1 Statistical results for GA and random search

	GA	Random
NORMAL	662.8 (18.9%)	2995.8 (90.7%)
RESET	496.4 (15.0%)	65.0 (2.0%)
CHANGING	375.2 (11.4%)	232.4 (7.0%)
SUCCESS	1807.2 (54.7%)	8.8 (0.3%)

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 Table 13.2, we give results for Random search and genetic algorithms when considering 500, 1000, and 2000 points. 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.

We depict the search space after random search and GA in Fig. 13.2a–f. Figure 13.2a, b give results for X and Y parameters. Figure 13.2c–f also depict intensity as a parameter. We depict both cases with and without NORMAL points to improve the readability. The number of points for each figure is 3300 (figures not depicting NORMAL points have fewer points).

Finally, we show the results for GA as it progresses through the evolution process. More precisely, in Fig. 13.3a, we depict the search results for the first 500 points. We can observe that although there are several SUCCESS points, in this phase GA mainly finds NORMAL points spread across the search space. Figure 13.3b depicts results for 500–1000 points range. Here, we can see that GA does not find so many NORMAL points but actually manages to find a significant number of CHANGING and RESET points. We also see a good amount of SUCCESS points. In Fig. 13.3c, we show the results for points between 1000 and 2343 (end of search). Here, we see a large number of SUCCESS points where there are one large cluster and three smaller ones. CHANGING and RESET points occur in the same large cluster as the majority of SUCCESS points. NORMAL points occur in a number of small clusters surrounding the main cluster. To conclude, we see that GA is able to find SUCCESS points even with a small number of examined points but its true strength lies when there is a sufficient number of evaluations in order to guide the convergence.

13.6.2 SHA-3 Attack in Practice

To the best of our knowledge, SHA-3 implementation has not yet been attacked in practice. Attacks do exist, but only on simulated data such as [4]. The authors 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 [16] (itself an extension of [15]), this is around 500 single-bit random faults for the whole state. The work in [16] generalizes the attack to a single-byte fault model, recovering the state in around 120 random faults.

Table 13.2 Random search and GA results for various search stages

Points	Algorithm	NORMAL	RESET	CHANGING	SUCCESS	#Faults	#Distinct
500	Random	452.6 (90.5%)	9.8 (2.0%)	36.0 (7.2%)	1.6 (0.3%)	33.4 (1.3%)	22.6 (0.9%)
	GA	315.2 (63.0%)	73.4 (14.7%)	79.0 (15.8%)	32.4 (6.5%)	260.8 (10.4%)	158.8 (6.3%)
1000	Random	910.4 (91.0%)	19.6 (2.0%)	67.2 (6.7%)	2.8 (0.3%)	58.8 (1.2%)	40.4 (0.8%)
	GA	381.8 (38.2%)	198.0 (19.8%)	169.2 (16.9%)	251.0 (25.1%)	1530.4 (30.6%)	956.6 (19.1%)
2000	Random	1814.6 (90.7%)	36.6 (1.8%)	144.2 (7.2%)	4.6 (0.2%)	130.6 (1.3%)	93.4 (0.9%)
	GA	541.6 (27.1%)	351.2 (17.6%)	285.0 (14.2%)	822.2 (41.1%)	4606.4 (46.0%)	2030.4 (20.3%)

Fig. 13.2 Results for GA (with local search) and random search. (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. 13.3 Results for GA with local search depicting several stages of the search process. **(a)** Results for points 0–500. **(b)** Results for points 500–1000. **(c)** Results for points 1000–2343

Algebraic fault analysis (AFA) seems more promising. Luo et al. 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 in the progress described in [15–17].

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 [17] 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 the 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 1600 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) 1600 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 [17].
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 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 [17]), or first recovering part of the χ_i^{23} bits (Method III in [17]).

We applied the 32-bit fault model from [17] 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.

13.7 Conclusions and Future Work

In this chapter, we investigate how genetic algorithms can be used to facilitate faster and more powerful EMFI. When considering the search space size one needs 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 the parameter search that is leading to faults, this opens a number of potential research directions. We believe the following two would be the most interesting: (1) exploring laser fault injection, and adding the notion of exploitability to the fitness function. The latter means that, instead of running a 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. (2) 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 also plan to investigate different targets and improve the performance of our algorithm.

Acknowledgements 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.

References

1. A. Aghaie, A. Moradi, S. Rasoolzadeh, F. Schellenberg, T. Schneider, Impeccable circuits, Cryptology ePrint Archive, Report 2018/203, 2018. <https://eprint.iacr.org/2018/203>
2. C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, J.-P. Seifert, Fault attacks on RSA with CRT: concrete results and practical countermeasures, in *CHES*, pp. 260–275 (2002)
3. T. Bäck, D.B. Fogel, Z. Michalewicz (eds.), *Evolutionary Computation 1: Basic Algorithms and Operators* (Institute of Physics Publishing, Bristol, 2000)
4. N. Bagheri, N. Ghaedi, S.K. Sanadhya, Differential fault analysis of SHA-3, in *Progress in Cryptology—INDOCRYPT 2015* (Springer, Cham, 2015), pp. 253–269
5. G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, The Keccak reference, January 2011. <http://keccak.noekeon.org/>
6. D. Boneh, R.A. DeMillo, R.J. Lipton, On the importance of checking cryptographic protocols for faults (extended abstract), in *Advances in Cryptology - Proceeding of the EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques*, Konstanz, May 11–15 (1997), pp. 37–51
7. E. Cagli, C. Dumas, E. Prouff, Convolutional neural networks with data augmentation against jitter-based countermeasures - profiling attacks without pre-processing, in *Cryptographic Hardware and Embedded Systems - CHES 2017 - Proceedings of the 19th International Conference, 2017, Taipei, September 25–28 (2017)*, pp. 45–68
8. R.B. Carpi, S. Picek, L. Batina, F. Menarini, D. Jakobovic, M. Golub, Glitch it if you can: parameter search strategies for successful fault injection, in *Smart Card Research and Advanced Applications*, ed. by A. Francillon, P. Rohatgi (Springer, Cham, 2014), pp. 236–252
9. A.E. Eiben, J.E. Smith, *Introduction to Evolutionary Computing* (Springer, Berlin, 2003)
10. J.H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence* (The MIT Press, Cambridge, 1992)
11. P.C. Kocher, Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems, in *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology* (Springer, London, 1996), pp. 104–113
12. P. Kocher, J. Jaffe, B. Jun, Differential power analysis, in *Annual International Cryptology Conference* (Springer, Berlin, 1999), pp. 388–397
13. O. Kömmerling, M.G. Kuhn, Design principles for tamper-resistant smartcard processors, in *Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology* (USENIX Association, Berkeley, 1999), pp. 2–2

14. L. Lerman, G. Bontempi, O. Markowitch, Side channel attack: an approach based on machine learning, in *Second International Workshop on Constructive SideChannel Analysis and Secure Design*, pp. 29–41 (Center for Advanced Security Research, Darmstadt, 2011)
15. P. Luo, Y. Fei, L. Zhang, A.A. Ding, Differential fault analysis of SHA3-224 and SHA3-256, in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 4–15 (2016)
16. P. Luo, Y. Fei, L. Zhang, A.A. Ding, Differential fault analysis of SHA-3 under relaxed fault models. *J. Hardware Syst. Secur.* **1**(2), 156–172 (2017)
17. P. Luo, K. Athanasiou, Y. Fei, T. Wahl, Algebraic fault analysis of SHA-3 under relaxed fault models. *IEEE Trans. Inf. Forensics Secur.* **13**, 1752–1761 (2018)
18. M. Madau, M. Agoyan, P. Maurine, 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, Cham, 2017), pp. 180–195
19. H. Martín, T. Korak, E.S. Millán, M. Hutter, Fault attacks on STRNGs: impact of glitches, temperature, and underpowering on randomness. *IEEE Trans. Inf. Forensics Secur.* **10**(2), 266–277 (2015)
20. C. O’Flynn, Fault injection using crowbars on embedded systems, Cryptology ePrint Archive, Report 2016/810 (2016). <https://eprint.iacr.org/2016/810>
21. S. Ordas, L. Guillaume-Sage, P. Maurine, EM injection: fault model and locality, in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), 2015* (IEEE, Piscataway, 2015), pp. 3–13
22. S. Picek, L. Batina, D. Jakobovic, R.B. Carpi, Evolving genetic algorithms for fault injection attacks, in *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May (2014), pp. 1106–1111
23. S. Picek, L. Batina, P. Buzing, D. Jakobovic, Fault injection with a new flavor: memetic algorithms make a difference, in *Constructive Side-Channel Analysis and Secure Design*, ed. by S. Mangard, A.Y. Poschmann (Springer, Cham, 2015), pp. 159–173
24. S. Picek, A. Heuser, A. Jovic, S.A. Ludwig, S. Guilley, D. Jakobovic, N. Mentens, Side-channel analysis and machine learning: a practical perspective, in *2017 International Joint Conference on Neural Networks, IJCNN 2017*, Anchorage, AK, May 14–19 (2017), pp. 4095–4102
25. J.-J. Quisquater, D. Samyde, Electromagnetic analysis (EMA): measures and counter-measures for smart cards, in *Smart Card Programming and Security*, ed. by I. Attali, T. Jensen (Springer, Berlin, 2001), pp. 200–210
26. N. Samwel, L. Batina, Practical fault injection on deterministic signatures: the case of EdDSA, in *Progress in Cryptology – AFRICACRYPT 2018*, ed. by A. Joux, A. Nitaj, T. Rachidi (Springer, Cham, 2018), pp. 306–321