TU Delft

Delft University of Technology

On the Importance of Initial Solutions Selection in Fault Injection

Krcek, Marina; Fronte, Daniele ; Picek, Stjepan

DOI
10.1109/FDTC53659.2021.00011

Publication date
2021

Document Version
Accepted author manuscript

Published in
2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)

Citation (APA)
Krcek, M., Fronte, D., & Picek, S. (2021). On the Importance of Initial Solutions Selection in Fault Injection. In M. O'Dell (Ed.), *2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC): Proceedings* (pp. 1-12). Article 9565588 IEEE. https://doi.org/10.1109/FDTC53659.2021.00011

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

On the Importance of Initial Solutions Selection in Fault Injection

Marina Krček
Delft University of Technology
Delft, The Netherlands
m.krcek@tudelft.nl

Daniele Fronte
STMicroelectronics
Rousset, France
daniele.fronte@st.com

Stjepan Picek
Delft University of Technology
Delft, The Netherlands
s.picek@tudelft.nl

***Abstract*—Fault injection attacks require the adversary to select suitable parameters for the attack. In this work, we consider laser fault injection and parameters like the location of the laser shot (x, y), delay, pulse width, and intensity of the laser. The parameter selection process can be translated into an optimization problem. A very popular and successful method for various optimization problems is the genetic algorithm. To further improve the performance of a genetic algorithm, it is possible to combine it with local search to obtain a memetic algorithm.**

We conduct several experiments comparing the performance of the memetic algorithm and the random search algorithm for finding faults. We investigate the influence of different initialization techniques on the performance of the memetic algorithm. In our experiments, the memetic algorithm is significantly better at finding faults than the random search. While evaluating different initialization techniques, we did not observe significant differences when averaging results. However, when considering the stability of the results with a memetic algorithm based on different initialization techniques, we can distinguish preferable techniques, such as LHSMDU and the Taguchi method.

***Keywords*-Laser fault injection; Genetic algorithm; Memetic algorithm; Initialization methods**

I. Introduction

Many hardware devices are used daily by millions of users, promising secure transactions. Security analysts evaluate the security of these hardware devices, and as adversaries, they have numerous possibilities of attacking them. Passive techniques exist, such as side-channel attacks, where the attacker monitors side-channel (unintended) information. Examples of the side-channel information are power consumption [1], electromagnetic radiation [2], and time [3]. On the other hand, there are also active attacks, specifically fault injection attacks, where the attacker aims to extract some secret/sensitive information from the target device by exposing the device to external interference. Doing this can cause the device to deviate from the typical behavior and correct execution, which the attacker might exploit, as in differential fault analysis (DFA) [4], fault sensitivity analysis (FSA) [5], and statistical fault attack (SFA) [6]. Attacks, such as ineffective fault attacks (IFA) [7], and statistical ineffective fault attacks (SIFA) [8], can exploit information even if induced faults do not change the output of the execution on the device. The attacks can be divided into two steps - inducing the faults and analyzing the target device's behavior and responses to achieve the attacker's goal. The focus of this work is on the first part - inducing the faults using external interferences that can come from different sources, such as optical (laser pulses) [9], electrical glitches (voltage) [10], electromagnetic (EM) radiation [11], and temperature changes [12].

The fault injection attacks can be divided by their invasiveness. Invasive attacks remove layers of the target to reach the silicon layer. In semi-invasive attacks, only the packaging is removed, and in non-invasive attacks, the target is attacked as it is, without any modifications. In this work, we consider the optical fault injection technique introduced in [9], namely Laser Fault Injection (LFI) attacks that are semi-invasive attacks. Like other fault injection attacks, for LFI to be successful, the adversary or security analyst must carefully select and tune the attack. In the case of the LFI attacks, the adversary needs to choose good parameters for successfully injecting faults, such as the location of the laser shot (x, y), focus, laser spot size, wavelength, laser trigger delay, laser pulse width, and laser intensity. These parameters are limited by the physical properties of the target and the equipment used for performing the attack. For example, x and y are limited by the target device and motorized stage (the minimum step it can make for each axis). On the other hand, the laser parameters are limited by the laser source equipment. In this work, we aim at injecting faults that lead to exploitable faulty outputs for attacks such as DFA. However, the same parameters can be considered for inducing so-called ineffective faults used in, e.g., SIFA.

The search space is often too large to perform an exhaustive search in a reasonable time, considering all the parameters and their possible values for laser fault injection. The attacker could decrease the ranges for the parameters, but this requires significant knowledge about the internal design of the target. Unfortunately, this is not often the case as the attacks are usually performed in a black-box setting. Consequently, limiting the ranges of the parameters can lead to testing many inadequate parameter combinations or failing to test parameter combinations that would be successful. Additionally, the laser effects on the target behavior can significantly vary between two physically close positions, representing a problem to find optimal parameter sets to perform the laser attacks.

There is a clear need for a "smart" technique for parameter selection leading to a more effective security evaluation,

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works

considering the mentioned issues. The problem of selecting suitable fault injection parameters can be easily translated into an optimization problem, for which we can apply various techniques. One example would be evolutionary algorithms (EAs), such as genetic algorithms (GAs). There are several papers where evolutionary techniques are used to optimize fault injection attacks. In [13], the authors investigated a new direction based on genetic algorithms for voltage (VCC) glitching and found it suitable when not much is known about the device under attack. A continuation of this work is done in [14] where the authors compared the Monte Carlo search with the genetic algorithm. Picek et al. also compared the standard GA to the performance of an enhanced GA called a memetic algorithm (MA) for voltage glitching [15]. The memetic algorithm iterates the standard process of GA but adds a local search. The memetic algorithm is also used by Maldini et al. for Electromagnetic Fault Injection (EMFI) [16]. Another interesting approach using artificial intelligence techniques, namely machine learning, was introduced in [17]. The authors presented a method for fast characterization of the laser fault injection settings on the target. However, the technique can be used for other semi-invasive fault injection attacks and is transferable to different samples of the same target.

This work applies a genetic algorithm with Hooke-Jeeves as the local search to select laser fault injection parameters. The algorithm begins with an initial population of solutions that is evolved through a number of generations to reach better solutions for the optimization problem. A well-selected initial population of solutions is crucial because it can cause the algorithm to converge to some local optima quickly or cover more search space, which could help reach the global optimum. After comparing the performance of the memetic algorithm to the Monte Carlo approach for the laser fault injection attacks, we experiment with several different initialization methods for creating the initial population of solutions for the memetic algorithm.

Our main contributions are as follows:

1) We use evolutionary algorithms (genetic algorithms) to find parameters leading to a successful laser fault injection attack. To the best of our knowledge, this is the first time that evolutionary algorithms are used for LFI. Differing from the discussion in [17], we do not observe genetic algorithms causing issues due to their exploratory character (i.e., damaging the target). Additionally, the laser effect may greatly vary between two very close positions that are of different nature. Evolutionary algorithms work well with non-differentiable functions, making them a natural choice for such settings.
2) We propose a memetic algorithm that combines genetic algorithm and Hooke-Jeeves local search. In our opinion, this improves over related works [15], [16] with memetic algorithms as Hooke-Jeeves is a well understood and efficient derivative-free optimization algorithm.
3) We are the first to explore the influence of the initial population on the performance of the population-based search algorithms for fault injection attacks.

II. Background

This section briefly explains the memetic algorithm and different initialization techniques for the initial population we use in our experiments.

A. Memetic Algorithm (MA)

The memetic algorithm (MA) is a population-based hybrid genetic algorithm enhanced with an added procedure performing local refinements on individuals from the population [18]. Such algorithms were applied in many different fields, such as business analytics and data science [19], designing and training artificial neural networks [20], and robotic motion planning [21]. The memetic algorithms have shown to be better than traditional GAs for some optimization problem domains [22], [23]. The reason could be the trade-off between the exploration abilities of the underlying GA and the exploitation abilities of the local search. The improvement cost is more fitness evaluations and a fast loss of diversity within the population.

Throughout the years, there have been many versions of memetic algorithms proposed [24]. However, our work considers a simple combination of the traditional genetic algorithm and Hooke-Jeeves local search. The pseudocode of the memetic algorithm can be seen in Algorithm 1. The pseudocode is general, and it shows how the local search is integrated into the genetic algorithm - after recombination of the genetic algorithm, we perform local refinements on some of the solutions from the population. To further explain the pseudocode and memetic algorithm, we describe the genetic algorithm and our chosen local search approach - the Hooke-Jeeves algorithm.

Algorithm 1 Memetic Algorithm.

```
1:  procedure MEMETICALGORITHM
2:      generate population of size N
3:      evaluate the population
4:      iteration = 0
5:      while stop condition not satisfied do
6:          new population = best solutions (elite size)
7:          select parent solutions for crossover
8:          children = crossover(parents)
9:          mutation(children)
10:         add children to new population
11:         evaluate the new population
12:         perform the local search
13:         iteration = iteration + 1
14:     return population
```

1) Genetic Algorithm: The genetic algorithm (GA) is the most widely known type of evolutionary algorithm inspired by the process of natural selection [25], [26]. Genetic algorithms are often used for optimization problems [27].

Genetic algorithms work on a population of solutions of size N. In the early application, solutions were represented by bit strings, but representation can be adjusted for many different problems nowadays. For example, considering the LFI problem, the solution can be an array of integer and floating-point values representing the parameter set of LFI, namely the $x, y, delay, pulse\ width$, and $laser\ intensity$ parameters. A set of such solutions constructs the population for the genetic algorithm. After the initial population is created, the solutions are evaluated to retrieve the fitness. Fitness is a value that gives information about how 'good' or 'bad' the solution is for the problem (the definition of 'good' or 'bad' are problem-specific). In simple cases of finding an optimum of well-defined functions, the evaluation retrieves the function result based on the input, which is one solution from the population. There, the function result is the fitness value. Evaluation and fitness values are adjusted based on the problem, as is the representation of the solutions. For the LFI problem, we explain our definitions in Section III.

After evaluating all the solutions in the population, we perform selection, which is usually done based on the fitness of the solutions. The idea is to choose better solutions as parents for reproduction since these solutions should have 'good' genes (solution values) to propagate into the next generation. The reproduction is done using the crossover and mutation operators by combining the genes of the selected parents and introducing minor variations. The crossover is applied to the selected parent solutions resulting in offspring solutions. There are different versions of the crossover operator, such as uniform crossover or average crossover. However, all these variations combine the genes from the parent solutions to produce a new solution (or several solutions). The new offspring solutions undergo mutation where each gene in the solution is modified with probability p_m. These variations help in the exploration of the search space and prevent fast convergence to a local optimum. The resulting intermediary population forms the next generation replacing the previous one entirely. However, complete individuals of the prior generation are transferred to the new generation without modifications when using elitism. Usually, a parameter often called the elite size defines how many solutions are transferred unmodified for the next generation.

Since we use the memetic algorithm, we select solutions that should undergo the individual improvement procedure after the described GA part. For the local improvements, we use the Hooke-Jeeves algorithm described in Section II-A2. We perform the local refinement on each of the selected solutions, and the algorithm starts a new iteration by evaluating the new solutions before parent selection. The process is repeated until the stop (termination) condition is met. This condition can consider different information about the progress of the algorithm. However, often simply the number of iterations, evaluations, or generations is considered. Further, one can consider the fitness values of the solutions - if we find a satisfying fitness, we could terminate the algorithm.

2) Hooke-Jeeves Algorithm: Hooke-Jeeves is a direct search algorithm used to search for the optimum of a nonlinear function without requiring derivatives of the function [28]. The algorithm combines exploratory moves with pattern moves. The step size can be different for each coordinate direction and change during the search in the exploratory move. The exploration starts from the initial point by exploring each coordinate direction by the specified step size. If the fitness does not improve, the opposite direction is considered. After all the coordinates are investigated, the exploration move is completed. If the exploration found a better solution, it is followed by the pattern move. Otherwise, only the step is decreased to try the exploration move again. The pattern move calculates the direction of the improvement and moves the starting point in that direction. The pattern move can be calculated as:

$$x_p{}^{k+1} = x^k + (x^k - x^{k-1}),$$

where $x_p{}^{k+1}$ is the temporary base point for a new exploratory move, x^k is the result of the exploratory move, and x^{k-1} is the previous base point. The algorithm ends when the step size cannot be reduced anymore.

B. Initialization Techniques for the Initial Population

The genetic algorithm requires an initial set of solutions called the population to start the process of evolution. In this work, we investigate different initialization techniques for the initial population of solutions. The initial population, if well-distributed, could lead to better performance of the genetic algorithm [29], [30], [31]. Usually, the initialization is done using Monte Carlo simulation, taking random values for each gene of the solution [32], [33], [16]. We additionally explore the Latin Hypercube Sampling (LHS) and a Taguchi method.

Next, we explain Latin squares and Orthogonal arrays necessary to understand the LHS and Taguchi method, respectively.

Latin Squares: A Latin square is an $n \times n$ array where each of the n different symbols occurs exactly once in each row and each column [34]. Table I gives an example of a Latin square with n equal to three.

TABLE I: An example of a Latin square with $n = 3$.

1	2	3
3	1	2
2	3	1

A Latin hypercube is a generalization of a Latin square concept from two dimensions to an arbitrary number of dimensions.

Orthogonal Arrays: Orthogonal arrays generalize the idea of mutually orthogonal Latin squares and are used in the statistical design of experiments [35], cryptography [36], and software testing [37], [38]. An orthogonal array is an array whose elements come from a fixed finite set of symbols arranged in such a way that for every selection of t columns of the table, all ordered t-tuples of the symbols, formed by taking

the elements in each row restricted to these columns, appear the same number of times [39]. An example of an orthogonal array is shown in Table II.

Definition 1 (Orthogonal array [39]): An $N \times k$ array A with entries from S is said to be an orthogonal array with s levels, strength t and index λ (for some t in the range $0 \leq t \leq k$) if every $N \times t$ subarray of A contains each t-tuple based on S exactly λ times as a row.

N, k, s, t, and λ are the parameters of the orthogonal array, denoted as $OA(N, k, s, t)$[1]. We can omit the λ as the other parameters determine it.

TABLE II: An example of an orthogonal array of strength two. The four ordered pairs (2-tuples) formed by the rows of the first and third columns, namely (1,1), (2,1), (1,2), and (2,2), are all the possible ordered pairs of the two-element set, and each appears exactly once. The same would hold with other combinations of two columns.

1	1	1
2	2	1
1	2	2
2	1	2

Following are short descriptions of Latin Hypercube Sampling and Taguchi method techniques.

Latin Hypercube Sampling: The traditional technique for generating samples of parameter values is Monte Carlo simulation (MCS) that randomly samples the cumulative distributions to obtain the samples. The Latin hypercube sampling (LHS) is a technique emphasizing uniform sampling of the univariate distributions [40], [41], [42]. LHS accomplishes this by stratifying the cumulative distribution function and randomly sampling within the strata. Uniform sampling increases realization efficiency while randomizing within the strata prevents introducing a bias and avoids the extreme value effect associated with regular stratified sampling. It has been demonstrated for many applications that LHS is a more efficient sampling method compared to MCS [43].

This work also considers the Latin hypercube sampling with multidimensional uniformity (LHSMDU) [44]. This method increases the multidimensional uniformity of a sampling matrix by increasing the statistical distance between realizations. The most closely related modification of Latin hypercube sampling is the *maximinLHS* algorithm proposed by Johnson et al. [45]. The LHSMDU uses MCS to generate many realization inputs and sequentially eliminate realizations near each other in the multidimensional space. The distributed realizations are then post-processed to enforce univariate uniformity.

Taguchi Methods: Taguchi methods are statistical methods initially developed to improve the quality of manufactured goods [46], but now these methods are applied in various areas, such as engineering [47] and marketing [48]. Taguchi method presents an experimental strategy utilizing a modified and standardized form of experiment design. Additionally, it provides tools to analyze the results of the experiments to determine the design solution producing the best quality. Two primary tools used in the Taguchi method are the orthogonal arrays to design the experiments and the signal-to-noise ratio to analyze them.

The design of an experiment involves several steps - first, selecting independent variables (factors) and the number of levels for each variable. Then, the user selects the orthogonal array and assigns the variables to each of the columns. After conducting all the experiments, the user analyzes the data to find the best values for the variables. In our work, since we use the memetic algorithm and the Taguchi method to reach better distribution of tested parameter values in the initial phase, we do not conduct the Taguchi recommended analysis using a signal-to-noise ratio. Therefore, we only focus on using orthogonal arrays to create the initial set of solutions for the memetic algorithm.

III. Implementation

We previously explained the memetic algorithm in a general setting, but here we describe our implementation of the algorithm in detail. Explanations follow the flow of the algorithm.

A. Solution Representation and Initialization Methods

As discussed, the memetic algorithm works on a set of solutions (population), improving it to reach the optimal solution for the given problem. For the LFI problem, the solutions are arrays representing the LFI parameters - $x, y, delay, pulse\ width$, and $laser\ intensity$. Bounds for these parameters are user-defined by setting values for the minimum and maximum value of the parameter and the allowed step.

In our experiments, we evaluate several initialization techniques for the initial solutions of the population. First, we use random initialization. That is a Monte Carlo sampling, where parameter values are taken randomly for each solution, with each value having the same probability of being selected. We do not allow duplicate solutions in the population to enable the random initialization to cover more search space. Additionally, identical solutions do not provide more information, and the laser shots would be wasteful.

The second initialization is using the Latin Hypercube Sampling. We use the existing Python package called *pyDOE2*[2]. There are multiple options on how to sample the points, and we use the default one, which randomizes the points within the intervals. The number of samples defines the number of intervals. The LHS method requires the number of factors and the number of samples to be defined, which, in our case, are the number of LFI parameters (five) and the population size, respectively. Another similar method is Latin hypercube sampling with multidimensional uniformity (LHSMDU) from the Python package *lhsmdu*[3]. This technique should improve the sampling for more dimensions than two, compared to the previously described LHS.

[1]This notation is not universally accepted.

[2]https://pypi.org/project/pyDOE2/

[3]https://pypi.org/project/lhsmdu/

Lastly, we use the Taguchi method by utilizing the Orthogonal Array (OA) package[4], which contains functionality to generate and analyze these types of designs. For generating the arrays and designs, the package uses the exhaustive enumeration algorithm from [49] and the optimization algorithm from [50].

B. Evaluation and Fitness Values

After the initialization of the population, the solutions need to be evaluated. This is done by executing the laser shots with the given parameters. We include a sorting algorithm since the physical operations needed to adjust the laser bench for the laser shot are time-consuming, and the most expensive operations are the motorized stage movements. The motorized stage is used to change the location of the laser shot (parameters x and y). We sort the x and y coordinates using a greedy algorithm with the Manhattan distance between the points as a metric. We start at the lowest x and lowest y coordinate and look for the nearest (x, y) coordinates based on the Manhattan distance. Greedy algorithms provide a combination of the best local choices, which does not guarantee the best global solution, or in our case, the shortest path through all the points. Nevertheless, greedy algorithms are helpful because they are fast and often give good approximations of the optimum.

The evaluation consists of conducting the laser shot several times for all the solutions. Each device response is categorized into one fault class - *pass*, *mute*, *changing*, or *fail*. If the device does not respond in a given time, the response is categorized as *mute*. If the response of the device is expected, it is categorized as *pass*, otherwise as a *fail*. If we get different classes with multiple measurements, the response is categorized as the *changing* class.

We need to have a fitness value for each fault class for the memetic algorithm to distinguish the solutions based on their quality. The fitness of the *changing* class is calculated as $\frac{f_P \cdot N_P + f_M \cdot N_M + f_F \cdot N_F}{N_P + N_M + N_F}$, where f_P, f_M, f_F represent the fitness values for fault class *pass*, *mute*, and *fail*, respectively, and N_P, N_M, and N_F the number of the *pass*, *mute*, and *fail* class occurrences in the number of measurements times. Therefore, the denominator, the sum of N_P, N_M, and N_F is equal to the number of measurements per parameter set. The fitness values for other fault classes are shown in Table III. The values are taken from [16] to present the preference of the fault classes and their relation. Since we are trying to maximize the fitness of the solutions, we set the fitness value for the *fail* class the highest followed by the *mute* and *pass* response. Accordingly, the *changing* class with *mute* and *fail* classes results in a higher fitness value than a combination of *mute* and *pass* classes.

TABLE III: Fitness values for *pass*, *mute*, and *fail* fault classes.

Fault class	Fitness value
PASS	2
MUTE	5
FAIL	10

[4]https://pypi.org/project/OApackage/

C. GA Operators

After the evaluation, the genetic algorithm part starts creating a new population using the GA operators. For the selection operator, we use a roulette wheel (fitness proportionate) selection. The fitness function assigns a fitness to possible solutions, and it is used to associate a probability of selection with each solution. Let f_i be the fitness of an individual i in the population, then its probability of being selected is $p_i = \frac{f_i}{\Sigma_{j=1}^{N} f_j}$, where N is the number of individuals in the population.

For recombination of the gene material from the selected parents, we use an average crossover. A child solution is created by taking the average value of the parents' values for every gene (parameter). Next is the uniform mutation, where a new random value can replace each gene (parameter) in the solution with probability p_m. We iterate this process until the whole population of size N is again created. However, we also use elitism, which means that we keep the best solutions in the next generation of the population without modifications. The elite size parameter defines the number of those solutions. Therefore, we generate $N - elite\ size$ new solutions for the new population. New solutions are evaluated before conducting the local search.

D. Local Search

We use the Hooke-Jeeves algorithm for the local search. Only solutions with a fitness larger than 85% of the maximum fitness value (fitness of the *fail* class - 10) are considered. From these solutions, at most, three solutions are randomly selected for the local search. Considering the conditions, the number of solutions taken for local search can vary from zero to three. In our first experiments, we performed a local search on all the solutions with good-enough fitness. This proved to be too many solutions going through the local search because as the population improved, more and more solutions satisfied the requirement. Thus, we limited it to only three randomly selected solutions that fulfill the condition on the fitness values. The steps of the parameter values are defined with their bounds. We only allow two iterations for the exploration phase of the Hooke-Jeeves algorithm, meaning we start with the step doubles, and if no improvements are found, we will only have one more exploration phase with the decreased step size. Again, in our initial testing, we allowed more exploration phases, where we started the local search with the larger step size. It showed too much exploitation of local space, neglecting the exploration of the search space. Regarding the nature of implemented local search as described in Section II-A2, the number of evaluations can vary during the execution of the local search. Therefore, the number of total tested parameter sets can vary with each execution of the memetic algorithm.

E. Termination Condition

Both the genetic algorithm part and local search part constitute one iteration of the memetic algorithm. The termination of the algorithm is done according to the stopping condition.

In our experiments, the stopping condition considers only the number of iterations - when the maximum number of iterations is reached, the algorithm is terminated. The user sets the maximum number of iterations as it is the parameter of the memetic algorithm.

The source code is published on GitHub as a public repository[5]. The repository only holds the implementation of the memetic algorithm. To run the code with a laser bench and setup, one needs to integrate the algorithm to the source code controlling the bench. This can be done by replacing the placeholders in the implementation with the real bench connections and functions.

IV. Experimental Setup and Results

A. Experimental Setup

We performed all the experiments on the same target, a test chip from STMicroelectronics. Considering the company policies, we may disclose only a portion of the confidential information about the target device and the utilized laser bench. The target chip is made using 40nm technology, and for our experiments, we ran an implementation in C programming language displayed in Pseudocode 1. The code running on the target device is a test program where data words are copied (loaded) from the non-volatile memory (NVM) into a register. The *trigger_event* function is a monitored event that is used to inject faults at the desired time - on loading a data word into a register (marked as a comment in Pseudocode 1). After the fault injection, the register is read, and there is a fault if the register value has changed (fault class *fail*).

Pseudocode 1: Pseudocode of the implementation running on the target device.

```
...
trigger_event()
load_register() // injection here
read_register()
...
```

Regarding the laser bench and the parameters, we adopted an infrared laser wavelength, and for each of the five parameters, we used a subset of the available values. We defined the subsets according to the previous knowledge of the target device. The same subsets for all the parameters were applied in all the experiments. We note there are 31 737 600 possible combinations considering the parameter values we use. Each test case is repeated five times for a better statistical representation of the results. We run an online optimization where the results are based on how much *fail* classes are found in a specific number of tested parameters instead of finding only one best parameter set (recall, there is no single best solution since all *fail* solutions are equally good and there can be multiple distinct regions of parameter values that can lead to *fail* classes), so repeating the same setting five times gives us sufficient information. We note that having a small number of experimental runs is more common for online optimization, e.g., in the case of evolutionary fuzzing [51]. Also, we perform the laser shot for each parameter set five times (number of measurements).

[5]https://github.com/AISyLab/memetic_alg_for_laser_FI

First, we compare the memetic algorithm with a simple random search of the LFI parameters. The random search will test a defined number of parameter sets allowing unique combinations. We test 9 130 different parameter sets and compare the results with the memetic algorithm with the random initialization technique. Running each setup five times, we average the number of total parameter sets tested and the fault classes' percentages. As already mentioned, as the local search in the memetic algorithm is not deterministic in the number of evaluations, the number of tested parameters varies, which is not the case with the random search.

Concerning the memetic algorithm parameters, we use the population size of 36 and the elite size of 2. We use a population size of 36 because we created a mixed-level orthogonal array with 36 samples having three levels for the x, y, and $delay$ parameters and two levels for the $pulse\ width$ and $intensity$. The number of iterations (200) and mutation probability (0.05) are the same throughout all the experiments. Since we have an expensive evaluation of the solutions, we want to keep the number of iterations low, but we chose to use 200 iterations since our population size is small. The mutation probability is low because a high mutation probability can cause the algorithm to behave like a random search. All these parameters influence the algorithm's performance, and we do not claim these values to be optimal. The tuning of these parameters could be further investigated. However, while testing the setup for the experiments, these parameters led to good convergence of the memetic algorithm and were kept for comparison with random search and different initialization methods.

In the experiments with different initialization methods for the initial population of the memetic algorithm, we test with a smaller population size of 36 with elite size 2, and then a larger population of size 128 and elite size 10.

B. Results

In Table IV, with the memetic algorithm, we find significantly more *fail* classes than with the random search having tested 9 034 parameter sets on average (over five runs). To be more precise, we find ≈ 30 times more *fail* classes, and ≈ 3 times more *changing* classes in a similar number of tested parameters (9 130 for the random search, and 9 034 for the memetic algorithm). We conclude that we benefit from using memetic algorithms instead of a simple random search when the exhaustive search is not feasible. An exhaustive search in our case with reduced parameter bounds would take ≈ 59 days if we ran each parameter set once and each evaluation lasted $\approx 0.16s$. However, if we do not reduce the parameter bounds, the exhaustive search could take years. On the other hand, one run of the memetic algorithm in our case was around two hours, with 9 034 parameter sets tested five times.

TABLE IV: Results on average for random search algorithm and memetic algorithm with random initialization of the population of size 36.

Algorithm	Tested parameters	Fails (%)	Changing (%)	Mute (%)	Pass (%)
Random	9 130.0	0.90	2.58	2.54	93.98
Memetic	9 034.6	31.03	7.11	3.64	58.22

To further improve the performance of the memetic algorithm, we test different initialization methods for generating the initial set of solutions. Thus, instead of generating only random initial solutions, we use techniques that try to distribute the samples taken from the allowed values over the search space and cover as many combinations as possible, considering the levels given to each parameter.

We compare four initialization methods, random initialization, LHS, LHSMDU, and the Taguchi method. For the Taguchi method, we use the orthogonal array of 36 samples, with strength two, and factor levels $3, 3, 3, 2, 2$ for the variables $x, y, delay, pulse\ width$, and $intensity$, respectively. The array can be seen in Table VII in Appendix A. We use the same population size (36) and elite size (2) as the first experiment with the memetic algorithm.

With the memetic algorithm, we are trying to maximize the fitness function. Therefore, we prefer laser injection to result in a corrupt register value which we categorize as the *fail* class. Looking at the averaged results from Table V, the difference between the initialization techniques is not significant. Furthermore, because of the variance in the number of tested parameters, we see that the LHSMDU technique has the largest number of tested parameters and the highest percentage of *fail* classes. We conclude that all the initialization techniques, on average, result in similar results. Therefore, if we have time and can use the average results, we might not need to consider using some of the proposed techniques to improve the sampling for the initial population compared to the random sampling.

TABLE V: Results on average for all initialization techniques in experiments with population size 36.

Initialization	Tested parameters	Fails (%)	Changing (%)	Mute (%)	Pass (%)
Random	9 034.6	31.03	7.11	3.64	58.22
LHS	8 812.8	28.80	7.73	4.59	58.88
LHS-MDU	9 605.0	33.92	7.67	3.02	55.40
Taguchi	9 070.2	31.58	6.87	3.33	58.23

However, we also looked into individual results as we are interested in the variation of the results. In Figure 1, we show percentages of different fault classes for settings with all the initialization techniques. For each initialization technique, we show the percentage for each experiment to see the variation and mark the average percentage with an X. What can be seen is that even though with the random initialization of the population we can get excellent results, we can also end with the worst results considering the percentage of *fail* classes (which we prefer) in Figure 1a. Similarly, using the LHSMDU technique, we had the lowest variation, but the results did not reach as good results as random initialization and Taguchi in one of the experiments.

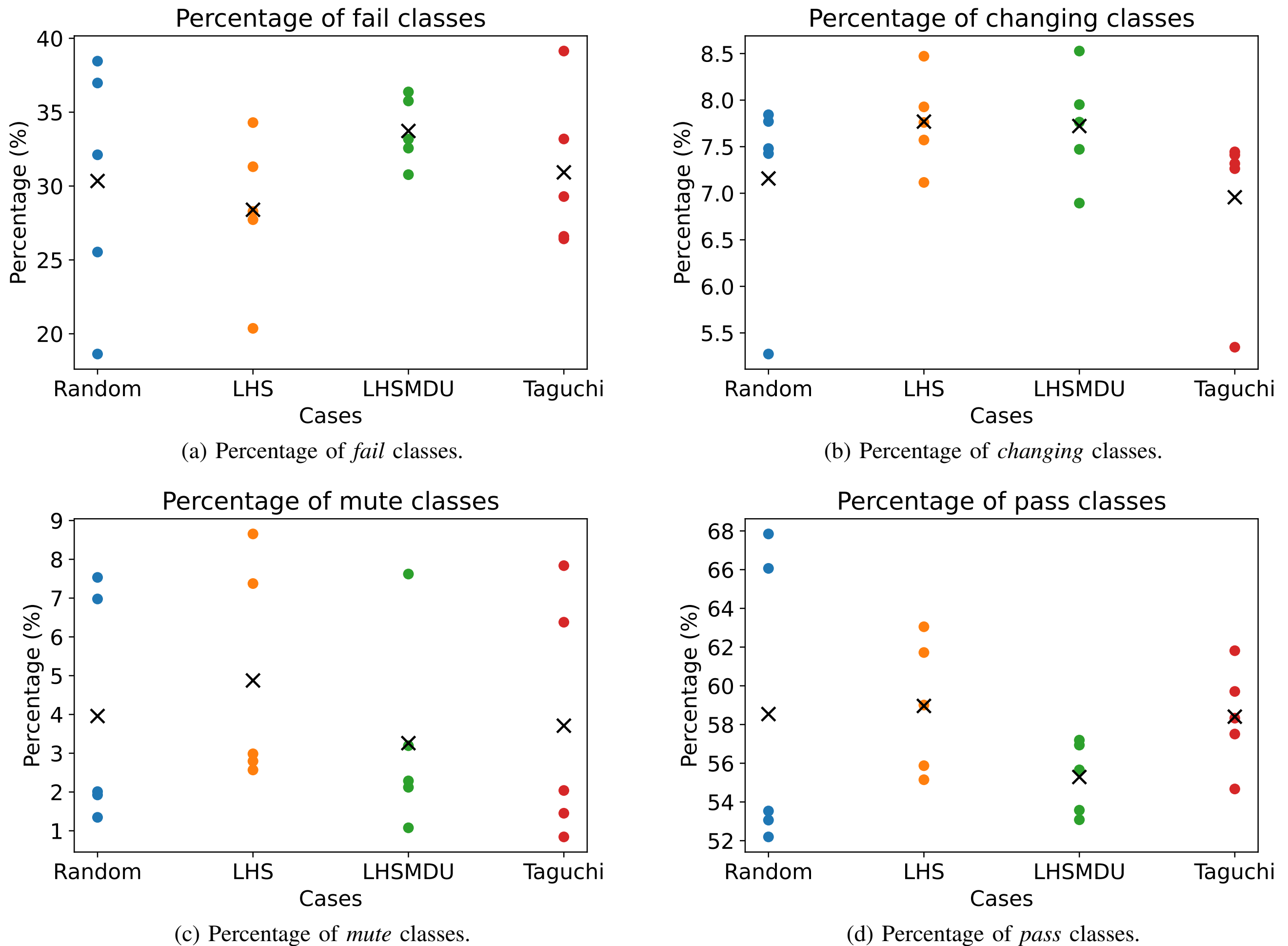
Looking at the number of tested parameters in Figure 2a, we observe that, in the case with a smaller population, the ranges among the different test cases are similar. With the LHSMDU, we had a more consistent number of tested parameters, with only one outlier. By combining the percentages of all the fault classes and the number of tested parameters, we see that the stability of the results is not coming directly from the number of tested parameters. *Having tested more parameters does not necessarily lead to finding more interesting parameter sets.* For example, in the test case with the LHSMDU technique, we often tested more parameters than random initialization. However, the percentage of *fail* classes is larger with random initialization than LHSMDU in some cases. Considering the best results on the number of found *fail* classes and the stability of the results (consistency), we can see that LHSMDU is the best choice, followed by the Taguchi method.

Next, we increase the population size to 128 and elite size to ten and run the same experiments. We generate an orthogonal array of 128 samples for the Taguchi method, with strength two and factor level two for all the variables. The array can be seen in Table VIII in Appendix A.

Table VI gives the averaged results, where we observe that, similarly, the experiments show, on average, no initialization technique is significantly better. Again the largest percentage of *fail* classes was found by the test case with the largest number of tested parameters. Here, that was the Taguchi method. The percentage of the *fail* classes is lower in these experiments with a larger population because only a limited number of *fail* classes can occur with our target and test program. We test more unique parameter sets since we have a larger population but the same number of iterations. The more parameters we test, the closer we get to the exhaustive search, which would only show the true distribution of the fault classes. However, we did not run the exhaustive search because, as already mentioned, it would take ≈ 59 days to run it with our reduced intervals if we ran each parameter set only once and each laser shot takes $\approx 0.16s$.

Looking at the number of tested parameters with the larger population in Figure 2b, with the Taguchi method, the number became more stable, and the variation is reduced. This is somewhat reflected in the results shown in Figure 3 presenting the variance of the percentages of different fault classes. Again, we see that the test case that tested the most parameters in one of the runs (Taguchi) did not lead to the highest percentage of *fail* classes (random initialization).

We can see that the experiments with Taguchi initialization



(a) Percentage of *fail* classes.

(b) Percentage of *changing* classes.

(c) Percentage of *mute* classes.

(d) Percentage of *pass* classes.

Fig. 1: Percentages of *fail*, *changing*, *mute*, and *pass* classes with all initialization techniques for a setting with a population size of 36 and elite size of 2.

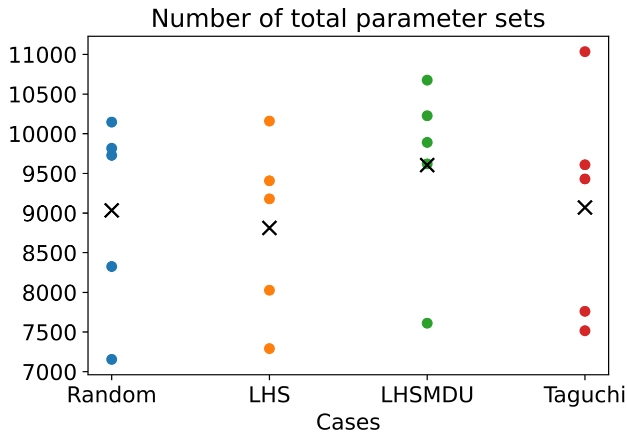
TABLE VI: Results on average for all initialization techniques in experiments with population size 128.

Initialization	**Tested parameters**	**Fails (%)**	**Changing (%)**	**Mute (%)**	**Pass (%)**
Random	23 226.6	22.07	6.55	4.19	67.18
LHS	22 183.0	20.24	6.55	3.42	69.80
LHS-MDU	22 578.8	21.19	6.77	3.32	68.72
Taguchi	24 440.0	22.85	6.99	2.31	67.86

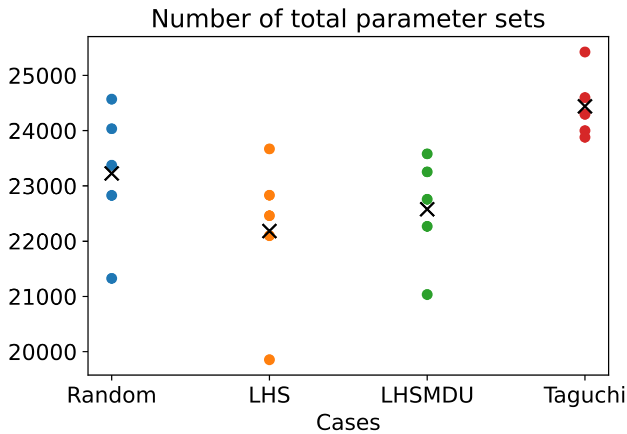
are more stable with a larger population. Considering the results with the LHSMDU initialization, the stability is slightly worse than with a smaller population. The LHSMDU divides the interval for the parameter in the number of samples we want to have, which, in this case, equals 128. For some smaller intervals, this means many values get mapped to the same value, which could be why the performance of this technique has deteriorated with larger population size. On the other hand, the stability of the random initialization technique is improved with a larger population as we do not allow duplicate parameter sets.

Considering the LHS method for initialization, LHSMDU and Taguchi would be preferred over simple LHS for both smaller and larger populations. The weaker performance of this technique could be the fact that we experiment with five parameters. We have a five-dimensional search space making it harder for LHS to reach a better distribution of samples than random sampling.

We conclude that we can benefit from using sampling techniques that produce well-distributed samples, such as the LHSMDU or Taguchi method, to create the initial population. However, there could be more techniques that can provide these characteristics. *Additionally, we observe that performance with random initialization improves with a larger population, so it is more important to have a better sampling method when working with a smaller population.* Moreover, we can see that the difference between the percentage of found



(a) The number of tested parameter sets with population size of 36.



(b) The number of tested parameter sets with population size of 128.

Fig. 2: The number of tested parameter sets in a setting with a smaller population (population size of 36 and elite size of two) in Figure 2a and a larger population (population size of 128 and elite size of ten) in Figure 2b.

fail classes in the worst and best case decreases with the size of the population. In the test cases with a smaller population (Table V), the difference between the worst (LHS) and best case (LHSMDU) is 5.12%, while with the larger population (Table VI), the difference between the worst (LHS) and best case (Taguchi method) is 2.61%.

V. Conclusions and Future Work

This work showed that using the memetic algorithm can greatly improve the results compared to the random search algorithm. While with 9 130 tested parameters, the random search found 0.9% of the *fail* classes, a memetic algorithm with 9 034 tested parameters had 31.03% of *fail* classes. The memetic algorithm used a random initialization of the population, and since the initial population can have a significant impact on the results of the algorithm itself, we tested different techniques for initialization. We experiment with techniques that aim to achieve a well-distributed set of samples for exploring more of the search space and possible combinations: the Latin Hypercube Sampling (LHS) and Taguchi methods. There are two implementations of the LHS. First, the simple one, where we sample from equally probable intervals of the variable range, and another implementation (LHSMDU), which sequentially eliminates realizations near each other in the multidimensional space. On average, in our results, the initialization technique does not significantly influence the performance of the memetic algorithm. However, when we consider all the experiments with the same setup, we see that it is beneficial to use other techniques rather than simple random sampling. With a smaller population size, LHSMDU has excellent stability, with the random initialization being the least stable and less reliable. With a larger population size, Taguchi showed the best results, and it was stable. We recommend using techniques that promote a better distribution of randomly selected samples for initializing the population for the memetic algorithms. It could make the algorithm more robust and avoid fast convergence to a local optimum. Finally, these techniques are shown to be very beneficial when working with a smaller population.

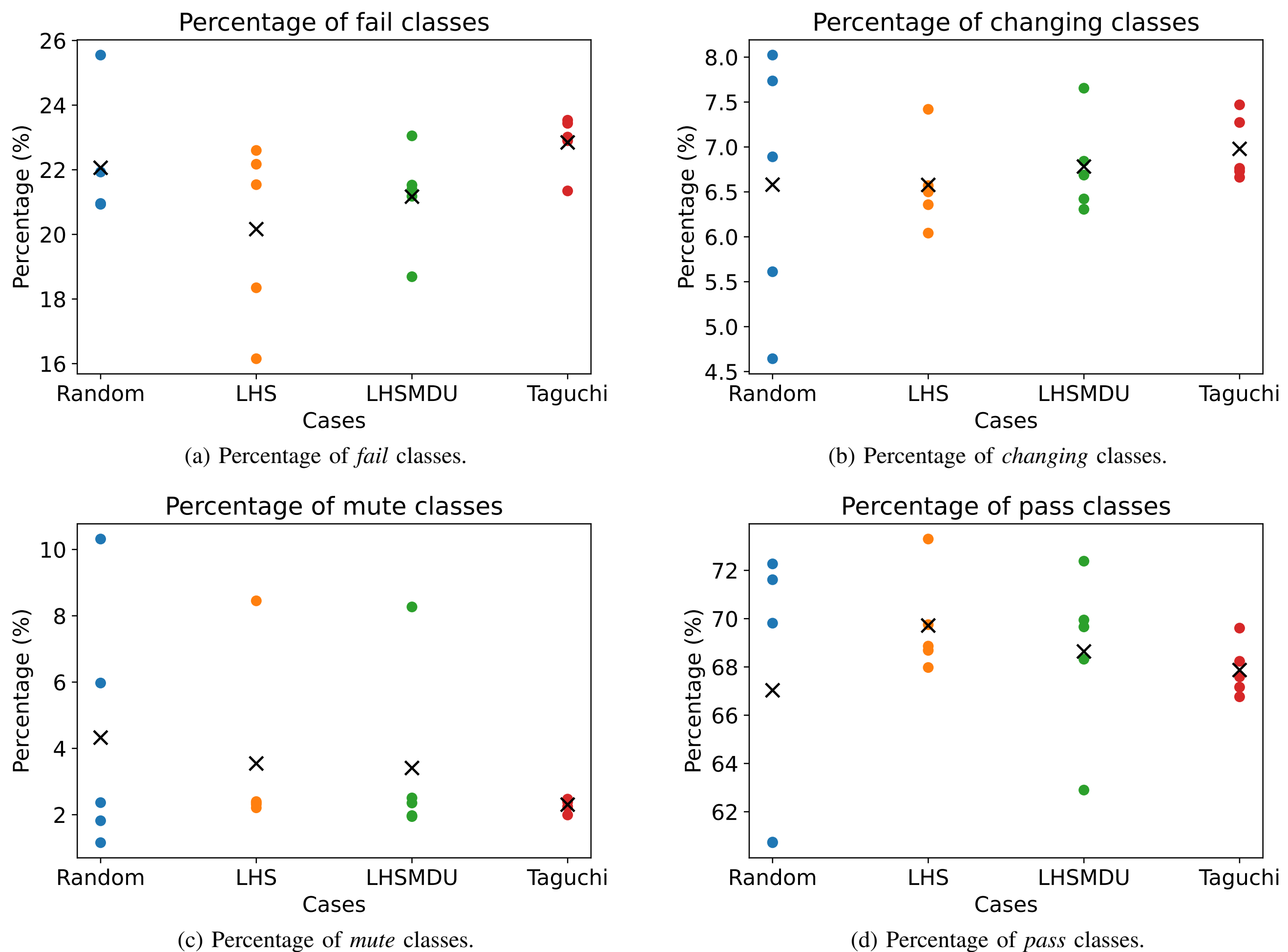
This work shows that memetic algorithms can be very beneficial and efficient in finding faults for laser fault injection attacks. Therefore, it offers excellent potential for improving these attacks. However, since this work is limited to one target sample, we do not consider the evaluation of the memetic algorithm complete. Therefore, we plan to test the algorithm on different targets to test the approach in a more general setting. We also plan to distinguish between exploitable faulty outputs found by memetic algorithms since this work considers only parameter sets that lead to a faulty output (a change in the register value, in our case) that might be exploitable. Therefore, this includes performing attacks such as DFA. Our memetic algorithm can also be tested with different fault injection methods, such as voltage glitching and EMFI. Further, this work can be improved by considering more population sizes as it could benefit some of the initialization techniques. For example, decreasing the population size from 128 could benefit the LHSMDU method. For Taguchi, we can experiment with a larger number of factor levels to help improve the algorithm's performance using this technique. Another interesting research direction could consider which parameters would benefit more from a good distribution in the initial set. Here, we considered all the parameters, but there could be a subset of the parameters that would benefit more than others.

References

[1] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '99. Berlin, Heidelberg: Springer-Verlag, 1999, p. 388–397.

[2] J.-J. Quisquater and D. Samyde, "Electromagnetic analysis (ema): Measures and counter-measures for smart cards," in *Proceedings of the International Conference on Research in Smart Cards: Smart Card Programming and Security*, ser. E-SMART '01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 200–210.

[3] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '96. Berlin, Heidelberg: Springer-Verlag, 1996, p. 104–113.



(a) Percentage of *fail* classes.

(b) Percentage of *changing* classes.

(c) Percentage of *mute* classes.

(d) Percentage of *pass* classes.

Fig. 3: Percentages of *fail*, *changing*, *mute*, and *pass* classes with all initialization techniques for a setting with a population size of 128 and elite size of 10.

[4] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," 1997.

[5] Y. Li, K. Sakiyama, S. Gomisawa, T. Fukunaga, J. Takahashi, and K. Ohta, "Fault sensitivity analysis," in *Cryptographic Hardware and Embedded Systems, CHES 2010*, S. Mangard and F.-X. Standaert, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 320–334.

[6] T. Fuhr, E. Jaulmes, V. Lomné, and A. Thillard, "Fault attacks on aes with faulty ciphertexts only," in *Proceedings of the 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, ser. FDTC '13. USA: IEEE Computer Society, 2013, p. 108–118. [Online]. Available: https://doi.org/10.1109/FDTC.2013.18

[7] C. Clavier, "Secret external encodings do not prevent transient fault analysis," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2007, pp. 181–194.

[8] C. Dobraunig, M. Eichlseder, T. Korak, S. Mangard, F. Mendel, and R. Primas, "Sifa: Exploiting ineffective fault inductions on symmetric cryptography," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, Issue 3, pp. 547–572, 2018. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/7286

[9] S. P. Skorobogatov and R. J. Anderson, "Optical fault induction attacks," in *International workshop on cryptographic hardware and embedded systems*. Springer, 2002, pp. 2–12.

[10] C. H. Kim and J.-J. Quisquater, "Fault attacks for crt based rsa: New attacks, new results, and new countermeasures," in *IFIP International Workshop on Information Security Theory and Practices*. Springer, 2007, pp. 215–228.

[11] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller," in *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2013, pp. 77–88.

[12] M. Hutter and J.-M. Schmidt, "The temperature side channel and heating fault attacks," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2013, pp. 219–235.

[13] R. B. Carpi, S. Picek, L. Batina, F. Menarini, D. Jakobovic, and M. Golub, "Glitch it if you can: parameter search strategies for successful fault injection," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2013, pp. 236–252.

[14] S. Picek, L. Batina, D. Jakobović, and R. B. Carpi, "Evolving genetic algorithms for fault injection attacks," in *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2014, pp. 1106–1111.

[15] S. Picek, L. Batina, P. Buzing, and D. Jakobovic, "Fault injection with a new flavor: Memetic algorithms make a difference," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2015, pp. 159–173.

[16] A. Maldini, N. Samwel, S. Picek, and L. Batina, "Genetic algorithm-based electromagnetic fault injection," in *2018 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2018, pp. 35–42.

[17] L. Wu, G. Ribera, N. Beringuier-Boher, and S. Picek, "A fast characterization method for semi-invasive fault injection attacks," in *Cryptographers' Track at the RSA Conference*. Springer, 2020, pp. 146–170.

[18] P. Moscato, "On evolution, search, optimization, genetic algorithms and martial arts - towards memetic algorithms," *Caltech Concurrent*

Computation Program, 10 2000.

[19] P. Moscato and L. Mathieson, “Memetic algorithms for business analytics and data science: a brief survey,” *Business and consumer analytics: new ideas*, pp. 545–608, 2019. [Online]. Available: https://doi.org/10.1007/978-3-030-06222-4_13

[20] W. Sheng, P. Shan, J. Mao, Y. Zheng, S. Chen, and Z. Wang, “An adaptive memetic algorithm with rank-based mutation for artificial neural network architecture optimization,” *IEEE Access*, vol. PP, pp. 1–1, 09 2017.

[21] P. Rakshit, D. Banerjee, A. Konar, and R. Janarthanan, “An adaptive memetic algorithm for multi-robot path-planning,” in *Swarm, Evolutionary, and Memetic Computing*, B. K. Panigrahi, S. Das, P. N. Suganthan, and P. K. Nanda, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 248–258.

[22] W. E. Hart, “Adaptive global optimization with local search,” Ph.D. dissertation, Citeseer, 1994.

[23] K. Michalak, “Evolutionary algorithm with a directional local search for multiobjective optimization in combinatorial problems,” *Optimization Methods and Software*, vol. 31, no. 2, pp. 392–404, 2016.

[24] X. Chen, Y.-S. Ong, M.-H. Lim, and K. C. Tan, “A multi-facet survey on memetic computation,” *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 5, pp. 591–607, 2011.

[25] A. E. Eiben and J. E. Smith, *Introduction to evolutionary computing*. Springer, 2003, vol. 53.

[26] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press, 1975, second edition, 1992.

[27] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. New York: Addison-Wesley, 1989.

[28] R. Hooke and T. A. Jeeves, "“ direct search” solution of numerical and statistical problems,” *J. ACM*, vol. 8, pp. 212–229, 1961.

[29] H. Maaranen, K. Miettinen, and A. Penttinen, “On initial populations of a genetic algorithm for continuous optimization problems,” *Journal of Global Optimization*, vol. 37, no. 3, p. 405, 2007.

[30] V. Toğan and A. T. Daloğlu, “An improved genetic algorithm with initial population strategy and self-adaptive member grouping,” *Computers & Structures*, vol. 86, no. 11-12, pp. 1204–1218, 2008.

[31] S. Poles, Y. Fu, and E. Rigoni, “The effect of initial population sampling on the convergence of multi-objective genetic algorithms,” in *Multiobjective programming and goal programming*. Springer, 2009, pp. 123–133.

[32] D. Whitley, “A genetic algorithm tutorial,” *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994.

[33] F. H.-F. Leung, H.-K. Lam, S.-H. Ling, and P. K.-S. Tam, “Tuning of the structure and parameters of a neural network using an improved genetic algorithm,” *IEEE Transactions on Neural networks*, vol. 14, no. 1, pp. 79–88, 2003.

[34] K. Kishen, “On latin and hyper-graeco-latin cubes and hyper-cubes,” *Current Science*, vol. 11, no. 3, pp. 98–99, 1942.

[35] C. R. Rao, “Some combinatorial problems of arrays and applications to design of experiments,” in *A Survey of Combinatorial Theory*. Elsevier, 1973, pp. 349–359.

[36] C. Koukouvinos, B. Lappas, and D. Simos, “Encryption schemes using orthogonal arrays,” *Journal of Discrete Mathematical Sciences and Cryptography*, vol. 12, no. 5, pp. 615–628, 2009.

[37] I. S. Dunietz, W. K. Ehrlich, B. Szablak, C. L. Mallows, and A. Iannino, “Applying design of experiments to software testing: experience report,” in *Proceedings of the 19th international conference on Software engineering*, 1997, pp. 205–215.

[38] L. Lazić and D. Velašević, “Applying simulation and design of experiments to the embedded software testing process,” *Software Testing, Verification and Reliability*, vol. 14, no. 4, pp. 257–282, 2004.

[39] A. Hedayat, N. Sloane, and J. Stufken, *Orthogonal Arrays: Theory and Applications*. Springer Science & Business Media, 1999.

[40] P. Audze and V. Eglais, “New approach to planning out of experiments,” *Problems of dynamics and strength (in Russian)*, vol. 35, pp. 104–107, 1977.

[41] R. Iman, J. Helton, and J. Campbell, “An approach to sensitivity analysis of computer models: Part i—introduction, input variable selection and preliminary variable assessment,” *J Qual Technol*, vol. 13, pp. 174–183, 07 1981.

[42] R. L. Iman, J. M. Davenport, and D. K. Zeigler, “Latin hypercube sampling (program user’s guide). [lhc, in fortran],” 1 1980.

[43] M. Mckay, R. Beckman, and W. Conover, “A comparison of three methods for selecting vales of input variables in the analysis of output from a computer code,” *Technometrics*, vol. 21, pp. 239–245, 05 1979.

[44] J. L. Deutsch and C. V. Deutsch, “Latin hypercube sampling with multidimensional uniformity,” *Journal of Statistical Planning and Inference*, vol. 142, no. 3, pp. 763–772, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378375811003776

[45] M. Johnson, L. Moore, and D. Ylvisaker, “Minimax and maximin distance designs,” *Journal of Statistical Planning and Inference*, vol. 26, no. 2, pp. 131–148, 1990. [Online]. Available: https://www.sciencedirect.com/science/article/pii/037837589090122B

[46] G. Taguchi and A. P. Organization, *Introduction to Quality Engineering: Designing Quality Into Products and Processes*. Asian Productivity Organization, 1986. [Online]. Available: https://books.google.hr/books?id=1NtTAAAAMAAJ

[47] G. Taguchi and V. Cariapa, “Taguchi on robust technology development,” 1993.

[48] P. Selden, *Sales Process Engineering: A Personal Workshop*. ASQC Quality Press, 1996. [Online]. Available: https://books.google.nl/books?id=Zqf-AQAACAAJ

[49] E. Schoen, P. Eendebak, and M. Nguyen VM, “Complete enumeration of pure-level and mixed-level orthogonal arrays,” *Journal of Combinatorial Designs*, vol. 18, pp. 123 – 140, 03 2009.

[50] P. T. Eendebak and A. R. Vazquez, “Oapackage: A python package for generation and analysis of orthogonal arrays, optimal designs and conference designs,” *Journal of Open Source Software*, vol. 4, no. 34, p. 1097, 2019. [Online]. Available: https://doi.org/10.21105/joss.01097

[51] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, “Vuzzer: Application-aware evolutionary fuzzing,” in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

APPENDIX

In Table VII, we present the orthogonal array used in the experiments with a population size of 36. Table VIII gives the orthogonal array used in the experiments with a population size of 128.

TABLE VII: The orthogonal array used in the experiments with a population size of 36. This is an array of strength two with the number of samples being 36, and the number of levels, for each factor, equals 3, 3, 3, 2, 2, respectively to the order of columns (LFI parameters).

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	1
0	1	1	0	1
0	1	1	0	1
0	1	1	1	0
0	1	1	1	0
0	2	2	1	0
0	2	2	1	1
0	2	2	1	1
0	2	2	1	1
1	0	1	1	0
1	0	1	1	1
1	0	1	1	1
1	0	1	1	1
1	1	2	0	0
1	1	2	0	0
1	1	2	0	0
1	1	2	0	1
1	2	0	0	1
1	2	0	0	1
1	2	0	1	0
1	2	0	1	0
2	0	2	0	1
2	0	2	0	1
2	0	2	1	0
2	0	2	1	0
2	1	0	1	0
2	1	0	1	1
2	1	0	1	1
2	1	0	1	1
2	2	1	0	0
2	2	1	0	0
2	2	1	0	0
2	2	1	0	1

TABLE VIII: The orthogonal array used in the experiments with a population size of 128. This is an array of strength two with the number of samples being 128, and the number of levels for every factor is two. In this representation, the first column represents the number of times the same combination repeats. There are eight different combinations of the factor levels, and each repeats 16 times.

16x	0	0	0	0	0
16x	0	0	0	1	1
16x	0	1	1	0	0
16x	0	1	1	1	1
16x	1	0	1	0	1
16x	1	0	1	1	0
16x	1	1	0	0	1
16x	1	1	0	1	0