The More You Know
Improving Laser Fault Injection with Prior Knowledge

Krcek, Marina; Ordas, Thomas ; Fronte, Daniele ; Picek, Stjepan

DOI
10.1109/FDTC57191.2022.00012

Publication date
2022

Document Version
Final published version

Published in
Proceedings of the 2022 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)

The More You Know: Improving Laser Fault Injection with Prior Knowledge

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

Thomas Ordas, Daniele Fronte
STMicroelectronics
Rousset, France
{thomas.ordas,daniele.fronte}@st.com

Stjepan Picek
Radboud University
Nijmegen, The Netherlands
stjepan.picek@ru.nl

2022 Workshop on Fault Detection and Tolerance in Cryptography (FDTC) | 978-1-6654-5442-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/FDTC57191.2022.00012

***Abstract*—We consider finding as many faults as possible on the target device in the laser fault injection security evaluation. Since the search space is large, we require efficient search methods. Recently, an evolutionary approach using a memetic algorithm was proposed and shown to find more interesting parameter combinations than random search, which is commonly used. Unfortunately, once a variation on the bench or target is introduced, the process must be repeated to find suitable parameter combinations anew.**

To negate the effect of variation, we propose a novel method combining a memetic algorithm with a machine learning approach called a decision tree. Our approach improves the memetic algorithm by using prior knowledge of the target introduced in the initial phase of the memetic algorithm. In our experiments, the decision tree rules enhance the performance of the memetic algorithm by finding more interesting faults in different samples of the same target. Our approach shows more than two orders of magnitude better performance than random search and up to 60% better performance than previous state-of-the-art results with a memetic algorithm. Another advantage of our approach is human-readable rules, allowing the first insights into the explainability of target characterization for laser fault injection.

***Keywords*-Laser Fault Injection; Decision Tree; Transferability**

I. Introduction

A secure device should be designed so that as little as possible secret information can leak to an attacker. Even if the algorithms (e.g., cryptographic algorithms) running on the device are mathematically secure, it does not mean the attacks are impossible. One well-known and powerful category of the attacks is called the implementation attacks, which aim at the weaknesses of the implementation and not the algorithms themselves. Two common implementation attacks are side-channel attacks (SCAs) and fault injection (FI) attacks. Side-channel attacks are passive, and the attacker tries to obtain some side-channel information from the execution on the hardware, such as time [10], power consumption [9], and electromagnetic radiation [26]. From this information, it is possible to obtain secret information. Fault injection attacks are active attacks where the attacker tries to force the device to make errors during its execution. This way, the adversary could reach some malicious goal, like passing the authentication or obtaining secret information from the target device. Both implementation attacks are prevalent in security evaluations but can be challenging to deploy in practice.

This work focuses on laser fault injections (LFI), as introduced by Skorobogatov et al. [27]. Laser fault injections are very powerful as they provide high precision for injecting faults by producing single-bit faults [6]. However, multiple parameters define the laser injections, such as location on the device (x and y coordinates), distance from the device to the microscope lens, lasers settings like *intensity*, *delay*, and *pulse width*. The attacker's goal is to find the parameters that lead to successful fault injection causing the device to skip instructions, change values in memory, etc. Afterward, the attackers can use various techniques, e.g., differential fault analysis (DFA) [1] to reach their malicious goals.

As mentioned, there are *many parameters* to be defined and *many possibilities* to consider. Consequently, the search space is large, and finding exploitable faults is difficult, making this attack challenging to perform. While an exhaustive search is commonly not reasonable, the location on the target is often searched in a grid-like manner using the same laser settings. The laser settings are usually based on previous experience, which might be lacking if the target or the bench is new. This process is often time-consuming, so a random search is applied as an alternative. However, both approaches could *miss interesting parameter sets* that lead to faults. Additionally, there is a problem with the *transferability of the results* in fault injection attacks [30], [22]. Results obtained with one setup are not necessarily easily reproduced on another, and changing the target can cause changes in the optimal parameters for the injection. Thus, the attackers need to repeat the same process of finding the optimal parameters for every change, increasing the difficulty of a successful FI.

There is a need for better, automated approaches to efficiently search the parameter space for FI. Evolutionary algorithms were proposed for laser fault injections [11], as well as other types of injections such as voltage glitching [4], [21], [20] and electromagnetic fault injection [14] since laser fault injections are not the only type of injections suffering from the previously described issues. Methods applied are either genetic algorithm (GA) or memetic algorithm (MA), which combines a genetic algorithm with local search. Besides evolutionary approach, hyperparameter optimization techniques [29] and reinforcement learning [17] were also investigated. Additionally, for laser injections, machine learning (ML) was used

978-1-6654-5442-1/22/$31.00 ©2022 IEEE
DOI 10.1109/FDTC57191.2022.00012

in the fast characterization method proposed in [30]. The authors generate a sensitivity curve and use neural networks (multilayer perceptron) to predict the target's behavior from the obtained data of the sensitivity curve. The authors also discuss the transferability issue and test their method on different samples of the same target.

In this work, we focus on the security evaluation process. We aim to find as many fault injection settings where the device does not behave as expected. At the same time, we do not consider exploiting the obtained faults with any specific attack. We start from the memetic algorithm for the LFI presented in [11] since it provides state-of-the-art results to the best of our knowledge. Then, we use the concept of prior knowledge to enhance the algorithm performance further and, consequently, find more faults. Our approach combines machine learning and a memetic algorithm where we use decision trees (DTs) to extract knowledge about the target's behavior and use it in the initialization phase of the MA.
Our main contributions are:

- We develop an approach to increase the number of desired outcomes in the initial population (i.e., parameters that lead to faults), which helps the memetic algorithm find more faults with less tested parameters. The improvement happens because the algorithm can distinguish between desired and other outcomes from the first iteration and learn from already found parameter sets. At the same time, the process is more efficient than running a memetic algorithm for more iterations as it can easily get stuck in specific regions of search space. Our results show that we can improve the efficiency of the search process by up to 60%.
- Our approach allows us to tackle the transferability issue. Changing the setup or the target requires repeating exploration of the search space to find faults. However, if the same target type is used, and we only change different samples of the same target while keeping (almost) the same bench setup, an intuitive assumption is that the resulting parameters should be (mostly) transferable. Indeed, minor differences in hardware introduced during the production or preparation of the target for conducting the laser injection (mechanical thinning of the backside silicon substrate) could be negligible for the LFI transferability. We test this assumption using our approach on 1) different samples of the same target and 2) slightly modified bench setups. We observe that prior knowledge is transferable and helps characterize the target more efficiently.
- As the decision tree outputs rules, we provide the initial results on the explainability of fault injection target characterization.

II. Background

A. Decision Tree (DT)

The decision tree is a supervised machine learning technique for classification and regression problems. More specifically, it is a tree-based technique in which any path starting at the root node separates data based on specific criteria in the internal nodes (also called decision nodes) until the outcome in the leaf node is reached. An example of a decision tree is given in Figure 1, where the mentioned nodes are distinguished. We consider a classification problem with two features and three target classes, and this example demonstrates how the conditions can be expressed for categorical and continuous numeric variables. Decision trees are easier to understand than, for example, Artificial Neural Networks (ANNs) because of their tree-like structure that mimics the human decision-making process and the rules generated from these trees. The rules are often structured as *if-then* statements.

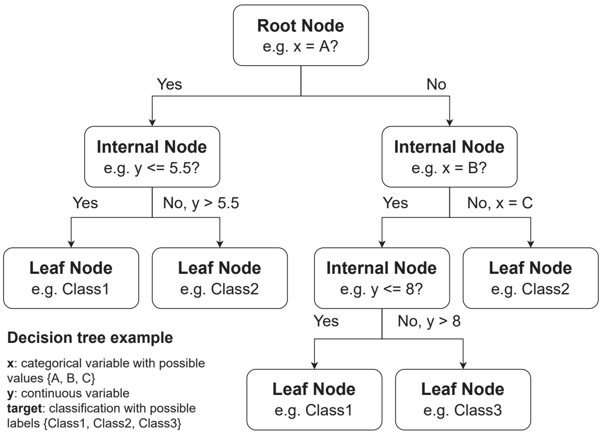

Fig. 1: Decision tree example. A possible structure of a binary decision tree is shown with different node types - root node, internal node, and leaf node. There are examples of conditions for both categorical variable x and continuous numeric variable y. The example has three classes visible in the leaf nodes.

Decision tree learning is done by finding the best set of conditions based on the features' values in the training data to split the datasets into subsets of instances corresponding to one (dominant) target outcome (class/label). There are many algorithms proposed on how trees can be constructed. Several surveys [3], [5], [12] discuss different decision tree algorithms, such as Iterative Dichotomies 3 (ID3) [23], successor of ID3 - C4.5 [24], an extension of C4.5 algorithm - C5.0 algorithm [25], Classification And Regression Tree (CART) [2], CHi-squared Automatic Interaction Detector (CHAID) [8], and Quick, Unbiased and Efficient Statistical Tree (QUEST) [13]. Of the mentioned algorithms, the most popular and successful are C4.5, C5.0, and CART. However, C5.0 is not publicly available. Between C4.5 and CART, based on the performance, there is no superior one, so we use CART as the implementation is easier to integrate. We first describe the general procedure and mention the differences between various proposed learning algorithms.

The decision tree learning algorithm starts with a collection of samples described by a specific number of input features. Each instance has a particular outcome (target): a numeric value for regression or a class/label for classification. Since we consider a classification problem where we predict fault classes based on the laser fault injection parameters, we further explain the decision tree learning process considering a classification problem. We aim to have pure subsets of samples in the decision tree, where pure means that all the instances in the subset belong to one specific target class. The learning algorithm selects the conditions by evaluating the resulting subsets. The idea is to lower the impurity of the subsets with each split until the subsets become pure. The internal nodes represent the conditions over the feature values, and each branch from that node is either one possible value of that feature or a subset of those values. A few examples of different types of conditions are visible in Figure 1. The branches lead to other internal nodes with different conditions over the input features or leaf nodes. Leaf nodes represent the target classes or a probability distribution over the target classes. Passing the internal node conditions and reaching the leaf node, the data sample gets classified as the specific class indicated by the leaf node. The goal for classification trees is to arrive at different classes with the least number of splits and the lowest misclassification error rates. However, there are trade-offs to be considered to avoid overfitting to training data that causes the model to generalize poorly.

The difference in the many proposed learning algorithms is how they measure which split is better and which feature and its corresponding value should be used to separate the dataset in the best way. Another difference is how many branches can exist from a condition: only two (binary) or multiple partitions. For multiple partitions, a good example is a categorical variable. If a categorical variable has only three possible values, the condition could have three branches, each corresponding to one of the variable's values. Splitting the dataset is done when the subset at a node is pure or splitting no longer improves the prediction performance. Additionally, the learning algorithm can stop if, for example, the tree reaches a defined maximum depth or a minimal number of samples for the split.

CART Learning Algorithm: The CART algorithm supports classification and regression problems. Decision trees created by the CART algorithm are binary trees, meaning that the node branches in only two subtrees. For numeric feature A, the internal node criteria are defined as $A \leq h, A > h$, where the threshold h is found by sorting the values of feature A and then choosing the split between successive values that are best depending on the criterion utilized. The midpoint between two consecutive values is selected in CART as the threshold. For categorical values, the conditions are all subsets from the possible set of categorical values. Different algorithms use different measures to select the condition in the internal nodes. CART originally proposed a Gini index for the splitting measure, also known as Gini impurity. The Gini index measures how often a randomly chosen sample from the set would be incorrectly classified. If Gini is 0, it expresses the purity of classification, i.e., all elements belong to a specified class. If Gini is 1, it indicates the random distribution of elements across various classes, and 0.5 shows an equal distribution of samples for all classes. On the other hand, Information Gain is calculated by subtracting the weighted entropy of each child node from the parent node, where entropy is a measure of impurity or randomness in the data points. Since the process continues until each subset contains samples of the same class or the splits no longer offer improvement, the resulting tree can often be very large and complex. Additionally, the generalization ability of such a tree beyond the training data might be poor. Thus, the CART includes pruning of the trees. The idea is to remove subtrees that do not contribute to the classification accuracy of unseen data. The pruning in CART is done from the bottom of the tree, examining each subtree. If replacing the subtree with a leaf or its most frequently used branch leads to a lower prediction error rate, the tree is pruned accordingly. We use the CART algorithm implemented in the Python package *scikit-learn* [19]. The *scikit-learn* implementation of the CART algorithm uses the total sample weighted impurity of the terminal nodes instead of estimating the prediction error rate. The impurity of a node depends on the criterion used. This way, there is no need for a test dataset for estimating the misclassification rate.

B. Memetic Algorithm (MA)

A memetic algorithm consists of a population-based search and a local search for some individuals [18]. The population-based search, in our case, is a genetic algorithm (GA), where the population is a set of solutions for the optimization problem. For each specific optimization problem, there is a particular solution representation. Once the solution representation is defined, the algorithm generates the initial population using an initialization procedure. Usually, the initialization procedure is random sampling. Then, the algorithm uses the GA operators. First, the individuals from the population get selected by a selection operator. The selection determines which solutions produce offspring. The fitness function samples the population to allocate the parent solutions. Fitness is a relative measure of how good solution A is compared to the rest of the current population or a different solution B. Usually, solutions with better fitness are selected as parents, reasoning that these solutions have "better genes". After selecting the parent solutions, the parents' genes are recombined in the crossover operator. Created offspring can then be modified using the mutation operator. The mutation operator introduces new variations in the solutions and thus the population. Mutation can be done by randomly changing some of the genes in the solutions or replacing an offspring with a completely new solution. A commonly used mechanism is elitism, where one or more best solutions from the current generation are placed directly into the next population. This mechanism ensures that the optimal solution will not be lost in the following generations once it is found. Before starting a new iteration, a portion of solutions is selected for the local improvements. A good option for the local search is the Hooke-Jeeves algorithm, an optimization algorithm that does not require derivatives of the objective function [7]. This algorithm in the context of FI is explained in [11].

III. Proposed Method

Our proposed method combines the explained decision trees and the current state-of-the-art memetic algorithm. The MA we use is an existing implementation described in [11]. The representation of the laser fault injection solution is an array of numbers indicating laser fault injection parameters - *x*, *y*, *trigger delay*, *laser pulse width*, and *intensity*. The user sets the bounds for these parameters that can be used in the algorithm. The algorithm starts with the initial population, created by an initialization method. In [11], the authors explored different initialization methods. As no specific one achieved significantly better results than the usual random sampling, we keep it as the initialization method for MA to compare with the newly proposed approach. The difference in the new approach is in the initialization method. We propose to use decision tree rules for initializing the population. We perform a campaign with laser fault injections on a specific target to obtain the data. We can then analyze the acquired data to determine what parameters caused *fail* responses vs. the other responses. We utilize the decision tree algorithm to learn when we obtain each fault class depending on the parameters. That is a classification problem, as our labels are categorical values *pass*, *mute*, *fail*, and *changing*. These are the names of fault classes that correspond to a specific device response, and more details can be found in Section IV-A. The input features are the five LFI parameters. The trained model can be used to predict the behavior of the target on parameter combinations that were not tested. Additionally, we can use it to analyze what parameters are more relevant for achieving *fail* responses if there are such. We, however, use the model to improve performance when conducting another campaign on a different integrated circuit (IC). Specifically, we test the approach on transferability between different samples of the same target.

Since the decision trees can be easily translated to *if-then* rules, we can acquire rules for the *fail* fault class. We select a trained model and traverse the decision tree to extract paths where the leaf nodes are *fail* classes. Since the paths contain conditions on the different parameters, we get intervals for the parameters that lead to *fail* classes based on the model. We denote the combination of the intervals for parameters from one path in the tree as a rule. After we obtain the rules for the *fail* class, we apply them to initialize the memetic algorithm's population. We use only a maximum of 25 rules with the highest number of classified *fail* examples as our initial exploration gave good results with this number. The limit can be further investigated for an even better choice. To create one solution for the population, we first select one rule if there are many. The probability of selecting a rule is proportional to the number of *fail* examples classified by the rule. Since the rule defines the parameters' intervals, we select the parameter value uniformly at random from the defined intervals. Each solution can be created from a different rule, and the whole population is built in the same manner.

One could argue that this might reduce the exploration ability of the memetic algorithm. However, since the MA operators (crossover and mutation) are not modified to use the prior knowledge, the algorithm can still explore outside initial solutions and exploit the "head start".

The rest of the memetic algorithm in the proposed methods remains the same as in [11], except for the crossover operator. Instead of using the average crossover, we use a uniform crossover. With uniform crossover, values for the child solution are taken randomly from the two selected parents, and the probability is equal for both parents' parameter values. Compared to the average crossover that calculates the mean for all parents' parameters and leads to one specific child every time, the uniform crossover can create more different children with the same parents. That leads to more possibilities which can lead to better results sooner.

The evaluation includes performing the laser shots and acquiring the devices' responses. The responses are categorized into fault classes - *pass*, *mute*, and *fail*. Each parameter set is tested several times, so if multiple fault classes appear with the same solution, the class for that parameter combination is *changing*. The MA does not work directly with these fault classes. We use a fitness value, which is given to each fault class. The fitness values for the *pass*, *mute*, and *fail*, taken from [14], are 2, 5, and 10, respectively. The values display the preference of the fault classes, where the *fail* class, being the desired one, has the highest fitness value. As in [11], 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, and f_F represent the fitness values for fault classes *pass*, *mute*, and *fail*, respectively. N_P, N_M, and N_F represent the number of times the *pass*, *mute*, and *fail* class occurred out of the number of measurements for a specific parameter set. The sum of N_P, N_M, and N_F is the number of measurements per parameter set.

The number of solutions taken for local search can vary, and, based on the improvements during it, the number of tested parameters can be different in each iteration. Additionally, MA keeps the tested solutions in a list to avoid repeating the measurements. Thus, if the algorithm creates the same solution already tested, we do not execute the laser injection again but return the previous result. That explains the variations in the number of tested parameters we display in different tables throughout the paper.

IV. Experimental Setup

A. Targets

We use products from STMicroelectronics. Due to confidentiality reasons, we cannot disclose the details of the targets and the utilized laser bench. Information about the laser bench might inform possible malicious attackers on what kind of bench they could use to attack the products. Utilized integrated circuits (ICs) are constructed with 40nm technology, and all went through a mechanical thinning process before the experiments as part of the preparation for conducting laser injections. Along with differences introduced during the hardware production, thinning of the backside silicon substrate can also affect differences in device sensitivity to laser injections. However, as mentioned, the assumption is that these differences are negligible. The code running on the target device is a test program where data words are loaded from the non-volatile memory (NVM) into a register. Its implementation is in the C programming language displayed in Pseudocode 1. 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 with 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*). If there is no response from the device, we categorize this as a fault class *mute*, and if the laser injection does not modify the data, the fault class is *pass*.

Pseudocode 1: Pseudocode of the program running on the target devices.

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

We optimize the following five parameters - *x*, *y*, *delay*, *laser pulse width*, and *intensity*. These parameters are often used in literature and practice during a security evaluation. We use a subset of the available values for each of the five parameters, defined according to the known layout. While we cannot share the parameter intervals as they are specific to the product and laser bench, we note that there are $305\,017\,650$ possible combinations of the parameter values. Thus, search optimization is highly relevant as the exhaustive search with the utilized subset of possible values would take around 529 days if we consider that one laser shot takes ≈ 0.15 seconds. Additionally, since we perform the laser shot several times with the same parameters, this would increase the necessary time to perform the exhaustive search.

B. Experimental Process

We use three samples of the same target in our experiments, referring to them as IC1, IC2, and IC3. IC1 is an integrated circuit for obtaining the decision trees' training and test data. One training dataset is obtained from the memetic algorithm with random sampling for initialization and another from a random search. The random search is defined to test $50\,000$ different examples. We train different decision tree models on the memetic algorithm (MA) and random search (RS) data. To test the models, we use prediction performance, and to calculate it, we need test data. Test data is again obtained from the IC1 using a random search with $5\,920$ examples and a search we refer to as the fast grid search (FGS). The FGS uses the same bounds for *x* and *y* as other algorithms, but the laser settings are fixed. The values of the laser settings are defined based on the most often values for *fail* class from the initial experiments on IC1 using a MA with random initialization. We also defined the number of parameter sets for the random search based on the same experiments where we tested on average $5\,917.4$ different parameter combinations. Later, we apply the memetic algorithm with the DT models, trained on data from IC1, for campaigns on IC2 and IC3. We investigate if we can use the knowledge from IC1 on IC2 and IC3 to improve the memetic algorithms' performance and test how well the obtained knowledge transfers between different samples. Additionally, we make changes to the bench setup while changing the ICs. The laser focus is less sharp for experiments on IC2 but was again improved for experiments on IC3. Additionally, for the experiments on IC3, we change the bench setup to lose less power from the laser source to the target. These changes have the most effect on the laser intensity parameter.

C. Memetic Algorithm Hyperparameters

For the memetic algorithm, we use a maximum number of iterations of 100 as the stopping criterion since our experiments indicate that this is sufficient to reach convergence. We perform five measurements with the same parameter combination, which means we conduct five laser shots per parameter combination. This way, we can obtain different responses categorized as a *changing* fault class. We use a population of size 100, with an *elite_size* of 10. Thus, the ten best solutions from the population are transferred to another generation without changes. Lastly, the mutation probability is 0.05. These hyperparameters stay the same in our experiments and are decided based on the preliminary investigations and information from previous work [11].

D. Decision Tree Hyperparameters

The *scikit-learn* implementation of CART has many DT hyperparameters, but we only used some to obtain models for comparison as many are not independent. The hyperparameters we chose to test and the specific values used are in Table I. First, the *criterion* hyperparameter is the function to measure the quality of the split. The first option is 'gini' for Gini impurity, and the other is 'entropy' for Information Gain. Next is the *splitter*, a strategy to choose the split at each node. Options are either the 'best' or the 'random' split. Then, *min_samples_split* that indicates the minimum number of samples required to split an internal node that by default is 2. *ccp_alpha* is a hyperparameter used for minimal cost-complexity pruning, but by default, no pruning is performed, and the value is 0. The intervals for the *min_samples_split* and *ccp_alpha* are defined based on the empirical study [16], [15], where the authors used a package *rpart* written in R

TABLE I: Decision tree hyperparameters with values used in our experiments. All combinations of listed values are tested. For the numerical hyperparameters, the interval is inclusive, and the step size is mentioned next to the interval.

Hyperparameter	**Values**
criterion	{'gini', 'entropy'}
splitter	{'best', 'random'}
min_samples_split	[2, 42], step = 4
ccp_alpha	[0, 0.020], step = 0.005
class_weight	{None, 'balanced'}
balance data	{True, False}

programming language[1] [28] for implementation of the CART algorithm. From their results, most values selected by the optimization techniques for the pruning hyperparameter were in the range 0 to 0.02. For the minimum number of instances necessary for a split to be attempted, most of the values were below 40. We note that the authors also considered the minimum number of samples in a leaf and the maximum depth of any node in the tree called *min_samples_leaf* and *max_depth* in *scikit-learn* implementation, respectively. However, we do not experiment with these two hyperparameters. The *min_samples_leaf* was mostly below 10, and the number of models with a certain *max_depth* value was increasing with a larger *max_depth* value. Thus, we keep the default value of 1 for the minimum number of samples in a leaf and enable the tree to grow until all leaves are pure or contain less than *min_samples_split* (default for *max_depth*). These two hyperparameters can regulate overfitting, but the *min_samples_split* and *ccp_alpha* for pruning do the same. Additionally, the *scikit-learn* has a slightly different pruning method, so we kept the *ccp_alpha* hyperparameter. Since there is an imbalance in the training set because there is usually a majority of *pass* classes compared to all others, we included the *class_weight* hyperparameter. The 'balanced' mode adjusts weights inversely proportional to class frequencies in the input data, and by default, with None, all classes have the weight one. We also included an option to balance the training dataset, which is not part of the *scikit-learn* implementation, but we added it to Table I. The method for balancing the dataset finds the class with the least number of samples and then randomly samples the exact number of samples for all other classes (random undersampling). Other hyperparameters from the *scikit-learn* implementation that are not mentioned in the table are kept at default values. We train 880 decision tree models based on the described hyperparameter combinations. Each model is trained ten times because of the randomness in the algorithm.

V. Experimental Results

A. Comparison of Different ICs with a Fast Grid and Random Search

We execute a fast grid search where the bounds for *x* and *y* are the same as in other experiments, but we use a large step to test the whole area with a grid search rather fast. Laser settings are fixed based on the memetic algorithm results conducted on IC1. From the memetic algorithm, we analyzed the results and found the most often values for the *delay*, *pulse width*, and *intensity* with *fail* class and used them for FGS. We conducted the same search with identical parameter combinations on all three ICs, and the results are provided in Table II. Similarly, we conduct a random search on all three ICs with the same bounds for all the parameters. We perform only one random search on each IC, and the results are shown in Table III. From both experiments with fast grid and random search, we get most *fails* on IC3. Recall that we have changed the bench setup slightly between experiments with each target sample, as described before. Considering the changes on the bench, the differences between the fault class distributions seem reasonable. With a worse focus on IC2 compared to IC1, we have fewer discovered *fails*, while with the improved setup on IC3, we have ≈ 3.7 times more *fails* with fast grid search and ≈ 2.7 times more *fails* with the random search.

TABLE II: Fast grid search with $12\,350$ tested parameters on different ICs.

Fast Grid Search (1 run)	**IC1**	**IC2**	**IC3**
Tested combinations	12 350	12 350	12 350
fail	9 (0.07%)	0 (0%)	33 (0.27%)
changing	87 (0.7%)	14 (0.11%)	154 (1.25%)
mute	43 (0.35%)	0 (0%)	126 (1.02%)
pass	12 211 (98.87%)	12 336 (99.89%)	12 037 (97.47%)

TABLE III: Random search with $5\,920$ tested parameters on different ICs.

Random Search (1 run)	**IC1**	**IC2**	**IC3**
Tested combinations	5 920	5 920	5 920
fail	7 (0.12%)	3 (0.05%)	19 (0.32%)
changing	74 (1.25%)	27 (0.46%)	66 (1.11%)
mute	43 (0.73%)	12 (0.2%)	99 (1.67%)
pass	5 796 (97.91%)	5 878 (99.29%)	5 736 (96.89%)

B. Training and Testing Datasets

In our approach with decision trees, we first need a training dataset. The training dataset, in our case, is the data from IC1. We create two datasets. First, we have a dataset acquired with a random search. The difference from the random search shown in Table III is that we execute the random search to test $50\,000$ unique five-parameter combinations instead of $5\,920$. The other training dataset comes from the experiments using the memetic algorithm with random initialization. The memetic algorithm is executed ten times to obtain ten independent results (runs). In total, with the memetic algorithm, we obtain $61\,107$ unique five-parameter combinations. The described training datasets can be seen in Table IV, showing the fault class distributions for both datasets. In both datasets, the number of instances for the *fail* fault class is low, but in total, with MA, we have $\approx 3\,600$ *fail* examples, and with random search, only 58. Thus, with the random search, the *pass* class is dominant as 98.53% of examples belong to the *pass* class while, with MA, 82.08% belong to the *pass* class. This might cause the classifier to ignore the classes with few instances, as predicting only the dominant class can still give high prediction accuracy. However, we include the MA data as another training dataset as there are more examples of *fail* class. Note that our results confirm observations from [11] that the memetic algorithm works better than random search.

We must define a way to evaluate the models trained on the datasets. The idea is to use the prediction performance of the decision tree models to evaluate how well these models would perform and improve the memetic algorithm when used in the initialization. We need test data different from

[1] https://cran.r-project.org/web/packages/rpart/index.html

TABLE IV: Training data on IC1: memetic algorithm (MA) with random initialization and random search (RS). The numbers present an average value over ten runs for MA and one run for RS.

Training data	**MA with Random Initialization** (10 runs)	**Random Search** (1 run)
Tested combinations	6 150.5	50 000
fail	366.5 (6.12%)	58 (0.12%)
changing	548.6 (8.92%)	451 (0.9%)
mute	182.6 (2.89%)	226 (0.45%)
pass	5 052.8 (82.08%)	49 265 (98.53%)

the training data (unseen examples) for predictions. In our case, we are interested in how well the model trained on IC1 predicts the fault classes on other ICs. However, since we want to use the proposed approach on other ICs without acquiring their campaign data, we also use IC1 for the test dataset. The test datasets are results from the fast grid search (FGS) and random search (RS) conducted on IC1, presented in Tables II and III, respectively. Therefore, this random search is another campaign with the same algorithm (random sampling) as the training data (Table IV) but with fewer tested combinations. We can see that the distribution of fault classes in the experiments with the random search for training and test are similar. Thus, we expect the models trained on random search data to perform well on the random search test data. On the other hand, since the data from FGS is very different from both training datasets, it might be more challenging for these models to predict correctly on the FGS test dataset.

C. Training the Models and their Prediction Performance

After we prepare the training and test datasets, we train 880 different decision tree models by applying different hyperparameters of the decision tree. In Table I, we report values for all the hyperparameters we tested. Each combination of the hyperparameters is tested ten times, and then we calculate the average of the metrics from the predictions on test datasets to assess the performance. Additionally, we train the DT models separately on random search (RS) data and memetic algorithm (MA) data. We do not consider fast grid search for training as it only has data for one combination of laser settings, and in total, there are only 12 350 data points for learning.

As mentioned, the *pass* class is the dominant class based on the number of examples per class. For this reason, we investigate different classification metrics prevalent in machine learning. Usually, *accuracy* is used, but with imbalanced data such as ours, one could use *recall*, *precision*, or *f1_score*. In multi-class classification, the metrics *precision*, *recall*, and *f1_score* are calculated for each class. However, in our work, we focus on the *fail* responses, so we only consider the *precision*, *recall*, and *f1_score* for the *fail* class, ignoring the rest. *Accuracy* is the fraction of the total samples that were correctly classified. Since we want the model to learn when *fails* happens, *accuracy* might not be a good choice because of a low number of samples for all classes except *pass*. *Precision* is the number of samples correctly predicted as a class for which the metric is calculated from all samples predicted as that class, including those belonging to some other class. With high *precision*, the rules from the DT model might be specific, e.g., defining one combination of parameters that leads to a *fail* class. However, considering our application of the rules, where we switch between different ICs, we can allow some samples to be classified as *fail* even if they are not *fails*. For example, if the same parameter set with a *fail* fault class on one IC does not lead to a *fail* class on another IC, a parameter set with minor differences may lead to a *fail*. The *recall* is the fraction of class samples for which the metric is calculated that were correctly predicted as that class. Thus, *recall* tells how many true *fails* in the test data were predicted as *fails*. We would like to get this metric quite high for our case, but the model may predict everything as a *fail* class to accomplish this, which again would not be useful. Thus, there is *f1_score*, calculated as a harmonic mean of *precision* and *recall*, to find a balance between the two metrics.

To get an idea about the models' predictions, we select the best model for each metric and compare the original distribution of the test data and the predicted distributions of the classes. In Table V, we show the original distribution of the random search data set on IC1 in the first column of the table. Similarly, we show the class distribution based on the DT models' predictions. As mentioned, we take the best models based on each metric. Thus, the rest of the columns show distributions from predictions of the best model based on the *accuracy*, *precision*, *recall*, and *f1_score*, respectively. The metrics are highlighted in the header to specify the metric used to select the best model.

TABLE V: Comparing prediction metrics. The first column is the original test data from IC1 (random search). The following columns correspond to predictions from models trained on MA data on the given test data from IC1. The predictions come from the best models based on the highlighted metric. The numbers represent the distributions of fault classes.

	Random Search test data IC1 True distribution	**Accuracy: 0.9796** Precision: 0 Recall: 0 f1_score: 0	Accuracy: 0.9774 **Precision: 0.3333** Recall: 0.1429 f1_score: 0.1999	Accuracy: 0.1873 Precision: 0.0018 **Recall: 0.8** f1_score: 0.0036	Accuracy: 0.9535 Precision: 0.1765 Recall: 0.4286 **f1_score: 0.25**
fail	0.12%	0.05%	0.05%	100%	0.29%
changing	1.25%	0.56%	0.19%	0%	2.4%
mute	0.73%	0.25%	0%	0%	1.72%
pass	97.91%	99.14%	99.76%	0%	95.59%

We can see that with *accuracy*, the model preferred to predict a *pass* fault class, so the percentage of predicted *fails* is lower than in the original dataset. With *precision*, even fewer data points are predicted as any other class except for *pass*. Then, with the *recall* of 0.8, the model predicts all classes as *fail*, which is not the desired behavior. Lastly, the best model based on *f1_score* generalizes the best. While other models in this table predict too little or too many *fail* classes, this model finds a balance. We notice this model predicts more *fails* than there are, but this can be the desired effect for our analysis. Indeed, if the area for the *fail* class is small and specific, the *fails* on another IC might not be at those exact points. However, they can be very close. Thus, if the model correctly allocates intervals for *fails* class and makes them slightly larger, it gives us more chances to find *fail* responses on another IC with those intervals. Therefore, we choose to use the *f1_score* metric for selecting the model to use its rules for initializing the population of the memetic algorithm.

There are two test datasets - fast grid search (FGS) and random search (RS), so we test on both and average the metric values. We show these average values in Table VI for the best models based on *f1_score* trained on memetic algorithm (MA) data and random search (RS) data, followed by other models used in our experiments. This table also shows the decision tree hyperparameters that define the model. Hyperparameters values in table correspond to splitting *criterion*, *splitter*, *min_samples_split*, *class_weight*, *ccp_alpha*, and flag for balanced data, in that order. The two best models differ in only the *min_samples_split*. The *min_samples_split* is a parameter that helps prevent overfitting in decision trees. If we allow this parameter to be very low (minimum is 2), the model can overfit to training data. However, if we increase the *min_samples_split*, the chances of overfitting are lower. Since in the MA training set, we have more examples of the *fail* class than with RS data, it is reasonable that the model trained on RS data requires a smaller *min_samples_split*. Smaller *min_samples_split* forces the model to learn when *fail* classes occur by finding those specific cases. On the other hand, if we include pruning or increase the *min_samples_split*, the model can predict only the *pass* class and, in most cases, it will be correct since the majority of examples in training and test data are indeed *pass* classes.

TABLE VI: All models used in the experiments with their prediction metrics and hyperparameters. Metric values are average from results on random search (RS) and fast grid search (FGS) on IC1. The best models are selected based on the highlighted *f1_score* metric. Model parameters are *criterion*, *splitter*, *min_samples_split*, *class_weight*, *ccp_alpha*, and a flag for balance data, in that order.

Average tested on RS and FGS data	*accuracy*	*precision*	*recall*	***f1_score***	**Model parameters**
Best model from MA data	0.9529	0.1227	0.3254	**0.1776**	gini, best, **30**, balanced, 0.0, False
Best model from RS data	0.9630	0.1476	0.3254	**0.1999**	gini, best, **10**, balanced, 0.0, False
Model from MA data with lower *f1_score* 1	0.9831	0.1666	0.0714	**0.0999**	entropy, best, 10, None, 0.01, False
Model from MA data with lower *f1_score* 2	0.9833	0.1666	0.0556	**0.0833**	gini, best, 30, None, 0.0 False
Second-best model trained on RS data	0.9534	0.1238	0.3254	**0.1756**	gini, best, 14, balanced, 0.0, False

D. Experiments on Different ICs

Once we obtained training and test data on IC1 and trained the decision tree models, we can change to other samples of the same target: IC2 and IC3. First, we ran experiments on IC2, starting with the already presented random search and then the memetic algorithm with random initialization. The target's behavior and the setup used for the injection can influence the results. When using the same target and setup and only changing different samples, the assumption is that the results should be similar, with minor variations. The variations can come, for example, from unplanned production differences in hardware, unintended chip alignment on the setup, and silicon thickness variations. However, these should be negligible. In our case, we use different samples of the same target, but we also change the focus of the laser spot on the chip. The focus while running experiments on IC2 was worse than for experiments on IC1. The focus directly influences the laser parameter *intensity*. The results from the random search were already given in Table III but are here presented in Table VII with the memetic algorithm with random initialization to allow easier comparison. We compare the results on IC2 with those on IC1, and, as seen in Table III, with RS, we found less than half of what we found on IC1 for all classes except *pass*. Accordingly, the number of found *pass* classes increased. However, with the MA on IC1 (Table IV), only for the *changing* class, we had around two times more examples than with MA on IC2 (Table VII). For other classes, while still slightly fewer examples were found on IC2, the results are comparable. Thus, the guided search in MA contributes to the algorithm being more transferable as it adapts to the behavior of the target and changes in the setup.

TABLE VII: Experiments on IC2: memetic algorithm (MA) with random initialization and random search.

	MA with Random Initialization (10 runs)	**Random Search** (1 run)
Total combinations	6 389.6	5 920
fail	304.7 (5.03%)	3 (0.05%)
changing	250 (3.88%)	27 (0.46%)
mute	147.9 (2.28%)	12 (0.2%)
pass	5 687 (88.82%)	5 878 (99.29%)

Finally, we run the newly proposed approach as described in Section III. With the new method, we should have more parameter sets with a *fail* response in the initial population and therefore provide the memetic algorithm with a "head start" which can be exploited to obtain more *fails* in a similar number of tested parameters.

We first run the experiment with the best model trained on MA data, and then we test a model with a lower *f1_score*. Indeed, the question is whether *f1_score* is a good metric to consider when selecting the model for our application - initialization of the memetic algorithm's population on a different sample of the same target. Thus, we ran a model with a lower *f1_score* than the best one, but not the worst because that is a model with an *f1_score* of 0. If the *f1_score* is 0, the model does not perform well even for the predictions on IC1, which is used for training, so we expect it to not perform well in our case. Therefore, we selected the first model trained on MA data with an *f1_score* lower than 0.1. The selected model has an *f1_score* of 0.0999, and we use it to test if the lower *f1_score* leads to fewer *fails* found by the memetic algorithm using the model in the initialization. The numbers from these experiments can be seen in Table VIII in the second and third columns. From the headers, we distinguish models based on their *f1_score*. We see that both models trained on MA data improved the performance of the MA algorithm compared to MA with random initialization. MA with random initialization had 5.03% of *fails*, while MA with models has 16.86% and 12.48% for models with *f1_score* 0.1776 and 0.0999, respectively. Notice the performance of the MA is worse with the model with a lower *f1_score*. However, it is still better than a MA with random initialization.

Then, we also test with the best model trained on the RS data. The results are visible in Table VIII. This model has a higher *f1_score* (0.1999) than the best model trained on MA data (0.1776), but it found the least *fails* from all the other models, even the model with an *f1_score* of 0.0999. We use data from a random search for training and calculating the metrics, which might be why a model with a higher *f1_score* trained on RS data does not find more *fails* when used in MA. It could be that the model created a precise interval for when the *fails* in RS data occurred, or it generated random intervals without learning what parameter values lead to *fail* responses. Thus, we ran the algorithm with a second-best model trained on RS data that has *f1_score* of 0.1756, which is close to the *f1_score* of the best model trained on MA data. The algorithm has found the most *fail* classes between the four tested decision tree models. We can see that the number of tested parameter sets (combinations) also increases, which could cause the difference in the found *fails* compared to the results with the model trained on MA data with *f1_score* of 0.1776. Thus, we conclude that the performance of the models trained on MA and RS data with a comparable *f1_score* is similar in the distribution of fault classes when used in the MA initialization.

TABLE VIII: Experiments on IC2: fault class distribution for the memetic algorithm (MA) with the random initialization, followed by MA with a decision tree (DT) rules used in initialization. The models are distinguished by their *f1_score*, and the hyperparameters for each are in Table VI. The header also states which data the model is trained on - memetic algorithm (MA) or random search (RS) data.

Mean (10 runs)	**MA with Random Initialization**	**MA with DT rules** f1: 0.1776 MA data	**MA with DT rules** f1: 0.0999 MA data	**MA with DT rules** f1: 0.1999 RS data	**MA with DT rules** f1: 0.1756 RS data
Tested combinations	6 389.6	5 059.3	5 284.2	5 581.3	5 507.9
fail	304.7 (5.03%)	848.4 (16.86%)	676.5 (12.48%)	633 (11.38%)	1 072.7 (19.39%)
changing	250.0 (3.88%)	234.6 (4.61%)	216.9 (4.11%)	160.2 (2.89%)	198.7 (3.6%)
mute	147.9 (2.28%)	37.3 (0.74%)	81.7 (1.53%)	16 (0.29%)	33 (0.58%)
pass	5 687.0 (88.82%)	3 939 (77.79%)	4 309.1 (81.51%)	4 772.1 (85.44%)	4 203.5 (76.43%)

We also compare the performance of the models based on the number of *fail* responses in the initial population and when the first *fail* is found. This can help understand how the model improves the initial population and impacts the algorithm's overall performance based on this information. The best model trained on the MA with *f1_score* 0.1776 finds the most parameter sets with *fail* response in the initial population - on average 18.2/100 compared to 6.3/100, 2.7/100, or 4.4/100, for the other three models. Both models trained on MA data had a higher number of *fail* classes in the first population than models trained on RS data. That might be because we have more parameter set examples for the *fail* response with MA data, which helps the model learn. The model trained on RS data with *f1_score* of 0.1756 on average had only 4.4/100 *fail* responses in the initial population. Still, in combination with MA, it found the most *fail* responses. Thus, we see that the rest of the algorithm can influence the overall performance of the memetic algorithm as it affects the exploration. Initially, we might focus on only one area found by the model, but crossover and mutation can find *fails* in other regions on another IC. Additionally, we noticed more tested parameters when the model in the initial phase found fewer *fail* responses. MA with DT rules finds the first *fail* response earlier in the algorithm compared to MA with the random initialization, and the initial population has more *fail* examples in the initial population. Therefore, the models improve the initial population. However, considering the overall performance, the algorithm with the most *fails* in the initial population did not find the most *fails*. Still, MA with all tested models improved the performance of the MA with random initialization by finding at least two times more *fail* classes.

TABLE IX: The first *fails* with all the DT models tested on IC2.

Population size = 100	**MA with Random Initialization**	**MA with DT rules** f1: 0.1776 MA data	**MA with DT rules** f1: 0.0999 MA data	**MA with DT rules** f1: 0.1999 RS data	**MA with DT rules** f1: 0.1756 RS data
First *fail* (0-indexed)	53, 3 335, 2 578, 377, 1 381, 1 381, 1 847, 3 071, 1 223, 5 681 **(avg 2 092.7)**	41, 36, 38, 41, 31, 34, 50, 36, 57, 36 **(avg 40.0)**	43, 46, 25, 39, 66, 58, 8, 23, 46, 82 **(avg 43.6)**	33, 203, 47, 84, 52, 59, 55, 53, 56, 46 **(avg 68.8)**	56, 38, 46, 28, 32, 60, 71, 53, 73, 59 **(avg 51.6)**
Number of *fails* in the first population	1, 0, 0, 0, 0, 0, 0, 0, 0, 0 **(avg 0.1)**	14, 34, 23, 18, 15, 17, 16, 15, 14, 16 **(avg 18.2)**	5, 6, 7, 8, 7, 3, 8, 8, 10, 1 **(avg 6.3)**	2, 0, 3, 4, 3, 1, 6, 3, 4, 1 **(avg 2.7)**	6, 4, 5, 4, 6, 6, 3, 5, 3, 2 **(avg 4.4)**

We further test the models on another sample of the same target, IC3, where the bench was slightly modified as described in Section IV-B. We already discussed the results of the fast grid and random search compared to other ICs in Section V-A. With IC3, we found more *fails*, even when the same points were tested (fast grid search). From the experiments on IC2, the memetic algorithm benefits from using DT models in the initialization. Thus, we do not test the memetic with random initialization on IC3. We test the best model trained on MA data with *f1_score* of 0.1776 and the second-best model trained on RS data with *f1_score* of 0.1756 as it was better than the model with 0.1999 *f1_score*. We also test another model trained on MA data with a lower *f1_score* of 0.0833. The results from the random search and the mentioned three models are in Table X. The algorithm finds at least two times more *fail* responses on IC3 than IC2 with the models, which is in line with experiments with random search and fast grid search because of the changes on the bench. Again the second-best model trained on RS data in combination with MA found more *fails* compared to the best model trained on MA data with *f1_score* of 0.1776. However, in the case of IC3, it is interesting that the model with a worse *f1_score* of 0.0833 that was trained on MA data performed better than the other two models. The difference in the number of tested parameter combinations in these experiments does not explain that improvement. Thus, we again compare the number of *fails* found in the initial population and discuss possible reasons. Compared to results on IC2, we see that both with IC2 and IC3, the model with the *f1_score* of 0.1776 found the most *fails* in the initial population, on average 33.5 examples. In both cases, the second model based on the number of *fails* found in the initial population is the model with a worse *f1_score* trained on MA data. For IC2, this was the model with *f1_score* of 0.0999 and 0.0833 for IC3. Then, it is the second-best model trained on RS data with *f1_score* of 0.1756 with 12.9 *fails* in the initial population. Considering the algorithm's overall performance, again, the model with the most *fail* examples in the initial population did not, in the end, find the most *fails*. Nonetheless, experiments on IC2 and IC3 show that the MA with rules improves the performance of random search by obtaining two orders of magnitude more *fails* and up to 60% more *fails* than the MA with random initialization.

TABLE X: Experiments on IC3: fault class distribution for a random search, followed by the memetic algorithm (MA) with decision tree (DT) rules used in initialization. The models are distinguished by their *f1_score*, and the hyperparameters for each are in Table VI. The header also states which data the model is trained on - memetic algorithm (MA) or random search (RS) data.

Mean (10 runs)	**Random Search** (1 run)	**MA with DT rules** f1: 0.1776 MA data	**MA with DT rules** f1: 0.0833 MA data	**MA with DT rules** f1: 0.1756 RS data
Tested combinations	5 920	5 262.7	5 495.1	5 554.5
fail	19 (0.32%)	1 662.8 (31.53%)	2 688.8 (48.85%)	2 325.7 (41.83%)
changing	66 (1.11%)	221.9 (4.23%)	210.1 (3.81%)	153.4 (2.76%)
mute	99 (1.67%)	126 (2.37%)	74.1 (1.35%)	101.5 (1.82%)
pass	5 736 (96.89%)	3 252 (61.86%)	2 522.1 (45.98%)	2 973.9 (53.58%)

We again notice that the number of tested parameters increases as the number of *fail* examples in the initial population decreases. Thus, we confirm that there is a need to balance the number of *fail* examples in the initial population and that the rest of the memetic algorithm improves the overall performance. This also raises the question of whether the prediction metric alone is the best metric for selecting a model for this specific use case. While all the models improve overall performance and the number of *fails* in the initial population compared to random search and MA with the random initialization, we might need to consider some other aspects of the models to improve the selection of the model. Since we do not use the models strictly for prediction purposes, possibly, instead of using a prediction metric *f1_score*, we need to analyze the models' size or some information about the rules. Combining different aspects of the model and its rules in one metric could give more reliable expectations of the models' overall performance for our use case. We leave this as future work.

TABLE XI: The first *fails* with all the DT models tested on IC3.

Population size = 100	**MA with DT rules** f1: 0.1776 MA data	**MA with DT rules** f1: 0.0833 MA data	**MA with DT rules** f1: 0.1756 RS data
First *fail* (0-indexed)	0, 1, 1, 0, 0, 1, 0, 1, 1, 1 **(avg 0.6)**	4, 5, 2, 20, 7, 2, 6, 9, 2, 2 **(avg 5.9)**	37, 8, 13, 10, 40, 7, 44, 2, 6, 7 **(avg 17.4)**
Number of *fails* in the first population	40, 36, 30, 37, 50, 32, 35, 32, 30, 13 **(avg 33.5)**	33, 24, 33, 25, 28, 31, 24, 24, 15, 22 **(avg 25.9)**	9, 17, 14, 7, 21, 10, 8, 18, 12, 13 **(avg 12.9)**

We use machine learning (decision trees) to provide a good initial population that will, in turn, help the memetic algorithm to find many *fail* responses. It stands to ask if it is possible to compare our approach with the deep learning (multilayer perceptron) approach followed by Wu et al. [30]. We consider those two approaches orthogonal as Wu et al. found a representative set of responses to predict future ones for the same sample of the target device. That way, the authors obtained the complete characterization from a small number of fault injections. This is only an estimate but still gives a great insight into the device's behavior. In our case, we use the model for the prediction on other samples of the same targets for transferability issues between targets. Additionally, we use machine learning to provide an initial population to guide the optimization process. Consequently, we use machine learning in different phases of the target characterization. Still, we believe it could be possible to use our approach to provide an initial population, which will then be used as the training set for a deep learning classifier. Doing this could make the target characterization even more powerful and efficient. We leave this as possible future work.

E. Obtained Rules for Initialization

In Table XII, we show the number of rules produced for the *fail* class by each of the utilized models. Additionally, we display the numbers for possible parameter combinations by a specific rule from the model's set of rules - minimum, median, mean, and maximum, and the total number of possible combinations for all the rules created by the model. It is essential to understand that there could be possible overlaps between the rules, which would lower the total number of unique combinations than those reported in the table. Lastly, we show the number of rules in which not all parameters were used to specify areas that the model predicts as the *fail* class. As mentioned, we use a maximum of 25 rules with the highest number of examples classified as a *fail* class by each rule for initialization. Thus, in the table, for the two models with more rules for *fail* than 25, we also show information specifically for the used 25 rules (separated by $|$ symbol).

TABLE XII: Information about the rules, specifically for *fail* class, found by the models used in our experiments.

Total possible combinations = 305 017 650	**MA with DT rules** f1: 0.1776 MA data	**MA with DT rules** f1: 0.0999 MA data	**MA with DT rules** f1: 0.0833 MA data	**MA with DT rules** f1: 0.1999 RS data	**MA with DT rules** f1: 0.1756 RS data
Number of rules for *fail*	359 | 25	1	205 | 25	24	22
Combinations per rule: Minimum	12 | 60	93 960	4 | 42	2 400	2 400
Median	360 | 135	93 960	96 | 110	25 875	43 310
Mean	2 093.82 | 491.24	93 960	690.6 | 169.28	26 772.25	41 237.27
Maximum	38 808 | 7 436	93 960	33 320 | 624	65 250	87 500
Combinations from all rules	751 682 | 12 281	93 960	141 574 | 4 232	642 534	907 220
Rules **not** defining all parameters	105/359 (29%) | 11/25 (44%)	1/1 (100%)	71/205 (34%) | 11/25 (44%)	18/24 (75%)	13/22 (59%)

While the table only shows information about numbers considering the rules for the *fail* class, the number of rules for classifying all classes for each of the models is $1\,646$, 14, $1\,614$, 901, and 782, following the order in Table XII. We see that the model with pruning (*f1_score* of 0.0999) is the smallest, with only 14 rules, compared to other models without pruning. Additionally, the models trained on MA data have more rules concerning all classes than those trained on RS data, except for the mentioned model with pruning. The same is visible for the number of rules, specifically for the *fail* class. Thus, the models trained on MA without pruning are more complex and larger than those trained on RS data. Additionally, the percentage of rules for the *fail* class is larger for models trained on MA data than RS data, which might be because there are more examples for the *fail* class in the MA training set.

The rule from the model with *f1_score* of 0.0999 (with pruning) considers a path in the decision tree where the conditions for the parameter x were

$$\{x > x1, x > x2, x \leq x3,\ and\ x \leq x4\},\\ where\ \{x1 < x2,\ and\ x3 > x4\}.$$

We then obtain the bounds $x \in \langle x2, x4]$ from this path. For y parameter, the path only had requirements $\{y > y1, y \leq y2\}$. Thus, the bounds are set to $y \in \langle y1, y2]$. For $pulse\ width$, in the path, there was only a condition stating $pw > pw1$, so the rule bounds include the user-defined upper bound for the $pulse\ width$ and the mentioned $pw1$ as the lower bound. The $delay$ and $intensity$ are not in the path, and the user-defined bounds are used in the initialization. With this rule, there are $93\,960$ unique possible combinations for the initial population compared to user-defined bounds that provide $305\,017\,650$ combinations. If the exhaustive search is done for only the rule bounds, it will last around 4 hours, compared to 529 days for the exhaustive search with user-defined bounds considering the laser shot lasts for ≈ 0.15 seconds.

While other models have more rules, each specific rule covers less search space. For example, a model trained on MA data has rules with as little as 4 possible combinations to a maximum of $33\,320$ unique combinations. Considering only the utilized rules, the number of combinations per rule goes from 42 to 624. We also show the number of possible combinations from all the rules, but this is not a unique number of possibilities because we do not consider the overlaps between them. Still, the rules from models trained on RS data have at least 50 times more combinations than the utilized 25 rules from models trained on MA data. The number of rules used in the initialization is similar for all models except for the one with pruning that had only one rule. Combining this information with the number of *fails* in the initial population, models trained on MA data seem to have better rules for transferability. Models trained on MA data on both IC2 and IC3 found more *fails* in the initial population. There are more examples for the *fail* class in MA training data than RS data that could lead to the model creating better rules. Additionally, the number of possibilities per rule from models with MA data is lower than with RS data. That makes the rule more specific, defining smaller intervals where the *fail* class can be expected. However, not too small as having only four combinations because we could miss a *fail* class with a slightly different intensity or location since we change the IC. On the other hand, the lower number of *fails* in the initial population from models trained on RS data can come from large intervals in the rules. For example, the minimum number of combinations in one of the rules is $2\,400$. We could miss the *fails* and select parameter sets leading to less interesting classes with that many possible combinations. We also present the number of rules where not all parameters included a reduced interval, but the user-defined intervals were used. Usually, the laser parameters were not reduced, while the location coordinates were specified in all the rules. Except for the model with pruning (*f1_score* of 0.0999), the models trained on MA data had a higher percentage of those rules specifying bounds for all parameters than models trained on RS data. That again suggests that the rules from models trained on MA data define smaller areas.

VI. Conclusions and Future Work

Previous work [11] showed that the memetic algorithm finds more interesting LFI parameter combinations than a random search that leads to possibly exploitable device responses. This work further improves the memetic algorithm approach by using decision tree models for cases where different samples of the same target are tested, or some minor changes are introduced on the bench. Such decision tree models store the knowledge in a tree structure from which *if-then* rules can be extracted. Thus, we extract rules for the device's interesting *fail* responses. The rules consist of intervals for the LFI parameters, and they can indicate areas where there were most *fail* responses. The approach uses the knowledge obtained from a campaign on one IC in the initialization phase of the memetic algorithm conducted on another IC. We only consider different samples of the same target and minor changes on the bench setup. Considering the number of found *fail* responses, we can see that the performance has significantly improved in the conducted experiments. More precisely, we obtain two orders of magnitude more *fail* responses than random search and up to 60% more *fail* responses than the previous state-of-the-art (memetic algorithm with random initialization).

We show this is possible with the simple version of decision tree learning. In future work, we could consider decision tree ensembles, like the random forest, to further improve the models. Still, we note that the possible improved performance of the models would come with higher computational complexity and more difficult interpretability of the rules due to having many decision trees in the random forest. This work is limited to experimenting with different samples of the same target. The trained models correspond to a specific target and utilized bench and cannot directly be used for other transferability issues, such as changing the bench entirely or using a different target. However, we believe the idea behind the approach can be extended to propose a similar algorithm or different training for those cases. We noticed that all of the models we tested improved the performance of the memetic algorithm. While in this work, we use *f1_score*, a popular prediction metric, to distinguish the best models to apply for the memetic algorithm, it might be beneficial to consider other aspects of the model. Since we do not use the model specifically for predictions, another metric might provide a more reliable way of selecting a model for the initialization phase of the memetic algorithm.

References

[1] Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Annual international cryptology conference. pp. 513–525. Springer (1997)

[2] Breiman, L., Friedman, J., Olshen, R., Stone, C.: Classification and regression trees. wadsworth int. Group **37**(15), 237–251 (1984)

[3] Brijain, M., Patel, R., Kushik, M., Rana, K.: A survey on decision tree algorithm for classification (2014)

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

[5] Charbuty, B., Abdulazeez, A.: Classification based on decision tree algorithm for machine learning. Journal of Applied Science and Technology Trends **2**(01), 20–28 (2021)

[6] Dutertre, J.M., Beroulle, V., Candelier, P., De Castro, S., Faber, L.B., Flottes, M.L., Gendrier, P., Hely, D., Leveugle, R., Maistri, P., et al.: Laser fault injection at the cmos 28 nm technology node: an analysis of the fault model. In: 2018 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC). pp. 1–6. IEEE (2018)

[7] Hooke, R., Jeeves, T.A.: " direct search" solution of numerical and statistical problems. J. ACM **8**, 212–229 (1961)

[8] Kass, G.V.: An exploratory technique for investigating large quantities of categorical data. Journal of the Royal Statistical Society: Series C (Applied Statistics) **29**(2), 119–127 (1980)

[9] Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Annual international cryptology conference. pp. 388–397. Springer (1999)

[10] Kocher, P.C.: 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. p. 104–113. CRYPTO '96, Springer-Verlag, Berlin, Heidelberg (1996)

[11] Krček, M., Fronte, D., Picek, S.: On the importance of initial solutions selection in fault injection. In: 2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC). pp. 1–12 (2021). https://doi.org/10.1109/FDTC53659.2021.00011

[12] Kumar, R., Verma, R.: Classification algorithms for data mining: A survey. International Journal of Innovations in Engineering and Technology (IJIET) **1**(2), 7–14 (2012)

[13] Loh, W.Y., Shih, Y.S.: Split selection methods for classification trees. Statistica sinica pp. 815–840 (1997)

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

[15] Mantovani, R.G., Horváth, T., Cerri, R., Vanschoren, J., de Carvalho, A.C.: Hyper-parameter tuning of a decision tree induction algorithm. In: 2016 5th Brazilian Conference on Intelligent Systems (BRACIS). pp. 37–42. IEEE (2016)

[16] Mantovani, R.G., Horváth, T., Cerri, R., Junior, S.B., Vanschoren, J., de Carvalho, A.C.P.d.L.F.: An empirical study on hyperparameter tuning of decision trees. arXiv preprint arXiv:1812.02207 (2018)

[17] Moradi, M., Oakes, B.J., Saraoglu, M., Morozov, A., Janschek, K., Denil, J.: Exploring fault parameter space using reinforcement learning-based fault injection. In: 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W). pp. 102–109. IEEE (2020)

[18] Moscato, P.: On evolution, search, optimization, genetic algorithms and martial arts - towards memetic algorithms. Caltech Concurrent Computation Program (10 2000)

[19] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)

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

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

[22] Polian, I., Gay, M., Paxian, T., Sauer, M., Becker, B.: Automatic construction of fault attacks on cryptographic hardware implementations. In: Automated Methods in Cryptographic Fault Analysis, pp. 151–170. Springer (2019)

[23] Quinlan, J.R.: Induction of decision trees. Machine learning **1**(1), 81–106 (1986)

[24] Quinlan, J.R.: C4. 5: programs for machine learning. Morgan Kaufmann Publishers (1993)

[25] Quinlan, J.R.: See5/c5. 0. http://www. rulequest. com/ (1999)

[26] Quisquater, J.J., Samyde, D.: Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In: International Conference on Research in Smart Cards. pp. 200–210. Springer (2001)

[27] Skorobogatov, S.P., Anderson, R.J.: Optical fault induction attacks. In: International workshop on cryptographic hardware and embedded systems. pp. 2–12. Springer (2002)

[28] Therneau, T., Atkinson, B., Ripley, B.: rpart: Recursive partitioning and regression trees. R package version **4**, 1–9 (2015)

[29] Werner, V., Maingault, L., Potet, M.L.: Fast calibration of fault injection equipment with hyperparameter optimization techniques. In: International Conference on Smart Card Research and Advanced Applications. pp. 121–138. Springer (2021)

[30] Wu, L., Ribera, G., Beringuier-Boher, N., Picek, S.: A fast characterization method for semi-invasive fault injection attacks. In: Cryptographers' Track at the RSA Conference. pp. 146–170. Springer (2020)