Fast Calibration of Fault Injection Equipment with Hyperparameter Optimization Techniques [⋆]

Vincent Werner[1,2], Laurent Maingault[1], and Marie-Laure Potet[2]

[1] Univ. Grenoble Alpes, CEA, LETI, DSYS, CESTI, F-38000 Grenoble, France
first.last@cea.fr
[2] Univ. Grenoble Alpes, CNRS, VERIMAG, F-38000 Grenoble, France
first.last@univ-grenoble-alpes.fr

Abstract. Although fault injection is a powerful technique to exploit implementation weaknesses, this is not without limitations. An important preliminary step, based on rigorous calibration of the fault injection equipment, greatly affects the exploitability and repeatability of injected faults. The equipment parameter space is usually explored with random search, grid search, and more recently with the help of metaheuristic algorithms. In this article, we apply, for the first time, two recent hyperparameter optimization techniques to fault injection. We evaluate these optimization techniques on three different 32-bit microcontrollers, and find better glitch waveforms than with metaheuristic algorithms. In addition, we propose a two-stage optimization strategy under black-box conditions to reduce the dimensionality of the parameter space and speed up the equipment calibration. Finally, we apply this approach to bypass the code read protection of a built-in bootloader faster than with genetic algorithms.

Keywords: Fault Injection · Voltage Glitch · Parameter Optimization

1 Introduction

Fault injection is a powerful technique to bypass security features of embedded systems, such as code protection mechanisms [8, 15, 26]. Using electrical glitches [2], focused light [31], electromagnetic pulses [13] or even nanofocused X-rays [1], one can locally perturb the chip environment to alter its behavior and gain access to critical information. Although fault injection can lead to impressive results, this is not without limitation. One of the biggest challenges is the calibration of fault injection equipment. Each fault injection equipment has multiple specific parameters that must be adjusted precisely, such as the positions x, y, z of an electromagnetic probe tip. This preliminary calibration step is required in order to find exploitable and repeatable faults.

The parameter space is often too large to be entirely covered manually during time-constrained security evaluation. The most commonly-used methods to explore the parameter space are Grid Search (GS) and Random Search (RS). GS is a semi-exhaustive search on a predetermined and progressively refined range of values. Although GS is

[⋆] This work is supported by the French National Research Agency in the framework of the "Investissements d'avenir" program (ANR-15-IDEX-02 and ANR-10-AIRT-05).

effective with small parameter space, this technique is inefficient to explore a high dimensional parameter space, as the number of evaluated configurations increases exponentially with the number of parameters considered. Even though RS is slightly better than GS for exploring large parameter space [6], both GS and RS select next configurations to evaluate independently of the previous results, thus, many evaluations are wasted on poorly-performing configurations.

Several approaches have been proposed to reduce the time spent on the equipment calibration, using more complex optimization techniques, such as metaheuristic algorithms. However, genetic and memetic algorithms are inherently chaotic and can suffer from premature convergence [19]. Accordingly, Bayesian and Bandit Optimization techniques are typically preferred over metaheuristic algorithms to optimize hard combinatorial problem solvers [16] or machine learning models [20]. To the best of our knowledge, such techniques have not been considered for fault injection yet. Therefore, in this article, we propose applying two efficient hyperparameter optimization techniques, so as to simplify and speed up the calibration of a fault injection equipment for a given target microcontroller. In addition, we also propose an optimization strategy to reduce the dimensionality of the parameter space in order to speed up even more the equipment calibration. To sum up, our contribution is threefold:

- We apply for the first time two hyperparameter optimization techniques, *Successive Halving Algorithm* (SHA) and *Sequential Model-based Algorithm Configuration* (SMAC), to find the best settings and induce repeatable and exploitable faults with our voltage fault injection (VFI) setup, on three different 32-bit microcontrollers.
- We propose breaking down the optimization problem into two stages, so as to simplify but also to speed up the equipment calibration; first, 1) during the *Calibration stage*, we focus on fault injection parameters only, using *fault characterization tests*, which are small programs running on the target device, designed to maximize fault propagation, and then, once the best configurations are identified, 2) during the *Exploitation stage*, we find the fault injection timing to exploit vulnerabilities on the target application.
- Using this strategy and SMAC, we successfully bypass the code protection mechanism of a built-in bootloader. Moreover, SMAC reduces the equipment calibration time by half compared to *Genetic Algorithm* (GA).

The outline of the rest of the article is as follows. After an overview of the related work to overcome the limitations of GS and RS in Section 2, we comprehensively explain our fault injection optimization strategy in Section 3. In Section 4, we detail SHA and SMAC optimization techniques, which are used for equipment calibration. In Section 5, to evaluate the performance of these optimization techniques, we calibrate our VFI setup for three different microcontrollers using SHA, SMAC, GA and RS. Finally, in Section 6, we apply our fault injection strategy using SMAC to bypass a read protection mechanism on a 32-bit microcontroller faster than with GA.

2 Related work

Parameter optimization has recently gained in popularity in the fault injection community. Different approaches have been proposed to speed up the equipment calibration step. When possible, reducing the parameter space by identifying the regions of interest helps considerably. For example, using a scanning electron microscope, Courbon et al. [12] find the most sensitive areas of the die to focus with Laser Fault Injection (LFI). Similarly, Schellenberg et al. [30] measure the optical beam induced current, as imaging technique, in order to localize flip-flops of an hardware AES accelerator. Madau et al. [23] propose to acquire EM emission traces, so as to detect EM hotspots and reduce the parameter space of EM Fault Injection (EMFI) equipment. Finally, to reduce the dimensionality of the problem, Carpi et al. [10] split the optimization problem into two stages, one focusing on voltage parameters and the other one on proper timing. Note that Picek et al. [28] also mention this approach, without further evaluating this idea.

Another way to find the best settings faster is to use better optimization algorithms than RS or GS. GA is a popular metaheuristic algorithm based on the evolutionary theory, which has been applied to EMFI [24] but also VFI [8, 10, 28] to find the best configurations. Picek et al. [27] use Memetic algorithm, which is an extension of the traditional GA with a local search technique, also to explore more efficiently the VFI parameter space. More recently, Wu et al. [35] have proposed a characterization method for LFI setups based on deep learning to tune the pulse width and the power of the laser.

Related Work	Optimization Technique	Dimension Reduction	FI Technique
Our contribution	Bandit Optimization	✓	VFI
	Bayesian Optimization		
[27]	Memetic Algorithm	✗	VFI
[24]	Genetic Algorithm	✗	EMFI
[8, 28]	Genetic Algorithm	✗	VFI
[10]	Genetic Algorithm	✓	VFI
[35]	Deep Learning	✗	LFI
[23]	Grid Search	✓	EMFI
[12, 30]	Grid Search	✓	LFI

Table 1: Comparison of the related work according to the optimization technique, the dimension reduction of the parameter space, and the fault injection technique.

Nevertheless, the main limitation of metaheuristic algorithms is the introduction of additional hyperparameters that must be configured, such as the size of the population, the mutation rate, or the fitness function. Moreover, depending on the optimization problem, metaheuristic algorithms can suffer from premature convergence. Similarly, finding the right number of hidden layers and neurons of the deep neural network is tedious.

More efficient optimization techniques have been proposed over the past decade, such as Bayesian optimization or Bandit optimization. Although already used for hyperparameter optimization of machine learning algorithms, these techniques have never been applied to fault injection. Accordingly, we propose for the first time to apply SMAC

(Bayesian optimization) and SHA (Bandit optimization) to improve the calibration of fault injection equipment. Moreover, we also reduce the dimensionality of the parameter space by splitting the optimization in two stages, but unlike [10], we decide to use fault characterization tests to find the best configurations.

3 Fault Injection Optimization Approach

In this section, we detail our general approach for fault injection optimization. This strategy aims to reduce the time spent on searching for the best equipment settings, by reducing the dimensionality of the parameter space. Speeding up the parameter space exploration is particularly important as security evaluations are often time-constrained.

3.1 Common Approach

The most common strategy to optimize fault injection consists to calibrate the fault injection equipment directly with the target application. But for large applications, identifying the critical sections, that can potentially lead to vulnerabilities, is tedious, therefore, it is nearly impossible during a black-box, time-constrained, security evaluation to find the right equipment settings and the right timing to inject the fault. In addition, the lack of feedback for some application further complicates the equipment calibration [33], and significantly increases the amount of work required.

3.2 Our Approach

In an effort to tackle these issues, we propose reducing the dimensionality of the parameter space by breaking down the problem of fault injection optimization into two stages, so as to simplify and speed up the parameter space exploration. First, 1) the *Calibration stage* optimizes the equipment calibration independently of the target application, using fault characterization tests and then, 2) the *Exploitation stage* finds the right timing to inject a fault in order to exploit a vulnerability on the target application. Figure 1 presents our fault injection optimization strategy.

Fault probability During the calibration stage, *only faults resulting in a faulty output are considered as effective*, while faults resulting in a crash, a timeout or a normal output are not taken into account. The *fault probability* is used as a metric to compare performance between configurations. The *fault probability* of a configuration is given by $\frac{\#\{\text{faulty results}\}}{\#\{\text{fault injected}\}}$ for this configuration.

Fault Characterization test The *fault characterization test*, is not the target application itself, but rather a series of instructions, arranged in such a way as to maximize the number of effective faults on the target microcontroller, in order to quickly find the settings with the highest fault probability. Fault characterization tests have been already applied to highlight fault effects on various microcontrollers with different fault injection techniques [4,11,14,25,29,32,34]. The main advantage of using a fault characterization

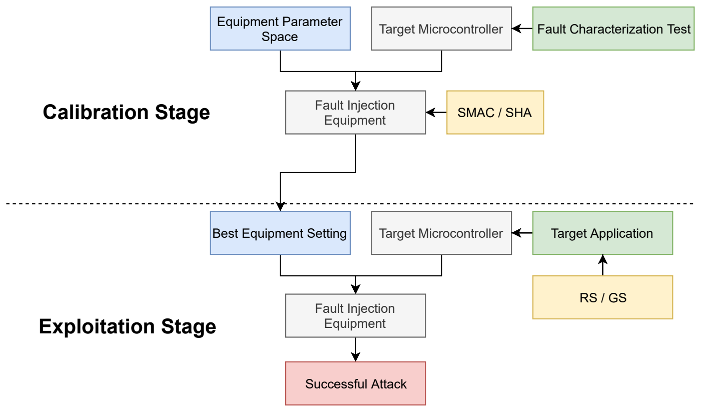

Fig. 1: Overview of our fault injection optimization strategy.

test is that we can completely ignore the injection timing during the optimization of our setup for the target microcontroller, which helps the exploration of the parameter space. In addition, a characterization test is often smaller than the target application, reducing the time required in the long run. Furthermore, a fault characterization test simplifies the equipment calibration by giving instant feedback on the effectiveness of the fault injection parameters, in comparison with an equipment calibration directly with black-box applications [33].

Optimization Techniques We use different optimization techniques for each step of our approach. During the *calibration stage*, we use hyperparameter optimization techniques such as SMAC or SHA, to quickly explore the equipment parameter space, way faster than with GS or RS (due to the curse of dimensionality [5]). Then, once the best settings are identified, the right timing to inject the fault can be found with a simple random/grid search on the target application during the *exploitation stage*.

4 Hyperparameter Optimization Techniques

In this section, we comprehensively explain the two hyperparameter optimization techniques, SHA and SMAC, which are used to improve the convergence speed towards the best fault injection settings during the *calibration stage*.

4.1 Parameter Space and Equipment Configuration

The parameter space Θ depends on the fault injection technique and the setup used. For example, our VFI setup has 9 free parameters defining the glitch waveform (8 voltage levels and the glitch duration, more detailed information is provided in Section 5.2, Figure 2). Each configuration $\theta \in \Theta$ describes how to adjust each parameter of the given fault injection equipment (e.g. the positions x, y, z of an electromagnetic probe tip). Depending on the number of equipment configurations possible within the parameter space, and the target microcontroller, the complexity of the search will vary. SHA or SMAC can significantly help to reduce the time spent identifying configurations that induce exploitable faults.

4.2 Successive Halving Algorithm

SHA has been originally proposed by Karnin et al. [18] to solve multi-armed bandits problems, but it can also be applied for hyperparameter optimization [36]. The main purpose of the algorithm (algorithm 1) is to identify the best arm correctly (the best configuration) within a fixed budget T, that is a limited amount of time or resources (e.g. the total number of fault injections). The total budget is evenly allocated across $\log_2(n)$ elimination rounds, where n is the number of initial configuration instances $\vec{\Theta}_0$. The algorithm evaluates the configurations in a uniform manner. At the end of each round, the worst ones are eliminated. Then, on each successive round, the remaining configurations are evaluated twice as much as the previous round, and the process repeats until only one remains.

Algorithm 1: Successive Halving Algorithm

```
Input: Total budget $T$, fault injection parameter space $\Theta$, $n$ initial configuration instances
       $\vec{\Theta}_0 \subset \Theta$
Output: Optimized configuration $\theta_{inc} \in \vec{\Theta}_{\lceil \log_2(n) \rceil}$
for $r = 0$ to $\lceil \log_2(n) \rceil - 1$ do
    $t_r \leftarrow \left\lfloor \frac{T}{|\vec{\Theta}_r| \lceil \log_2(n) \rceil} \right\rfloor$;
    foreach $\theta_i \in \vec{\Theta}_r$ do
        Test $t_r$ times each configuration $\theta_i$;
        Compute the empirical mean $\mu_{r,i}$ of $\theta_i$;
    $k_r \leftarrow \lceil |\vec{\Theta}_r|/2 \rceil$;
    /* Keep the $k_r^{th}$ best $\theta_i$ with the largest $\mu_{r,i}$ */
    $\vec{\Theta}_{r+1} \leftarrow$ BestKthConfigurations($\vec{\Theta}_r, k_r$);
return $\theta_{inc} \in \vec{\Theta}_{\lceil \log_2(n) \rceil}$;
```

The main concern is, for a fixed budget T, whether to consider many configurations (large n) with smaller number of trials for each (t_r); or a small number of configurations (small n) with larger number of trials for each (t_r). A solution, proposed by Aziz [3], is to take a budget $T = n \log_2(n)$, resulting in an aggressive selection of configurations after just a single shot ($\Rightarrow t_r = 1$) in the first round. Although only a conjecture has been presented to give an upper bound on the simple regret, the particular parameterization $T = n \log_2(n)$ of the algorithm 1 is better empirically than more complex solutions, also based on SHA, such as HyperBand [20].

4.3 Sequential Model-based Algorithm Configuration

SMAC, proposed by Hutter et al. [16], is a general framework for Sequential Model-Based Optimization (SMBO), also known as Bayesian Optimization. SMAC has been successfully applied for hyperparameter optimization of hard combinatorial problem solvers and various machine learning algorithms. Contrary to classical Bayesian-based approaches, SMAC supports all types of parameters, including continuous, discrete, categorical, but can also handle non-deterministic processes which is a key feature to optimize fault injection parameters. In Section 4, we will see that SMAC outperforms common approaches used to optimize the fault injection equipment. In the following, we explain the SMAC algorithm in detail.

Sequential Model-Based Optimization Unlike previous approaches, SMBO keeps track of past results to fit iteratively a probabilistic model, in order to select the next fault injection configurations which could potentially maximize the number of effective faults on the target microcontroller.

SMBO, as detailed in algorithm 2, is structured around two key components, a probabilistic model and a selection function, also called the surrogate model and the acquisition function respectively. The probabilistic model $\mathcal{M}$ is fitted (`FitModel`) to previous results $\mathbf{R} = \{(\theta_1, o_1), ..., (\theta_n, o_n)\}$ where θ_i is a possible configuration of the fault injection equipment, and o_i is the observed fault probability with configuration θ_i. The model aims to predict the fault probability o_{i+1} of a new configuration θ_{i+1} to determine if θ_{i+1} is worth being evaluated. The new configurations $\vec{\Theta}_{new}$ are selected from the fault injection parameter space Θ by the acquisition function (`SelectConfigurations`) which keeps balance between *exploitation* (sampling where the model predicts the highest fault probability) and *exploration* (sampling where the model has no prior distribution). On top of that, SMBO adds an intensification mechanism (`Intensify`), which determines 1) the budget allocated for each configuration θ_i and 2) the best known configuration so far θ_{inc} [16].

SMAC uses Random Forests (RF) as a surrogate model instead of more commonly-used Gaussian process models, which explains how SMAC supports discrete and categorical parameters. RF [9] is an ensemble method that grows many individual decision trees, which together, can be used to solve both classification and regression problems. For the latter, decision trees take continuous values (e.g. fault probability) rather than class labels at their leaves (also called *regression trees*). SMAC estimates the performance (fault probability) mean μ_θ and variance σ^2_θ for a new configuration θ by computing the empirical mean and variance of the individual regression trees prediction of the RF. By default, and to maintain a low computational cost, SMAC builds $B = 10$ regression trees with a maximum depth of 20. Each tree is grown to the largest extent possible, based on a training set of n results sampled at random with replacement from the previous results $\mathbf{R}$ (also called *bagging*). Then, at each node, m features (e.g. fault injection parameters) are randomly selected from the initial features, and the one minimizing the reduced squared sum loss among the training set is chosen to split the node.

Finally, the acquisition function of SMAC is based on Expected Improvement (EI), which is used to quantify how much a new configuration θ should improve performance

Algorithm 2: Sequential Model-Based Optimization

```
Input: Total budget $T$, fault injection parameter space $\Theta$, initial configuration instances
       $\vec{\Theta}_{init} \subset \Theta$
Output: Optimized parameter configuration $\theta_{inc}$
$\mathbf{R}, \theta_{inc} \leftarrow$ Initialize($\vec{\Theta}_{init}$);
repeat
    /* Fit the model $\mathcal{M}$ based on results $\mathbf{R}$ */
    $\mathcal{M} \leftarrow$ FitModel($\mathbf{R}$);
    /* Select promising configurations $\vec{\Theta}_{new}$ */
    $\vec{\Theta}_{new} \leftarrow$ SelectConfigurations($\mathcal{M}, \Theta$);
    /* Find the best configuration $\theta_{inc}$ */
    $\mathbf{R}, \theta_{inc} \leftarrow$ Intensify($\theta_{inc}, \vec{\Theta}_{new}$);
until total budget $T$ is exhausted;
return $\theta_{inc}$;
```

(fault probability) over our current optimum θ_{inc}. Formally, the improvement $I(\theta) = \max(f(\theta_{inc}) - f(\theta), 0)$ compares the performance between the new configuration θ with the best known configuration so far θ_{inc}. As the objective function f is unknown, EI is computed instead using the posterior distribution of θ given the predictive mean μ_θ and variance σ^2_θ obtained with RF and the empirical mean performance $f_{\theta_{inc}}$ of the best configuration seen so far [16, 17]. Next, the new configurations which yield to the highest expected improvement are selected and evaluated.

Initial Configuration Instances One main limitation of SMAC is that initial conditions can greatly affect the convergence speed, thus we propose our additional two-step procedure to select the initial configuration instances to better calibrate a given fault injection equipment. Without at least one configuration in $\vec{\Theta}_{init}$ which induces an effective fault, SMAC struggles to identify the best settings. This procedure ensures that we do not start SMAC without at least one working configuration.

- *Pure exploration*: configurations $\theta \in \Theta$ are sampled at random and tested until 1) at least k_{min} configurations that generate an effective fault have been found, **and** 2) n_{min} faults have been injected. By default, $k_{min} = 1$ and $n_{min} = 1000$.
- *Mutation*: the set $\vec{\Theta}_{init}$ of initial configuration instances includes at least the k_{min} configurations identified during the pure exploration step, **and** additional configurations generated with a gaussian mutation operator [7] using the configurations found so far, so as to reach $|\vec{\Theta}_{init}| = k_{init}$ configurations. By default, $k_{init} = 100$.

Based on the target microcontroller, k_{min}, n_{min} and k_{init} can be adjusted. For example, SMAC may struggle with some secure microcontrollers. Extending the pure exploration phase (i.e. $k_{min} > 1$ and $n_{min} > 1000$) can significantly help SMAC in early stages, especially when only a few configurations induce effective faults.

5 Equipment Calibration with Different Microcontrollers

In this section, we optimize our VFI setup for three different 32-bit microcontrollers, using SMAC, SHA, GA and RS. In these experiments, SMAC outperforms other optimization techniques and consistently identifies the best settings for our VFI setup. First, we present the target microcontrollers and general information about the experiments. Then, we detail our VFI setup and the parameter space associated. Afterwards, we compare the performance (fault probability and convergence speed) of SMAC and SHA with more commonly-used techniques, such as GA and RS.

5.1 Target Microcontrollers

We have selected three different 32-bit microcontrollers, based on different Cortex-M cores. The die of these microcontrollers are different, thus, they will not react the same way to voltage fault injections. Therefore, the best settings for our VFI setup will be different for each microcontroller. The selected microcontrollers are:

– **µC-M0** is a Cortex M0+ running at 24Mhz, based on the ARMv6-M architecture with 2 stages pipeline.
– **µC-M3** is a mainstream microcontroller based on the Cortex M3 running at 24Mhz, which implements the ARMv7-M architecture with 3 stages pipeline.
– **µC-M4** is ultra-low-power microcontroller based on the Cortex M4, running at 72Mhz. The core is based on the ARMv7E-M architecture with 3 stages pipeline and branch speculation.

5.2 Setup

General Information During the *Calibration Stage* (Figure 1), we use the fault characterization test detailed in Table 2. This test has been designed to maximize the propagation of bit-set or bit-reset on the fetched instruction, but also instruction skips (not detailed in this study). For each optimization technique (SMAC, SHA, GA and RS), we inject 50,000 faults ($\approx$ 6 hours). For SMAC, we use the Python library SMACv3 [21], and more precisely the class `SMAC4HPO`. For SHA, GA and RS, we do not use an external library.

For SHA, as described in Section 3, we use the parameterization $T = n \log_2(n)$, with $n = 4096$. For GA, each individual of the population represents a valid configuration of the fault injection equipment considered. We train a population of 50 individuals over 200 generations, where each individual is tested five times. In addition, we use a gaussian mutation operator [7], a roulette-wheel selection via stochastic acceptance [22] and the fitness of an individual is given by its fault probability. For RS, we evaluate 10,000 configurations, where each configuration is tested five times. In the following, we detail our VFI setup and its associated parameter space.

Instruction Corruption (IC) Test			

`adds r2, #1`
`subs r7, #ff`
`adds r2, #1`
`subs r7, #ff`
	Repeat *n* times		

R0	`0x00000000`	R1	`0x11111111`
R2	`0x22222222`	R3	`0x33333333`
R4	`0x44444444`	R5	`0x55555555`
R6	`0x66666666`	R7	`0x77777777`

Table 2: Instruction Corruption (IC) Test for the ARMv7-M instruction set, as well as the initial values of registers.

Voltage Fault Injection Setup Our VFI setup is similar to the Bozzato et al. [8] test bench. We use a custom 30 MSps Digital-to-Analog Converter (DAC) to generate arbitrary glitch waveforms instead of an external arbitrary waveform generator. The DAC is a simple R–2R ladder with 8-bit resolution, which converts digital input byte into analog output voltage. The glitch waveform, sent to the DAC, is generated with a function that takes a set of 8 instantaneous voltage levels, that are then interpolated with cubic interpolation on a grid, up to 2048-by-256, that depends on the waveform size requested. This setup is cheap (≈ 100$) and yet offers great versatility to adapt to different targets with the ability to generate a large spectrum of glitch waveforms ([8]).

However, the versatility comes at a price, as the parameter space of our VFI setup, presented in Figure 2, is larger than those of more commonly-used VFI setups. Indeed, most of the time, only two parameters are used (glitch duration and glitch amplitude), while our setup has 9 free parameters (8 voltage levels and the glitch duration). Therefore, our VFI setup is a good candidate to evaluate the relevance of SMAC and SHA optimization techniques.

		Parameter	
		$x_0 \ldots x_7$	*duration*
Digital	Range	$[\![80, 180]\!]$	$[\![8, 128]\!]$
	Resolution	8-bit	8-bit
Analog	Range	$[0.6, 5.1]$	$[0.2, 3.2]$
	Unit	V	µs

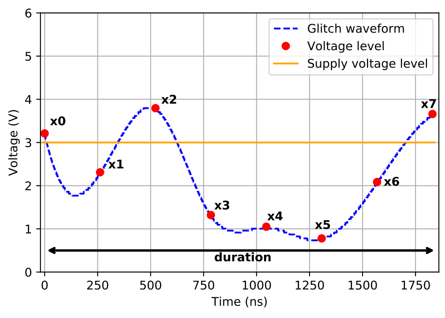

Fig. 2: VFI parameter space, ≈ 10^{18} configurations. The glitch waveform is defined with 8 voltage levels ($x_0 \ldots x_7$) and the duration.

5.3 Experimental Protocol

The results of the fault injection optimization with SMAC, SHA, GA and RS are heterogeneous. While SMAC and SHA, by design, return a single configuration (the best found), RS and GA return several configurations. Indeed, SMAC and SHA progressively increase the number of test to better approximate the fault probability in order to select the best configuration whereas RS and GA always evaluate each configuration the same number of times, thus several configurations can end up with the same fault probability. Accordingly, to fairly compare the fault probability evolution over time of the configuration(s) found with SMAC, SHA, GA and RS, several considerations have to be taken into account:

- *SMAC*: by design, with SMAC, the best configuration known so far is updated during runtime execution, thus no post-processing required.
- *RS*: unlike SMAC, post-processing is required for RS. Every 5000 fault injections, we inject 1000 more faults to evaluate the fault probability of the best configuration(s) found so far.
- *GA*: The same post-processing as RS is required.
- *SHA*: We evaluate the average fault probability at each halving of the remaining configurations.

For each microcontroller considered, we optimize our VFI setup using SMAC, SHA, GA and RS and we compare the fault probability evolution over time of the configuration(s) found. The best optimization technique is the one that finds the configuration with the highest fault probability, within a minimum number of fault injections.

5.4 Results

The results of the experiments are summarized in Figure 3 and Table 3. In the Figure 3, we compare the evolution of fault probability over 50,000 fault injections, to visually determine the convergence speed of each optimization technique (fast or slow). Table 3 presents the fault probability of the best settings found with each technique.

For each microcontroller, SMAC is significantly faster than other optimization techniques. In particular, in less than 10,000 fault injections, SMAC systematically identifies configurations with higher fault probability than GA, RS and SHA. Therefore, SMAC can be used to calibrate an equipment faster than more commonly-used optimization techniques, hence saving valuable time during security evaluations. On the other hand, SHA slowly converges towards the best configuration. However, at the end, after 50,000 fault injections, SHA finds the configuration with the best fault probability for µC-M0 and µC-M3.

By design, SHA uses all the allocated budget T, and removes iteratively the worst configurations at each round, which explains the slow convergence speed, in comparison with other optimization techniques. Nevertheless, we find that SHA wastes many evaluations on poorly-performing configurations during the first rounds, in particular with µC-M0. Our additional procedure for SMAC, described in Section 4.3, could also help SHA to select the initial configuration instances $\vec{\Theta}_0$, so as to reduce the time spent

		SMAC	SHA	GA	RS
µC-M0	Max Fault Probability	0.52	**0.53**	0.49	0.49
	Convergence Speed	**Fast**	Slow	**Fast**	Slow
µC-M3	Max Fault Probability	0.77	**0.81**	0.52	0.24
	Convergence Speed	**Fast**	Slow	Slow	Slow
µC-M4	Max Fault Probability	**0.95**	0.79	0.81	0.71
	Convergence Speed	**Fast**	Slow	**Fast**	Slow

Table 3: Performance comparison between optimization techniques.

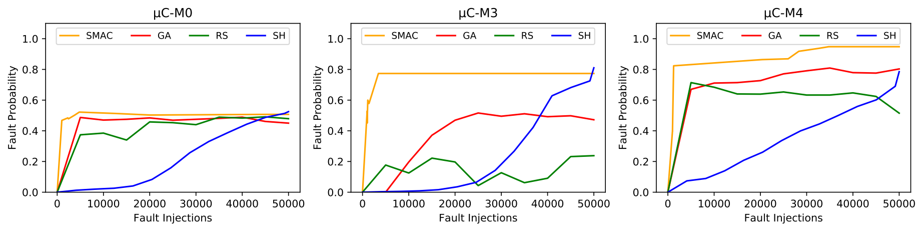

Fig. 3: Evolution of fault probability over 50,000 fault injections, according to SMAC, GA, SHA and RS, with VFI

on poorly-performing configurations. Although we have not evaluated SMAC or SHA with other fault injection techniques, we believe that these optimization techniques can be easily adaptable to EMFI or LFI. Regarding the results, SMAC is more efficient than GA, RS, and SHA, in particular to quickly calibrate fault injection equipment for a given microcontroller. In the following, we will show that SMAC can also be used to exploit vulnerabilities faster than GA.

6 SMAC to Bypass a Code Protection Mechanism

In this section, we apply our two-stage strategy with SMAC to bypass a code protection mechanism, with VFI, on a 32-bit microcontroller. The presented attack is a known attack [8] which downgrades the security level of the target, so as to extract the firmware. We will show that SMAC is better than GA at identifying the best settings within a limited number of fault injections, and therefore that SMAC can save valuable time during security evaluations.

6.1 STM32F103RB

The microcontroller STM32F103RB is a 32-bit ARM Cortex-M3 core operating at 24MHz. The preprogrammed bootloader offers code protection mechanisms to prevent any read or write operations from the bootloader on the user flash memory. In practical terms, once the read protection (RDP) is enabled, the bootloader returns a negative response (NACK) when a Read Memory command is issued. To disable RDP, the flash must be completely erased.

Attack The known attack [8] to bypass the read protection mechanism consists in injecting a fault during the Read Memory command. Indeed, when the bootloader receives the Read Memory command, it checks the RDP value and returns the ACK or the NACK byte, depending on whether RDP is disabled or enabled, respectively. By injecting a fault during the RDP checking phase, an attacker can deceive the read protection mechanism and retrieve the content of the selected memory block.

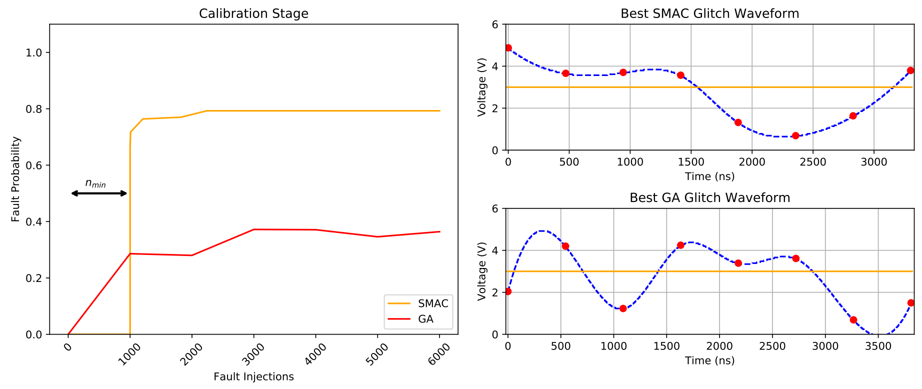

Fig. 4: Evolution of fault probability over 6000 fault injections, according to SMAC and GA, on the STM32F103RB; and the best glitch waveforms found with SMAC and GA during the calibration stage.

Calibration Stage In order to find the best settings for our VFI equipment to glitch the STM32F103RB, we will use SMAC and GA, and compare the fault probability evolution. For both SMAC and GA, we perform the *calibration stage* with 6000 fault injections (24 generations for GA) during $\approx$ 15 minutes, with the fault characterization test in Table 2, and with the default parameters. Figure 4 presents the fault probability evolution over time of the best configuration(s) found with SMAC and GA. We have arbitrarily chosen a small number of fault injections during the *calibration stage*, so as to show that SMAC is definitely faster at identifying the best settings than more commonly-used optimization techniques, such as GA. Not only does SMAC converge faster than GA, but SMAC also identifies configurations twice as efficient as those found with GA (Table 4).

Exploitation Stage We compare the average of the elapsed time to perform the attack to bypass RDP (*exploitation stage*) with SMAC and GA, using the best glitch waveforms

		Number of Fault Injections	
		6000	12000
SMAC	Max Fault Probability	**0.79**	0.79
	Calibration Time	15 min	30 min
	Exploitation Time	**<5 min**	<5 min
GA	Max Fault Probability	0.37	0.55
	Calibration Time	15 min	30 min
	Exploitation Time	N/A	<5 min

Table 4: Performance comparison between SMAC and GA on the STM32F103RB with VFI.

found during the *calibration stage*. The attack is easily achieved with the best configuration found with SMAC, *on average in less than 5 minutes*. On contrary, with the best configurations found with GA, we have not been able to bypass the read protection mechanism of the STM32F103RB. This shows that with only 6,000 fault injections during the *calibration stage*, GA clearly underperforms SMAC. Figure 5 presents the oscilloscope traces of the attack to bypass RDP on the STM32F103RB, using the best glitch waveform found with SMAC.

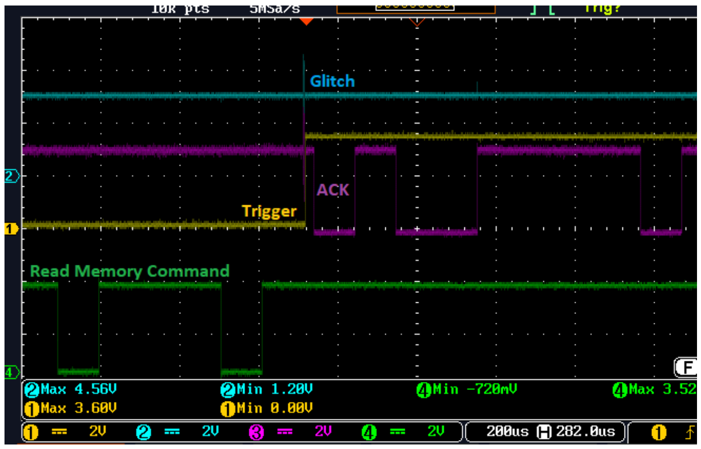

Fig. 5: Oscilloscope traces of the glitch attack to bypass RDP on the STM32F103RB.

Note that with a larger number of fault injections during the calibration stage, it is also possible to bypass RDP using GA. For example, with twice as many fault injections during the calibration stage (i.e. 12,000 instead of 6,000), GA identifies equipment settings that can successfully glitch the STM32F103RB and bypass the code protection mechanism (Table 4). But even after 12.000 fault injections, the configurations identified with GA have a lower fault probability than with SMAC.

7 Conclusion

Fault injection requires a preliminary step of equipment calibration in order to find exploitable and repeatable faults. In this article, we have proposed applying state-of-the-art optimization techniques, already used for machine learning and other hard combinatorial problems, to fault injection. Bayesian Optimization (SMAC) and Bandit Optimization (SHA) are used to identify the best equipment configurations which maximize exploitable faults on a target microcontroller. While SHA is a simple algorithm, easily adaptable to fault injection and yet offers decent performance, SMAC is arguably the most interesting optimization technique, finding better equipment configurations faster than metaheuristic algorithms.

In addition, to simplify and speed up the equipment calibration, we have proposed splitting fault injection optimization into two stages, *the calibration stage* and *the exploitation stage*. We optimize fault injection parameters independently of the target application with a fault characterization test and then, once the best configurations are identified, we find fault injection timings to exploit vulnerabilities on the target application. With SMAC and this strategy, we successfully bypass a code protection mechanism of the STM32F103RB bootloader. In particular, the calibration stage with SMAC is twice as fast as with GA. Furthermore, SMAC and SHA have systematically identified better configurations than metaheuristic algorithms, and although it has not been studied in this article, finding configurations with high fault probability is even more important when multi-fault injections are necessary, as inducing more repeatable faults greatly help in carrying out complex multi-fault attacks.

As future work, it will be interesting to apply other promising optimization techniques such as HyperBand (Bandit Optimization) or Tree-structured Parzen Estimator (Bayesian Optimization). Moreover, we will investigate the applications of hyperparameter optimization techniques to find exploitable faults with other fault injection techniques, such as LFI or EMFI. Finally, our ongoing research is focused on direct applications of fault injection optimization with SMAC or SHA on secure microcontrollers. For example, we believe that we can find exotic waveforms with SMAC that can bypass voltage glitch attack detectors.

References

1. Stéphanie Anceau, Pierre Bleuet, Jessy Clédière, Laurent Maingault, Jean-luc Rainard, and Rémi Tucoulou. Nanofocused x-ray beam to reprogram secure circuits. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 175–188. Springer, 2017.
2. Christian Aumüller, Peter Bier, Wieland Fischer, Peter Hofreiter, and J-P Seifert. Fault attacks on rsa with crt: Concrete results and practical countermeasures. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 260–275. Springer, 2002.
3. Maryam Aziz. *On Multi-Armed Bandits Theory and Applications*. PhD thesis, Ph. D. Thesis, Northeastern University, Boston, MA, USA, 2019.

4. Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 105–114. IEEE, 2011.
5. Richard E Bellman. *Adaptive control processes*. Princeton university press, 1861.
6. James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.
7. Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies–a comprehensive introduction. *Natural computing*, 1(1):3–52, 2002.
8. Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the glitch: optimizing voltage fault injection attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 199–224, 2019.
9. Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.
10. Rafael Boix Carpi, Stjepan Picek, Lejla Batina, Federico Menarini, Domagoj Jakobovic, and Marin Golub. Glitch it if you can: parameter search strategies for successful fault injection. In *International Conference on Smart Card Research and Advanced Applications*, pages 236–252. Springer, 2013.
11. Brice Colombier, Alexandre Menu, Jean-Max Dutertre, Pierre-Alain Moëllic, Jean-Baptiste Rigaud, and Jean-Luc Danger. Laser-induced single-bit faults in flash memory: Instructions corruption on a 32-bit microcontroller. *IACR Cryptology ePrint Archive*, 2018:1042, 2018.
12. Franck Courbon, Philippe Loubet-Moundi, Jacques JA Fournier, and Assia Tria. Increasing the efficiency of laser fault injections using fast gate level reverse engineering. In *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 60–63. IEEE, 2014.
13. Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria. Electromagnetic transient faults injection on a hardware and a software implementations of aes. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 7–15. IEEE, 2012.
14. Louis Dureuil, Marie-Laure Potet, Philippe de Choudens, Cécile Dumas, and Jessy Clédière. From code review to fault injection attacks: Filling the gap using fault model inference. In *International conference on smart card research and advanced applications*, pages 107–124. Springer, 2015.
15. Chris Gerlinsky. Breaking code read protection on the nxp lpc-family microcontrollers, 2017.
16. Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.
17. Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Kevin P Murphy. An experimental investigation of model-based parameter optimisation: Spo and beyond. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 271–278, 2009.
18. Zohar Karnin, Tomer Koren, and Oren Somekh. Almost optimal exploration in multi-armed bandits. In *International Conference on Machine Learning*, pages 1238–1246. PMLR, 2013.
19. Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. A review on genetic algorithm: past, present, and future. *Multimedia Tools and Applications*, pages 1–36, 2020.
20. Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.
21. Marius Lindauer, Katharina Eggensperger, Matthias Feurer, Stefan Falkner, André Biedenkapp, and Frank Hutter. Smac v3: Algorithm configuration in python. `https://github.com/automl/SMAC3`, 2017.
22. Adam Lipowski and Dorota Lipowska. Roulette-wheel selection via stochastic acceptance. *Physica A: Statistical Mechanics and its Applications*, 391(6):2193–2196, 2012.

23. Maxime Madau, Michel Agoyan, and Philippe Maurine. An em fault injection susceptibility criterion and its application to the localization of hotspots. In *International Conference on Smart Card Research and Advanced Applications*, pages 180–195. Springer, 2017.
24. Antun Maldini, Niels Samwel, Stjepan Picek, and Lejla Batina. Optimizing electromagnetic fault injection with genetic algorithms. In *Automated Methods in Cryptographic Fault Analysis*, pages 281–300. Springer, 2019.
25. Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 77–88. IEEE, 2013.
26. Johannes Obermaier and Stefan Tatschner. Shedding too much light on a microcontroller's firmware protection. In *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017.
27. Stjepan Picek, Lejla Batina, Pieter Buzing, and Domagoj Jakobovic. Fault injection with a new flavor: Memetic algorithms make a difference. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 159–173. Springer, 2015.
28. Stjepan Picek, Lejla Batina, Domagoj Jakobović, and Rafael Boix Carpi. Evolving genetic algorithms for fault injection attacks. In *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1106–1111. IEEE, 2014.
29. Lionel Riviere, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer, and Laurent Sauvage. High precision fault injections on the instruction cache of armv7-m architectures. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 62–67. IEEE, 2015.
30. Falk Schellenberg, Markus Finkeldey, Bastian Richter, Maximilian Schäpers, Nils Gerhardt, Martin Hofmann, and Christof Paar. On the complexity reduction of laser fault injection campaigns using obic measurements. In *2015 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 14–27. IEEE, 2015.
31. Sergei P Skorobogatov and Ross J Anderson. Optical fault induction attacks. In *International workshop on cryptographic hardware and embedded systems*, pages 2–12. Springer, 2002.
32. Thomas Trouchkine, Guillaume Bouffard, and Jessy Clédière. Fault injection characterization on modern cpus. In *IFIP International Conference on Information Security Theory and Practice*, pages 123–138. Springer, 2019.
33. Jan Van den Herrewegen, David Oswald, Flavio D Garcia, and Qais Temeiza. Fill your boots: Enhanced embedded bootloader exploits via fault injection and binary analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 56–81, 2021.
34. Vincent Werner, Laurent Maingault, and Marie-Laure Potet. An end-to-end approach for multi-fault attack vulnerability assessment. In *2020 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, pages 10–17. IEEE, 2020.
35. Lichao Wu, Gerard Ribera, Noemie Beringuier-Boher, and Stjepan Picek. A fast characterization method for semi-invasive fault injection attacks. In *Cryptographers' Track at the RSA Conference*, pages 146–170. Springer, 2020.
36. Li Yang and Abdallah Shami. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, 415:295–316, 2020.