Modelling Delay-based Physically Unclonable Functions through Particle Swarm Optimization

Nimish Mishra
neelam.nimish@gmail.com
Indian Institute of Technology
Kharagpur, West Bengal, India

Kuheli Pratihar
its.kuheli96@gmail.com
Indian Institute of Technology
Kharagpur, West Bengal, India

Anirban Chakraborty
ch.anirban00727@gmail.com
Indian Institute of Technology
Kharagpur, West Bengal, India

Debdeep Mukhopadhyay
debdeep.mukhopadhyay@gmail.com
Indian Institute of Technology
Kharagpur, West Bengal, India

ABSTRACT

Recent advancements in low-cost cryptography have converged upon the use of nanoscale level structural variances as sources of entropy that is unique to each device. Consequently, such delay-based *Physically Unclonable Functions* or (PUFs) have gained traction for several cryptographic applications. In light of recent machine learning (ML) attacks on delay-based PUFs, the common trend among PUF designers is to either introduce non-linearity using XORs or input transformations applied on the challenges in order to harden the security of delay-based PUFs. Such approaches make machine learning modelling attacks hard by destroying the linear relationship between challenge-response pairs of a PUF. However, we propose to perceive PUFs, which are fundamentally viewed as Boolean functional mapping, as a set of delay parameters drawn from normal distribution. Using this newfound perception, we propose an alternative attack strategy in this paper. We show that instead of trying to learn the exact functional relationship between challenge-response pairs from a PUF, one can *search* through the search space of all PUFs to find *alternative* PUF delay parameter set that exhibits "similar" behaviour as the target PUF. The core intuition behind this strategy is that one can consider a PUF as a set of stages wherein, depending on the corresponding input challenge bit, one of the several signals within a PUF's stage win a *race* condition. To utilize this idea, we develop a novel Particle Swarm Optimization based framework inspired by the biomimicry of amoebic reproduction. The proposed algorithm, called `CalyPSO`, avoids the pitfalls of textbook Genetic Algorithms and demonstrates complete break of existing delay-based PUFs which are based on arbiter chains. More specifically, we are able to model higher-rder k-XOR PUF variants which are resistant to all-known ML modelling techniques, including $k = 13, 15$ and 20, without the knowledge of reliability values. In addition to that, we also model PUFs that incorporate input transformation, like variants of IPUF and LP-PUF. Furthermore, we take forward this idea across different search spaces in order to learn a higher order PUF using a lower order (and simpler) PUF architecture. This allows us to explore a novel class of attacks, including modelling a k-`XOR` PUF using a $(k-1)$-`XOR` PUF as well as bypassing input transformations based PUF designs.

1 INTRODUCTION

1.1 Physically Unclonable Functions

The advent of ubiquitous computing and its tremendous growth and pervasiveness in the last couple of decades has opened new arenas and challenges for the security and integrity of resource-constrained devices. Being deployed in-the-wild as edge devices, they are susceptible to a multitude of attack surfaces and possibilities, including physical and side-channel attacks [18, 21]. In such scenarios, classical cryptography, which builds its security claims by assuming the non-availability of secret key to the attacker, does not fulfil the practical requirements in presence of such strong adversarial threat models. In this context, *Physically Unclonable Functions (PUFs)* [9, 30] have shown great promise and received interest from the security research community due to their inherent feature of being "keyless", thereby circumventing the threat of physical attacks that could leak the key. A PUF can be viewed as a physical system that relies on intrinsic hardware randomness as the source of entropy. Given a challenge $\mathbf{c}$ as external stimulus, a PUF is essentially an activated hardware component that depends on nanoscale structural variances like multiplexer delays [9, 16, 30], ring oscillators [15], start-up values of an static random access memory (SRAM) [3, 38] and so on, to produce output as response r.

1.2 Modelling attacks on PUFs

Although PUFs, by design, are supposed to be unclonable, there have been modelling attacks on them. In this section, we briefly explain the reason behind the same. PUFs have been subjected to modelling attacks where an adversary tries to create an algorithmic model which can predict PUF's response to arbitrary challenges with high probability. Traditionally, machine learning (ML) techniques have been used in literature as a tool for such modelling attacks. We begin with the simplest of delay-based PUFs: an Arbiter PUF (APUF) [1] An APUF takes as input a single challenge $c = (c_1, c_2, c_3, ..., c_n) \in \{-1, 1\}^n$ where n is the challenge length, and outputs a single bit response $r \in \{-1, 1\}$. The algebraic representation of 1 and -1 corresponds to the 0 (`LOW`) and 1 (`HIGH`) states in digital logic respectively. The additive delay model forms the following relationship:

[1]We only consider APUF-based designs as it is the most commonly used PUF architecture that forms the basis of several constructions in the literature. Some other PUF architectures like PAPUF [16] and BR-PUFs [4] also exist in the literature but have simpler delay model when compared to an APUF.

$$\Delta = \sum_{i=1}^{n} \left(\delta_{c_i}^{(i)} \prod_{j=i}^{n} c_j \right) \tag{1}$$

wherein Δ controls the value r takes, i.e. $\mathcal{G}(\Delta) = r$ with $\mathcal{G}$ being the *sign* function. In this model, $\delta_1^{(i)}, \delta_{-1}^{(i)}$ represents i-th delay parameter in case of $\mathbf{c}_i = 1, \mathbf{c}_i = -1$ respectively. Note that δ is an encapsulation of how the i-th stage is excited by the input challenge bit c_i. It can therefore be decomposed into its constituent parameters, through which it is possible to linearize this equation wrt. $\Phi_i = \prod_{j=i}^{n} c_j$ by introducing a new variable ω [35].

$$\Delta = \sum_{i=1}^{n} \omega_i \Phi_i \tag{2}$$

where $\omega_i = \frac{1}{2}(\delta_{-1}^{i} + \delta_1^{i} - \delta_{-1}^{i-1} + \delta_{-1}^{i-1})$. Note that Φ is the parity vector which is derived from the input challenge $\mathbf{c}$. Hence, a modelling attack has to successfully model $\boldsymbol{\omega}$ in order to successfully model the target PUF. This objective can be easily achieved by any machine learning algorithm that relies upon the successful separation of linearly separable inputs.

1.3 Defences against ML attacks on PUFs

A successful defence against PUF modelling attacks would involve *de-linearizing* $\Delta = \sum_{i=1}^{n} \omega_i \Phi_i$, i.e. breaking the linear relationship of Δ with $\boldsymbol{\omega}$, making it harder for machine learning to learn a polynomialy separable decision boundary on $\mathcal{G}(\Delta)$.

One method to de-linearize the aforementioned equation is to explicitly introduce non-linearity without changing the underlying APUF model [20]. This is typically achieved by introducing several APUF chains and combining them with an XOR. A k-XOR PUF consists of k arbiter chains and computes the XOR of k individual responses. Each of the APUFs are given an input challenge $c = (c_1, c_2, c_3, ..., c_n) \in \{-1, 1\}^n$ and the XORed output is response r. Representing a k-XOR PUF in terms of the functional mapping $\mathcal{G}$ gives us the following equation:

$$r = \mathcal{G}(\Delta_1) \oplus \mathcal{G}(\Delta_2) \oplus \mathcal{G}(\Delta_3) \oplus \cdots \oplus \mathcal{G}(\Delta_k) \tag{3}$$

where each APUF is represented using a mapping $\mathcal{G}$ and respective Δ and therefore can be further expanded using Eqn. 1 into:

$$r = \mathcal{G}\left(\sum_{i=1}^{n} \left((\delta_1)_{c_i}^{(i)} \prod_{j=i}^{n} c_j \right) \right) \oplus \cdots \oplus \mathcal{G}\left(\sum_{i=1}^{n} \left((\delta_k)_{c_i}^{(i)} \prod_{j=i}^{n} c_i \right) \right) \tag{4}$$

In a k-XOR APUF, k different APUF chains are XORed together to generate one bit of response. Any arbitrary APUF chain has its own delay parameter set δ_i, $1 \leq i \leq k$. A modelling attack tries to learn this non-linearity as well, which is a far more difficult problem.

One other way of achieving the same objective is to borrow design principles of block ciphers to *hide* the otherwise publicly available challenge bits by introducing input transformations [17, 25, 34]. Concretely, given a challenge $\mathbf{c} = \{c_1, c_2, \cdots, c_n\}$, such defences implement a one-way function $f_s(\mathbf{c})$ parameterized by some secret s such $\mathbf{c}$ is converted into a *private* challenge $\mathbf{c}' = \{c_1', c_2', c_3', ..., c_n'\}$, which is then passed as input to the PUF. Formally, we write a generic k-XOR equation in presence of input transformations as follows:

$$r = \mathcal{G}\left(\sum_{x=1}^{n} (\delta_1)_{f_s(c)_x}^{(x)} \prod_{y=x}^{n} f_s(c)_y \right) \oplus \\ \ldots \oplus \mathcal{G}\left(\sum_{x=1}^{n} (\delta_k)_{f_s(c)_x}^{(x)} \prod_{y=x}^{n} f_s(c)_y \right) \tag{5}$$

Any modelling attack on such PUFs implementing input transformations has to learn both f_s as well as the non-linear XOR.

1.4 Motivation

The *cat and mouse game* between PUF designers and attackers has led to the development of higher order complexity PUFs such as k-XOR PUFs and LP-PUFs that are fundamentally variants of APUF. While the attackers attempted to use different ML techniques to learn the challenge to response mapping by learning the delay model, the designers focused on increasing the non-linearity and diffusion of challenges. In short, the attack strategy can be summarized as the following observation.

✓ **O1**. For an arbitrary PUF specification, challenge set $C \in \mathcal{U}_C$, and original response set $\mathcal{R} \in \mathcal{U}_R$, a machine learning algorithm dwells on a search space of functions $\mathcal{P} : \mathcal{U}_C \rightarrow \mathcal{U}_R$. Here, $\mathcal{U}$ represents the universal set notation.

The observation **O1** helps explain the motivation behind the two design approaches undertaken to counter ML attacks (as discussed in Sec. 1.3). Since a machine learning algorithm fundamentally *searches* for a specific function in the search space of functions $\mathcal{P}$, mapping the set of challenges $\mathcal{U}_C$ to the set of responses $\mathcal{U}_R$, if this mapping $\mathcal{P}$ is *obfuscated* by breaking the linear relationship of $\Delta = \sum_{i=1}^{n} \omega_i \Phi_i$, the searching process would not be able to converge successfully. This happens since machine learning can no longer find a linear separation between the two kinds of output responses (1 and −1). Therefore, the "tug-of-war" between the attacker and designer communities focussed mainly on the perceived hardness of the problem of finding the mapping $\mathcal{P} : \mathcal{U}_C \rightarrow \mathcal{U}_R$ by *de-linearization*. However, one might be curious to ponder if there exists a more efficient way of attacking delay-based PUFs other than directly learning the $\mathcal{U}_C \rightarrow \mathcal{U}_R$ mapping through ML.

We note that the nanoscale structural variances of a delay-based PUF can be modelled by a normal distribution with suitable variance. This includes the inherent delays of CMOS circuitry [12] as well as additional noise that arises in hardware [6]. In this work, we utilize this information to form a new perspective towards the modelling of delay-based PUFs. In other words, we express PUFs as a Boolean function composed of a set of normal random variables $\boldsymbol{\delta} = \{\delta_1, \delta_2, \delta_3, ..., \delta_n\}$ where $\delta_i \sim \mathcal{N}(0, \sigma^2) \, \forall \, 1 \leq i \leq n$ for standard deviation σ. We note that every delay-based PUF, including APUF, generates responses depending *not on individual stage delays, but rather on the combined effect of those delays*. Formally, this *combined effect* phenomenon can be expressed as $r = \mathcal{G}(\Delta)$, where Δ is the *delay model* signifying the conbined effect of all delay parameters $\{\delta_1, \delta_2, \delta_3, ..., \delta_n\}$. This observation constitutes the basis of our attack strategy that we propose in this paper.

✓ **O2**. A successful strategy to model a PUF would require approximating the *combined effect* Δ by constructing another set of normal variables $\boldsymbol{\delta}^{'} = \{\delta_1^{'}, \delta_2^{'}, \delta_3^{'}, ..., \delta_n^{'}\}$ such that $\delta_i^{'} \sim \mathcal{N}(0, \sigma^2) \forall 1 \leq i \leq n$, it can construct another *combined effect* function $\mathcal{G}^{'}$ corresponding to the modelled *combined delay* $\Delta^{'}$ such that $r = \mathcal{G}(\Delta) = \mathcal{G}^{'}(\Delta^{'})$.

A logical inference of observation **O2** allows the adversary to claim that $\mathcal{G}^{'}$ is a *model* of the $\mathcal{G}$, thereby allowing a successful model of the target PUF. Therefore, instead of *searching* over a function space $\mathcal{P} : \mathcal{U}_C \rightarrow \mathcal{U}_R$, we *search* over the space of PUFs parameterized by the normal distribution $\mathcal{N}(0, \sigma^2)$. Furthermore, we can formalize the *successful* modelling of a PUF by establishing a sufficiently *small* threshold ϵ such that following equation holds:

$$| \Delta - \Delta^{'} | \leq \epsilon$$

Moreover, treating the problem of modelling PUFs as a search problem allows us to launch novel *downgrade* attacks (c.f. Sec. 5 for details). This happens because the specific search space to work upon depends on the underlying PUF architecture being targeted. The foundation of our *downgrade* attacks look to approximate a target delay Δ by searching in a search space belonging to *simpler* PUF. In this work, we classify such attacks into two types:

- **Degrees of freedom reduction attack**: wherein we reduce the security of a k-XOR PUF into a $(k-1)$-XOR PUF.
- **Input transformation bypass attack**: wherein we reduce the security of PUFs implementing input transformations to equivalent k-XOR PUFs without any input transformations.

Both types of downgrade attacks are possible because our perspective of looking at PUF modellings from the viewpoint of a *search* problem on delay parameter set δ allows a richer search space. For instance, **Degrees of freedom reduction attack** relies upon the event of searching over a $(k-1)$-XOR PUF search space, and finding *some* $(k-1)$-XOR instance with delay $\Delta^{'}$ that behaves *similar* to a target k-XOR instance with delay Δ.

1.5 Contributions

In this work, we make the following contributions:

❶ We introduce a new model of attacking delay-based PUFs by focusing not on learning the challenge-response functional mapping, but rather by learning the delay parameters themselves. This allows us to reason out failures of machine learning into certain kinds of APUF variants, prompting our study of evolutionary algorithms in the context of PUFs.

❷ We look into prior works using textbook genetic algorithms (GA), develop a bound on the probability of success of textbook genetic algorithm and show how certain properties of a textbook GA are unsuitable in the context of our attack vectors.

❸ Motivated by our observations on the reasons of failures of textbook genetic algorithm, we develop a novel variant of Particle Swarm Optimization algorithm inspired by the natural process of amoebic asexual reproduction.

❹ Our proposed algorithm, to the best of our knowledge, is the first attempt to model higher order k-XOR PUFs (as high as 20-XOR PUFs) using far less number of challenge-response pairs.

❺ Our proposed algorithm, to the best of our knowledge, is also the first attempt to attack delay-based PUFs deriving their security from input transformations. One prime example is LP-PUF, which has been successfully modelled yet.

❻ We also introduce a novel circuit composition of a k-XOR PUF that allows us to reduce its security to a $(k-1)$-XOR PUF.

❼ We additionally put forward a novel attack strategy allowing us to ① reduce the security of a higher order k-XOR PUF variant to a lower order $(k-1)$-XOR PUF, and ② bypass complex input transformations like substitution permutation networks (as is the case with LP-PUF) allowing us to successfully model LP-PUF.

Organization: In Sec. 2, we introduce background information about PUFs and prior attacks on them, provide an overview of evolutionary algorithms (EA) which we build upon later. In Sec. 3, we note prior attempts to use EAs to attack PUFs and provide a mathematical reasoning for their failure. In Sec. 4, we then propose a novel algorithm to attack PUFs, based on Particle Swarm Optimization. Furthermore, in Sec. 5, we provide novel insights allowing cross-architectural modelling attacks on PUFs, which coupled with the majority voting rationale presented in Sec. 6 speeds up convergence. We then present experimental results in Sec. 7 and compare them with prior attacks in Sec. 8. Finally, we conclude with future research directions in Sec. 9.

2 PRELIMINARIES

2.1 Arbiter PUF and its XOR composition

Arbiter PUF (APUF) is the most popularly used delay-based PUF [10]. The architecture of an APUF consists of two branches, the delay of which are determined based on the given challenges (c.f. Fig. 1a). An arbiter, in the end, compares the delay of two branches and evaluates the response. An APUF takes as input a single challenge $\mathbf{c} = (c_1, c_2, c_3, ..., c_n) \in \{-1, 1\}^n$ where n is the challenge length, and outputs a single bit response $r \in \{-1, 1\}$. The algebraic representation of $1, -1$ corresponds to the $0, 1$ (LOW,HIGH) states in digital logic. The APUF consists of path swapping switches which leads to the delay difference Δ being proportional to the dot-product between weight vector $\boldsymbol{\omega}$ and parity vector Φ.

$$\Phi_j = \prod_{i=j}^{n} \mathbf{c}_j; \Delta = \langle \boldsymbol{\omega}, \boldsymbol{\Phi} \rangle; r = sign(\Delta) \tag{6}$$

The linear representation of APUF makes it vulnerable to modelling attacks, therefore complex compositions such as XOR-APUFs and LP-PUFs were proposed.

2.1.1 ***XOR-APUF.*** One method to de-linearize APUFs is by explicitly introducing non-linearity without changing the underlying APUF model, by introducing several APUF chains and combining them with a XOR as shown in Fig. 1c. The relationship between challenge and response is described by:

$$r = \prod_{t=1}^{k} sign\left(\langle \boldsymbol{\omega}_t, \boldsymbol{\Phi} \rangle\right) \tag{7}$$

Each arbitrary APUF chain $\mathcal{P}_t \forall 1 \leq t \leq k$ has its own delay parameter set ω_t. A successful modelling attack needs to learn the non-linearity induced due to XORs in addition to of all the delays as shown in Eq. 7, which is a far more difficult problem. Machine learning based modelling attacks need to *de-linearize* Δ, as the decision boundary would no longer to polynomially separable.

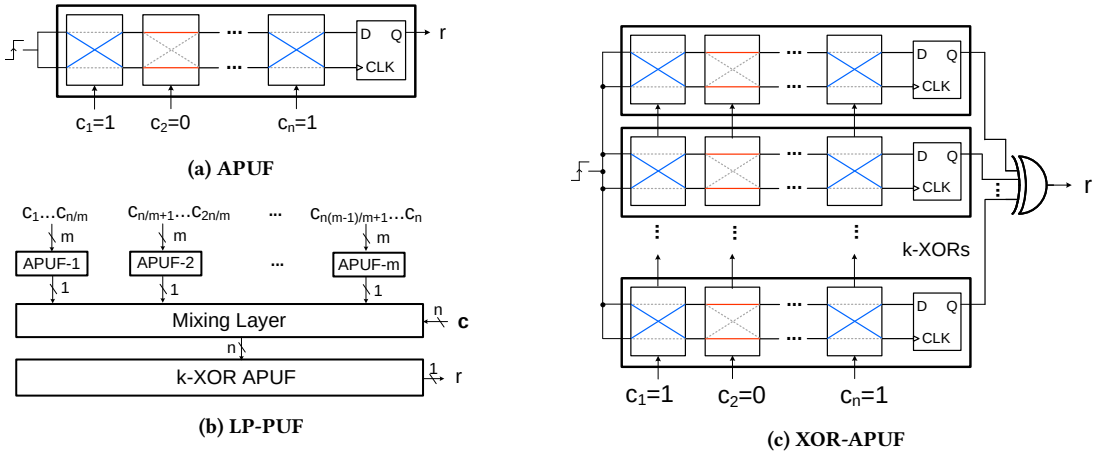

Figure 1: PUF Architectures for (a) APUF (b) LP-APUF (c) XOR-APUF

However, increasing number of APUFs in XOR-APUF negatively impacts the PUF reliability at the cost of high security. Other techniques to boost up the security of APUF include challenge obfuscation with random data such as in LP-PUFs.

2.2 LP-PUF

LP-PUF [34] takes advantage of Substitution-Permutation networks (SPNs), a technique used for the design of block ciphers. Formally, given a one-way SPN function $f_s(\mathbf{c})$ parameterized by some secret s, we get a scrambled output $\mathbf{c}_s$ which is provided as an input to the k-XOR PUF. LP-PUF generates responses as:

$$r = \prod_{t=1}^{k} sign\left(\langle \boldsymbol{\omega}_t, \boldsymbol{\Phi}_s \rangle\right) \tag{8}$$

where $\boldsymbol{\Phi}_s$ is the parity vector of the scrambled challenge.

The architecture of LP-PUF consists of three layers, namely an Arbiter layer, a mixing layer and an XOR layer (see Fig. 1b). The APUFs in the first stage are of length n/m which generate m random bits. The hardware dependent random bits prevent any circular dependency problem during scrambling. In the layer 2 each bit of the input challenge $\mathbf{c}$ is XORed with a randomly selected set of $m/2$ APUF outputs out of the m available outputs of the first stage. This XOR operation at the Layer-2 encodes each of the individual challenge bits based on selection of random bits. The final output of LP-PUF is that of a k-XOR Arbiter PUF.

The security of the LP-PUF depends on the challenge obfuscation at the input of the Layer-3. Any modelling attack on LP-PUF thus has to learn both SPN and the non-linear XOR.

Evaluation Metrics: The commonly used metrics to evaluate PUFs are:

(1) *Uniformity:* It estimates the distribution of responses corresponding to a challenge set and is calculated with the Hamming Weight in the responses to 50% (ideal value).
(2) *Uniqueness:* This property describes the difference in responses obtained when an identical challenge set is given as input to a pair of PUF instances (ideally 50%).
(3) *Reliability:* A PUF is said to be reliable when the responses are reproducible for an identical challenge over time and operating conditions. The ideal value is 100%. However, a maximum error rate of 5% is tolerable and can be corrected using Error-Correcting Codes (ECC) [27].

2.3 Modelling and reliability attacks

Recent works on PUF modelling use machine learning techniques such as logistic regression [28] and neural networks [20, 29, 37]. ML attacks are black box attacks which try to find a relationship between the challenge response pairs. The attacks are performed by creating training and test datasets from the largely collected uniform random challenges and their corresponding responses. However, the success of the ML attack depends on nature of the hyperplane separating the +1/-1 responses. Logistic regression easily learns APUFs due to the presence of a linear hyperplane which separates the -1/1 responses. But, XOR-APUFs and LP-PUF de-linearize the challenge response pair relationship thereby, making it challenging for ML model to learn the PUF behaviour.

Furthermore, PUF reliability can be used to attack APUFs, XOR APUFs [2, 31] and Interpose-PUFs [31]. For an APUF, the response is termed unreliable, if the delay difference between two branches is less than a threshold. Evolution Strategies (ES) such as CMA-ES [2] find the best estimate of $\boldsymbol{\omega}$ by maximizing the correlation between the measured reliability h_i and hypothetical reliability $\widetilde{h_i}$ computed from a PUF model. A stronger version of the CMA-ES based reliability attack was proposed in [31] which used gradient decent based approach. Composite PUF such as IPUFs were attacked by this approach but, reliability based attacks haven't been demonstrated for complex PUFs such as LP-PUFs.

2.4 Evolutionary algorithms

Evolutionary algorithms (EA) are a class of algorithms that solve a variety of computational problems that have well defined *search space* [1, 22, 39]. Theoretically, assuming infinite computational resources, it is possible to search through the entire search space for optimal solution through either a brute force strategy or a randomized search strategy. However, evolutionary algorithms make

intelligent search choices that allow them to reach a solution without the need to search through the entire search space. To do so, each EA takes a well defined *fitness function* that formalizes how good the search is progressing, with respect to some global optimum. Each evolutionary algorithm handles the following two opposing forces in balance in order to streamline its search [33]:

- **Exploration**: This property defines how much search space has been explored by the EA. Too low exploration can cause the EA to fit into local optimum. Too large exploration is no different from a completely random search.
- **Exploitation**: This property defines convergence towards the global optimum. Too large exploitation risks the EA getting caught in global optimum. While too low exploitation is similar to a large exploration strategy, which is no different from a completely randomized search of the search space.

EAs are further classified based on how exactly they evolve. We mention two textbook algorithms derived from natural processes: ① genetic algorithms, derived from biology of gene evolution, and ② swarm algorithms, derived from social behaviour of swarms like ants, honey bees, birds, and so on.

2.4.1 Genetic Algorithms. Genetic algorithms are inspired from the evolution in genes. The core philosophy is to mimic nature's *survival of the fittest* strategy. Every genetic algorithm is a composition of the following four sub-parts [7]:

- **Genotype**: Genotype encapsulates the genetic encoding of the problem under observation.
- **Selection operator**: This operator chooses which members of the population shall reproduce in a certain generation.
- **Crossover operator**: This operator controls how to mix two genotypes of a population. This operator mainly controls the **exploitation** phase of the genetic algorithms.
- **Mutation operator**: This operator controls the random mutations that come into the population as it evolves. This operator controls the **exploration** phase of genetic algorithms.

Refer Fig. 2 for a pictorial depiction of these concepts. The δ values capture the exact genotype in the context of PUFs. The selection phase selects two random parents from the population, which (by action of the crossover operator) split their genotypes into two and swap them. Finally, the mutation operator adds a small noise to the genotype of one genotype.

2.4.2 Swarm Algorithms. Swarm algorithms [24] draw upon the efficiency of various available swarms in the animal kingdom that allow them to achieve an objective of collective interest. One such example of Particle Swarm Optimization (PSO), which is inspired by the behaviour of a swarm of birds foraging for food. PSO works upon a swarm of particles in a search space with a single food source. The **exploration** phase of the algorithm allows the particles to move randomly (or heuristically) in the search space. As soon as a path to the food source is expected to be discovered, the swarm/**exploitative** behaviour kicks in wherein other particles in the search space also try to move along the newly discovered path in order to get a better convergence to the global optimum.

3 EVOLUTIONARY SEARCH (ES) STRATEGY

In Sec. 1.4, we elaborated a new perspective of conceiving the problem of modelling PUFs by *searching* for an approximation to delay Δ instead of trying to learn the mapping $\mathcal{P} : \mathcal{U}_C \rightarrow \mathcal{U}_R$. Usually, in literature, *search* problems are attacked by evolutionary strategies. In this context, prior works [14, 32] have done some investigation into the problem of modelling PUFs by using evolutionary algorithms. However, their *search strategy* still focused on using evolutionary algorithms to *exlore* the search space of the mapping $\mathcal{P} : \mathcal{U}_C \rightarrow \mathcal{U}_R$. *To the best of our knowledge, no prior work has been done to investigate the applicability of evolutionary algorithms to search for an approximation to delay* Δ.

In this section, as a consequence, we first discuss why prior attempts of using textbook evolutionary algorithms in the context of PUFs have been unsuccessful. We note pitfalls of such algorithms, and use these insights to propose a variant of the Particle Swarm Optimization algorithm based on a PUF's internal architecture.

3.1 Failures of prior ES attacks on PUFs

The prior works on classical evolutionary algorithms for PUFs use genetic algorithm to attack a class of delay-based PUFs including APUF, XOR-APUF, feed forward APUFs and analog PUFs [14, 32]. The standard genetic algorithms (cf. Sec. 2.4) are a combination of three generic operators tuned to specific applications: selection, crossover, and mutation. It is derived from the idea of human evolution, coupled with Darwinian concept of *survival of the fittest*, ensuring that the human population gets fitter over sufficient number of generations. Specifically, given two DNA sequences, human evolution creates a new DNA sequence by intermixing parent genes as well as by occasional mutation. Genetic algorithm mathematically implements the evolution of the population parameter.

Application of a genetic algorithm in the case of PUFs requires the knowledge of the following:

- **Genotype representation**: PUF representation used by genetic algorithm.
- **Hyperparameters**: parameters for selection, crossover and mutation.

Previous works [14, 32] on standard genetic algorithms choose the hyperparameters based on the default settings of genetic algorithms. It is the genotype representation that needs deliberation from the perspective of a PUF. In [14, 32], this representation is a set of table entries used by the PUF. However, while [14] does not move beyond feed-forward APUF into XOR designs, [32] explicitly claims the failure of genetic algorithms in breaking PUFs.

In this work, we come up with a better genotype representation for PUFs. We have noted that a PUF's security is enhanced using *de-linearization* defences and input challenge transformations. However, even in the presence of such defences, the functionality of any delay-based PUF is determined by $\boldsymbol{\omega}$ vector which is used to evaluate Δ. To the best of our knowledge, this is the first work to directly estimate Δ, and therefore, model the PUF. Formally, our ES algorithm's genotype representation is a normally distributed delay parameter set $\boldsymbol{\delta} = \{\delta_1, \delta_2, \delta_3, \cdots, \delta_n\}$, and the objective of the search is to converge towards $\boldsymbol{\omega}$.

Why genetic algorithm is still not suitable? One could argue that the genetic algorithm strategies discussed in [32] could still

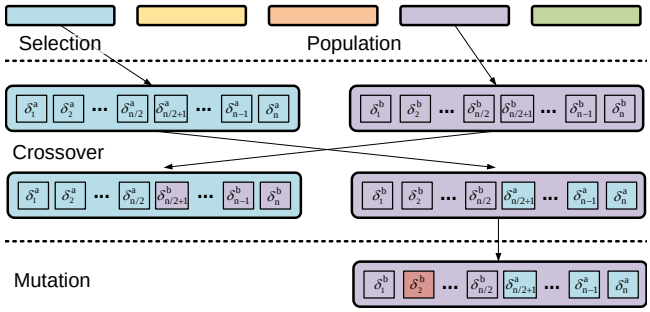

Figure 2: A sample iteration of the genetic algorithm on the genotype suggested in this work.

be potentially applicable with this new genotype representation $\boldsymbol{\delta}$. However, we show this is not the case. A standard genetic algorithm cannot be applied in the context of PUFs without verifying the correctness of a genotype crossover between parent PUFs producing a valid child PUF. Let us consider the case of a simple APUF, as all delay-based PUFs can be derived from APUF.

$$\Delta = \sum_{x=1}^{n} (\delta_{c_x}^{(x)} \prod_{y=x}^{n} c_y)$$

From observation **O2** (c.f. Sec. 1.4), we understand that although the individual stage delay set $\boldsymbol{\delta}$ can take different values, it is their *combined effect* Δ that has the final control over generated responses. It is this *combined effect* Δ that makes genetic algorithm difficult to apply in the context of PUFs, which we show next. Without loss of generality, let us consider one iteration of a standard genetic algorithm for one challenge-response pair with the following genetic algorithm's operator descriptions:

- **Optimization function**: Accuracy of PUF responses
- **Selection**: Roulette selection
- **Crossover**: Single-point crossover with rate 80% [32].
- **Mutation**: Real-valued normal mutation with mutation rate $\sqrt{\tau}$, where τ is the length of genotype representation [32].

Fig. 2 pictorially summarizes one iteration of the algorithm. Out of a population of several candidates, two candidates are chosen by the algorithm's selection operator. Let these candidates' genotype be described by the parameter set $\boldsymbol{\delta}^a = \{\delta_1^a, \delta_2^a, \cdots, \delta_n^a\}$ and $\boldsymbol{\delta}^b = \{\delta_1^b, \delta_2^b, \cdots, \delta_n^b\}$. Then, the crossover operator creates a central split and combines halves of $\boldsymbol{\delta}^a$ and $\boldsymbol{\delta}^b$ to create two children (say $\boldsymbol{\delta}^c$ and $\boldsymbol{\delta}^d$). Finally, the mutation operator randomly changes δ_2^b (the second element from $\boldsymbol{\delta}^b$). We now formalize this experimental setup and establish theoretical bounds on the probability of $\boldsymbol{\delta}^c$ and $\boldsymbol{\delta}^d$ being *fitter* than $\boldsymbol{\delta}^a$ and $\boldsymbol{\delta}^b$, thereby highlighting the shortcomings of standard genetic algorithm used for PUFs.

To do so, let us consider a probabilistic-polynomial time adversary $\mathcal{A}$ and a challenger $\mathcal{C}$. $\mathcal{C}$ encapsulates two oracles $\mathcal{O}_\mathcal{U}$ and $\mathcal{O}_\mathcal{P}$. Upon each query made by $\mathcal{A}$, $\mathcal{C}$ flips an unbiased coin, chooses one oracle, and returns its response to $\mathcal{A}$. This constitutes one iteration of the genetic algorithm. The oracles behave as follows after being queried by $\mathcal{A}$:

- $\mathcal{O}_\mathcal{U}$ returns a randomly sampled vector $\{\delta_1^U, \delta_2^U, ..., \delta_n^U\}$ where each of the variables are normally distributed with mean 0 and standard deviation 0.5.
- $\mathcal{O}_\mathcal{P}$ returns $\boldsymbol{\delta}^c$ without loss of generality. The analysis is unchanged even if $\boldsymbol{\delta}^d$ is returned in this stage.

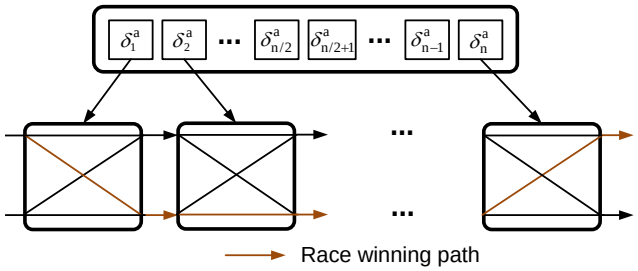

Figure 3: An example of conversion of genotype into PUF delays. Each δ_i^a affects the i-th stage. The combined effect Δ is obtained by the combined effects of each of δ_i which allow one path to win the race.

Under this model, we claim the following. Here, $\epsilon(n)$ is a negligible function.

Lemma 1 (Indistinguishability of combined effect Δ). *Let $\mathcal{E}_{\mathcal{O}_\mathcal{U}}$ denote the event where the adversary correctly guesses that $\mathcal{C}$ chooses $\mathcal{O}_\mathcal{U}$ to answer $\mathcal{A}$'s query. Likewise, let $\mathcal{E}_{\mathcal{O}_\mathcal{P}}$ be the the event where the adversary correctly guesses that $\mathcal{O}_\mathcal{P}$ was used. Finally, let $\mathbb{P}_\mathcal{A}$ be the probability that the adversary wins. Then, the following holds:*

$$\mathbb{P}_\mathcal{A}[\mathcal{E}_{\mathcal{O}_\mathcal{U}} = 1] = 1 - \mathbb{P}_\mathcal{A}[\mathcal{E}_{\mathcal{O}_\mathcal{P}} = 1] = \frac{1}{2} + \epsilon(n)$$

Proof of a "loose" bound on $\mathbb{P}_\mathcal{A}$. The delay Δ^a for the PUF parameterized by $\boldsymbol{\delta}^a$ is given by:

$$\Delta^a = \sum_{i=1}^{n} (\delta_i^a \Phi_i)$$

Each stage i is parameterized by delay value δ_i that chooses one of the two possible paths at that stage as shown in Fig. 3.

This leads us to the following observation: in order for $\boldsymbol{\delta}^c = \{\delta_1^a, \delta_2^a, ..., \delta_{\frac{n}{2}}^a, \delta_{\frac{n}{2}+1}^b, ..., \delta_n^b\}$ to be **at least** as fit (using accuracy of responses as the fitness function) as $\boldsymbol{\delta}^a$, the *behaviour* of each of $\{\delta_{\frac{n}{2}+1}^b, \delta_{\frac{n}{2}+2}^b, ..., \delta_n^b\}$ should be *comparable* to that of $\{\delta_{\frac{n}{2}+1}^a, \delta_{\frac{n}{2}+2}^a, ..., \delta_n^a\}$. Formally, for $\boldsymbol{\delta}^c$ to be **at least** as fit as $\boldsymbol{\delta}^a$, the *combined effect* Δ^c should be similar to the *combined effect* Δ^a, allowing $\boldsymbol{\delta}^c$ to correctly predict no less responses than what $\boldsymbol{\delta}^a$ does. Now we know that every element in the parameter set $\boldsymbol{\delta}^a$ translates to a either a cross or straight path (c.f. Fig. 3). For $\boldsymbol{\delta}^c$ to be **as least** as fit as $\boldsymbol{\delta}^a$, the $\{\delta_{\frac{n}{2}+1}^b, \delta_{\frac{n}{2}+2}^b, ..., \delta_n^b\}$ parameters should allow the same cross/straight path wins which $\{\delta_{\frac{n}{2}+1}^a, \delta_{\frac{n}{2}+2}^a, ..., \delta_n^a\}$ allowed, in order for $\boldsymbol{\delta}^c$ to win races *exactly* or better than $\boldsymbol{\delta}^a$.

However, there is a catch here. The evolutionary algorithm learnt values $\{\delta_{\frac{n}{2}+1}^b, ..., \delta_n^b\}$ in the genotype $\boldsymbol{\delta}^b$, implies $\boldsymbol{\delta}^c = \{\delta_1^a, \delta_2^a, ..., \delta_{\frac{n}{2}}^a, \delta_{\frac{n}{2}+1}^b, ..., \delta_n^b\}$ is a *fit combination* if and only if the first $\frac{n}{2}$ positions of $\boldsymbol{\delta}^c$ allow the same cross/straight path wins as $\{\delta_1^b, \delta_2^b, ..., \delta_{\frac{n}{2}}^b\}$. Without any further assumption on the algorithm, an arbitrary stage δ_i^a for $1 \leq i \leq \frac{n}{2}$ has probability $\frac{1}{2}$ of performing the same *behaviour* (i.e. allowing either the cross or the straight path to win) as δ_i^b did. Thus, the probability of $\boldsymbol{\delta}^c$ being **at least** as fit as $\boldsymbol{\delta}^a$ is $(\frac{1}{2})^{\frac{n}{2}} \leq \epsilon(n)$, which is a negligible function in n (the number of stages in the PUF chain). Note that this is indistinguishable from a randomly

sampled PUF $\{\delta_1^a, \delta_2^a, \delta_3^a, ..., \delta_{\frac{n}{2}}^a, \delta_{\frac{n}{2}+1}^U, \delta_{\frac{n}{2}+2}^U, ..., \delta_n^U\}$, where each of δ_i^U for $\frac{n}{2}+1 \leq i \leq n$ is a randomly sampled parameter and has a **loose** upper bound of $\frac{1}{2}$ of matching the same *behaviour* (i.e. letting either the criss-cross or the straight path winning) as δ_i^a, amounting the total probability to $(\frac{1}{2})^{\frac{n}{2}}$.

The main takeaway is that a genetic algorithm's *combination* of two PUF parameters to create new PUF parameters has a negligible ($< \epsilon(n)$) probability of being fitter than the previous population. In light of this, we need to devise a better evolutionary strategy to this end. We take inspiration from the generic design of particle swarm optimizations (briefed in Sec. 2.4) that we discuss next.

4 AMOEBIC SWARM OPTIMIZATION

In this section, we introduce a novel Particle Swarm Optimization algorithm inspired by the biomimicry of amoebic reproduction. Any evolutionary search strategy needs to delicately balance between two equally important yet competing strategies: ① exploration, and ② exploitation. Exploration controls how much the algorithm *explores* for new solutions in the search space. Exploitation, on the other hand, focuses on developing on previously found solutions in order to make them even better. Too much of either is disastrous for the convergence of the algorithm. Overdoing exploration shall be no better than a blind random search. And overdoing exploitation risks being caught in local extremums.

4.1 Algorithm design intuition

Let us consider the PUF's parameters $\boldsymbol{\delta}^a = \{\delta_1^a, \delta_2^a, \delta_3^a, \cdots, \delta_n^a\}$ learned over a period of several generations of the evolutionary learning algorithm. However, unlike the changes a genetic algorithm does, we do not want to throw away half of the parameters learnt and replace them with parameters of another PUF (which is a bad design decision since no two PUFs are comparable). Rather, we want to perturb a small set $S \subseteq \boldsymbol{\delta}^a$ *in-place*. Based on the idea of race between cross/straight paths (c.f. Fig. 3), this perturbation may cause the *behaviour* of few stages of the PUF to change. For example, perturbing δ_i^a to $(\delta')_i^a$ (for a specific i where $1 \leq i \leq n$) may cause the originally winning criss-cross path to now lose the race to straight path, leading to a change in performance/accuracy on target response set. The new PUF genotype shall then be $(\boldsymbol{\delta}')^a = \{\delta_1^a, \delta_2^a, \delta_3^a, \cdots, (\delta')_i^a \cdots, \delta_n^a\}$. If $(\boldsymbol{\delta}')^a$ improves upon the performance of $\boldsymbol{\delta}^a$, then all future generations will now build upon $(\boldsymbol{\delta}')^a$ instead of $\boldsymbol{\delta}^a$ to improve even further.

The major difference between this approach and the genetic algorithm approach is that by choosing to update δ_i^a and evaluating the correctness of the newly generated PUF, the algorithm allows $(\delta')_i^a$ to be affected by all other (already present) parameters, while also focusing on improving overall performance/accuracy on target response set. This allows the algorithm to evaluate the effect on $\Delta = \sum_{i=1}^{n} (\delta_i^a \Phi_i)$ by changing δ_i^a to $(\delta')_i^a$ and preserving all other parameters, thereby allowing a mode of convergence towards the optimal solution. Over time, the algorithm converges to approximate the correct *behaviour* of each stage (i.e. winning either the cross or the straight race) that is reminiscent of the stage-wise *behaviour* of the target PUF.

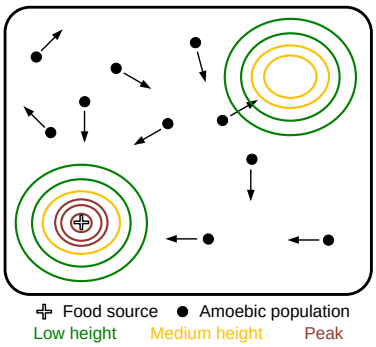

Figure 4: Initial amoebic population with a food source in the landscape defined by contours. It is assumed that the food source is at the highest peak in the landscape. Arrows indicate the directions in which the population advances.

4.2 Algorithm design decisions

In this work, we develop a variant of the generic Particle Swarm Optimization procedure named **Amoebic Swarm Optimization** or `CalyPSO`. The motivation of the algorithm lies in the following problem in nature: *how does a population of amoeba move towards a food source (i.e. an objective)?* Consider Fig. 4. There is a landscape with hills and valleys of varying heights. The objective (i.e. the food source) is the highest peak of the landscape. Initially, we have a population of amoebas randomly scattered in the landscape. Based on the *fitness* of each member of the population (i.e. the remaining distance from the food source), each member of the population takes one step towards the direction which takes that member closer to the food source. Intuitively, the higher the peak is in the landscape, the more *fit* an amoeba becomes when it reaches there.

However, each member of the population does not have the complete view of the landscape. Hence, every step an amoeba takes is according to the *local* best decision it can make. Hence, we have our first challenge **C1** that the algorithm needs to solve:

- **C1**. Ensure the amoebic population escapes local extremums, over a sufficient iterations of the algorithm.

Secondly, a generic Particle Swarm Optimization would involve swarm behaviour, in which a single amoeba, as soon as it finds a new optimal path to the food source, will broadcast this information to other members of the population. Henceforth, other population members can use the findings of one member to their advantage. However, as discussed in Sec. 3.1, in the context of PUFs, this broadcast is not of much use. Every member of the population will find its own path to the global optimum. Mixing different solutions in the context of PUFs is likely to be no better than a random search. Thus, we have a second challenge:

- **C2**. According to Lemma 1, mixing states of two PUFs into one solution is no better than a random guess. Thereby, a generic particle swarm optimization procedure is not much helpful in this context, since we cannot use one PUF instance to evolve another PUF instance.

4.2.1 Solving C1: Landscape evolution. We note that the generic landscape depicted in Fig. 4 is constructed by the specific data input to the algorithm, which is inspired by the particular problem being

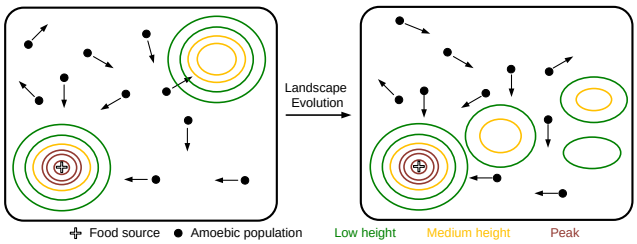

Figure 5: An example of landscape evolution. Note how the contours change as the landscape evolves.

solved. In our context, this landscape is the target PUF mapping $\mathcal{P}$ (c.f. Sec. 3.1). A landscape described by a huge amount of data extracted from $\mathcal{P}$ is bound to create a smooth landscape wherein the particles can converge upon the final objective. However, in practice, the amount of data available to the algorithm is limited.

Solution to C1. Given limited data from $\mathcal{P}$, construct partial landscapes from subsets of the overall data. Over a sufficient number of generations, the *false* optimums will smooth out, leaving the global optimum visible for convergence. Consider the illustration in Fig. 5. Because of using a subset of total data available, *false* contours may spring up. However, when the landscape is evolved over the course of the algorithm, these *false* contours can not occur in every sampled subset. Thereby, over sufficient runs of the algorithm, the members of the population stuck in *false* contours will also start converging towards the global optimum. Thereby, using landscape evolution as an essential portion of the algorithm allows it to prevent from being caught up in local extremums.

4.2.2 Solving C2: Mixing amoebic reproduction with Particle Swarm Optimization procedure. In a generic (PSO), the following two orthogonal forces balance out the convergence:

- **Search space exploration**: This is captured by the *particle* behaviour of the PSO. Given a member of the population, the algorithm will attempt to move it in a random direction and check how close the member moved towards the global extremum.
- **Search space exploitation**: Once a path to the global extremum is found, the *swarm* behaviour kicks in. Every member then follows closely the discovered optimal path to the global extremum.

However, as challenge **C2** points out, we cannot utilize the *swarm* behaviour of a generic PSO because that would require mixing genotypes from two members of the population (which, according to Lemma 1), is no better than a random genotype sample. This means that *search space exploration* is no longer possible: every member of the population will simply keep on doing a random search in their own specific directions. To mitigate this, we merge the generic concept of a PSO with *amoebic reproduction*. It is worth mentioning that, to the best of our knowledge, merger of an amoebic reproductive step to the evolution of a swarm in PSO has not been attempted before in literature.

Solution to C2: Since the generic swarm behaviour of a PSO is not relevant in the context of PUFs, in order to ensure sufficient exploration of the search space, we merge the concept of *amoebic reproduction* to PSO. We get two advantages from this design. ① *Amoebic reproduction*, being asexual, prevents the need to merge two PUF solutions into one (as genetic algorithm does), thereby

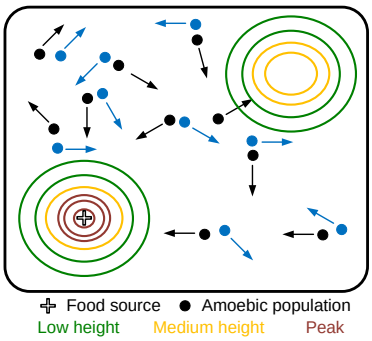

Figure 6: An example of amoebic reproduction applied in the algorithm.

avoiding the pitfalls that genetic algorithm has in the context of PUFs (c.f. Sec. 3.1). And, ② it is able to reproduce progressively fitter amoebas because the parents themselves are getting fitter by each generation. Point ② is in stark contrast with a genetic algorithm's reproduction step, which has no control on where in the landscape the populated children with spawn (because the children are produced by intermixing two parent genotypes), thereby risking bad solutions in the search process. Consider Fig. 6. Every iteration of the algorithm, in addition to moving in the locally optimal direction, also (asexually) reproduces to generate a progeny population. This population inherits the same representation as the parent, but takes its own path across the landscape.

4.3 Algorithm description

We summarize `CalyPSO` in Algo. 1. The algorithm assumes challenge-response tuple (C, R) and `target_puf_arch` as input. The parameter `target_puf_arch` abstracts the details of the architecture of the target PUF. Likewise, the tuple (C, R) is fetched from the target PUF, and used to evaluate fitness function. Intuitively, *fitness* quantifies how *close* a member of the population is to the target PUF. The algorithm then initializes a population of size `population_size` in the function `INITIALIZE_POPULATION`. Each member is of the same architecture as `target_puf_arch` and has randomly sampled delay parameter set δ.

Post that, the algorithm enters into an infinite loop of a set of sequential processes that allows convergence. Firstly, fitness is computed for each member of the population in function `COMPUTE_POPULATION_FITNESS`. The fitness function used in our algorithm is $fitness = acc - bias$. Here, accuracy acc abstracts the number of responses being correctly predicted. While the bias abstracts the property of a PUF to probabilistically generate one kind of responses *more* than others. For instance, a bias value close to 1 means the PUF is highly likely to generate more number of 1. Similarly, a bias value close to -1 means the PUF is highly likely to generate more number of -1.

The function `STEP` abstracts the method with which the population evolves over time and closely follows the approach detailed in Sec. 4 (refer to Sec. 7 for exact implementation specifics). Likewise, `AMOEBIC_REPRODUCTION` implements the solution to challenge **C2** by spawning amoebic children, who carry out the search in a completely independent manner than their respective parents. Moreover, `le_parameter` controls the frequency of landscape evolution

(solution to challenge **C1**) by resetting the training data (C, R) every few generations. Finally, `sort()` and `trim()` functions ensure survival of the fittest members of the population, while keeping the population size capped at `population_size`.

We note that the algorithm described so far tries to model a target PUF using *all* its delay parameters. Hence, as the complexity of the target PUF increases (for example, the number of Arbiter chains in the `XOR` PUF increases), the algorithm takes longer to converge. As a consequence, in the next section, we introduce novel attack strategies that allow modelling a higher order PUF using a lower order and simpler variant, allowing for faster convergence as well as interesting theoretical results regarding the security of PUFs.

Algorithm 1 `CalyPSO`: Amoebic Swarm Optimization algorithm

```
procedure INITIALIZE_POPULATION(target_puf_arch)
    set population_size ← 500
    set population_list ← NULL
    while population_list.size() != population_size do
        /* Randomly generate an instance of target_puf_arch*/
        population_list.append(RANDOM(target_puf_arch))
    return population_list
procedure COMPUTE_POPULATION_FITNESS(C, R, population_list)
    for member in population_list do
        predicted_responses = member.evaluate(C)
        member.fitness = /*compare similarity between R and predicted_responses*/
    return
procedure AMOEBIC_REPRODUCTION(population_list)
    population_list.append(CLONE(population_list))
procedure STEP(population_list, delay_param)
    for member in population_list do
        /*Add normal noise to each member's delay parameters contained in the list delay_param*/
procedure ATTACK_WRAPPER
    set GENERATION ← 0
    (C, R) ← challenge-response tuple of the target PUF
    set target_puf_arch
    le_parameter ← landscape evolution hyperparameter
    set delay_param ← 1          ▷ delays to perturb in one generation.
    population_list ← initialize_population(target_puf_arch)
    while True do
        call compute_population_fitness(C, R, population_list)
        population_list.sort()                    ▷ Sort based on fitness
        population_list.trim()                    ▷ Trim population to size 500
        call amoebic_reproduction(population_list)
        call step(population_list, delay_param)
        if not GENERATION % le_parameter then
            /*randomly sample a new challlenge set C'*/
            R ← target PUF response on C'
```

5 DOWNGRADE ATTACK: REDUCING ORIGINAL SECURITY GUARANTEES

So far, the algorithm `CalyPSO`, detailed in Sec. 4, allows *exact* replication of a target PUF architecture. For instance, given k-XOR PUF with challenge length n as the target, the algorithm allows learning a suitable delay parameter set of size $n \times k$ that approximates Δ. However, the perspective of viewing the problem of modelling PUFs as a search problem allows another kind of cross-architecture modelling attacks using the algorithm described in Sec. 4. We detail two such attacks here: ① attacks allowing bypassing the transformations on input challenge set (as in LP-PUF), and ② allowing a cross-architecture learnability attack on k-XOR PUFs.

5.1 Degrees of freedom reduction attack

We propose a *degree of freedom* reduction attack wherein we reduce the security of a k-XOR PUF to a $(k-1)$ XOR PUF using the idea of `FORMULA-SATISFIABILITY` of Boolean functions. XOR PUFs achieves de-linearization by increasing the number of XORs thereby, increasing the non-linearity in modern PUF designs (c.f. Sec 1.3). In order to reduce degrees of freedom, we model the problem of learning a k-XOR PUF as a `FORMULA-SATISFIABILITY` problem. Referring to Fig. 1c, a k-XOR PUF can be algebraically modelled by Eq. 7 as a product of the sign function of k Arbiter PUFs. At the hardware level k APUF outputs (0,1) are XORed, which can be represented by the function:

$$R = f_1(\mathbf{C}, \boldsymbol{\delta}_1) \oplus f_2(\mathbf{C}, \boldsymbol{\delta}_2) \oplus \cdots \oplus f_k(\mathbf{C}, \boldsymbol{\delta}_k) \tag{9}$$

where $\mathbf{C}$ is the input challenge and individual arbiter chains are represented by $\boldsymbol{\delta}_1, \boldsymbol{\delta}_2, \ldots, \boldsymbol{\delta}_k$ delay vectors and f_i functions generating the response R.

Eqn. 9 comprises of a commutative/associative operation with k variables, each of which can be represented by a Boolean function implemented using AND and OR gates. In the case of PUFs, since the adversary has access to R, this equation in k variables has actually just $k-1$ degrees of freedom. Formally, eqn. 9 can be re-written as:

$$R = f_1(\mathbf{C}, \boldsymbol{\delta}_1) \oplus f_2(\mathbf{C}, \boldsymbol{\delta}_2) \oplus \ldots f_{k-1}(\mathbf{C}, \boldsymbol{\delta}_{k-1}) \oplus b \tag{10}$$

where the final bit b is a deterministic constant $\in \{0, 1\}$ that can be evaluated from the other variables. Therefore, given a response set R, Eq. 10 reduces the effect of the last Arbiter chain to a deterministic bit. From the point of an adversary, it no longer needs to learn the behaviour of the last Arbiter chain. Hence, we draw the following observation:

✓ **O3**. Any k-XOR PUF fulfils the `FORMULA-SATISFIABILITY` equation in k variables and $k-1$ degrees of freedom. In the context of our algorithm, an adversary only needs to learn $k-1$ arbiter chains. The contribution of the final k-th arbiter change can be evaluated from the final response, thereby allowing us to reduce the security of a k-XOR PUF to a $(k-1)$-XOR PUF.

5.2 Input transformation bypass attack

By viewing modelling of PUFs as not learning the challenge-response mapping, but rather approximating Δ, we allow ourselves an important freedom, which becomes the basis of this bypass attack. We exemplify this by exploiting LP-PUF's input transformation that depends upon substitution-permutation networks (SPN) like diffusion. Recalling our discussion in Sec. 2.2, LP-PUF is delay-based PUF that relies upon a SPN to transform the input challenge set C to C', before providing C' to a standard k-`XOR` PUF. The SPN is parameterized by secret bits, which are again generated by a series of k `APUF`s, thereby tying the SPN's security to the hardware itself. Because of this, to the best of our knowledge, there has been no machine learning attack or evolutionary algorithm based attack on LP-PUF because mapping C directly to the response set R is equivalent to learning the SPN without knowledge of C', which

is a difficult problem. However, the algorithm reported in Sec. 4 is able to attack LP-PUF's SPN because it considers the SPN's structure also as a part of its genotype, thereby allowing the swarm to converge upon a solution that models both ① SPN, as well as ② k-XOR PUF components of LP-PUF.

However, we reason out that it is possible to attack PUFs involving input transformations from another perspective, using our algorithm. We give an intuitive understanding here. We again build upon the same ideas of *combined effect* Δ introduced in earlier sections, albeit in a slightly different way. We make the following observation:

✓ **O4**. Any input transformation based PUF (like LP-PUF) places a layer of input transformation before the actual PUF (like a k-XOR PUF). This implies the PUF functions not on the actual input (C, R) but rather on the transformed tuple $(C^{'}, R)$. The **Input transformation bypass attack** aims not to learn both ① $C \rightarrow C^{'}$ mapping as well as ② the k-XOR mapping $C^{'} \rightarrow R$, but rather randomly sample the input transformation function (i.e. derive a $C \rightarrow C^{''}$) and learn a k-XOR mapping $C^{''} \rightarrow R$.

In other words, instead of asking the algorithm in Sec. 4 to learn the mapping and then the PUF itself, we let the spawned members of the population implement their own *unlearned* transformation (i.e. $C \rightarrow C^{''}$) and then learn *some* other k-XOR mapping from $C^{''}$ to R. Note that, given a random $C^{''}$, it is not always possible to find such a mapping, our empirical results show that for sufficient size of input data, the probability of this event is negligible. Therefore, our algorithm (which launches an exploration in the search space of all PUFs of a given architecture) is able to find *some* PUF which maps $C^{''}$ to R, and by extension *models* the target PUF.

6 OPTIMIZATION BY MAJORITY VOTING

Any evolutionary algorithm tries to evolve its population in a way that makes the population *fitter* over time. Often, this results in longer run-times than usual machine learning attacks. In light of the same, the following observation can help.

✓ **O5**. So far, `CalyPSO` uses the *fittest* member of the entire population in order to report accuracy. However, the algorithm exhibits *swarm* behaviour, which means all members of the population are moving towards the final objective in their own trajectory. As such, instead of throwing away the *learning* done by all members in the population and considering only the fittest, a *majority vote* based prediction strategy gives better empirical performance and faster convergence.

Here, it is sufficient to show: ① the majority voting over the entire population is expected to perform better than the *fittest* member, and ② the probabilistic bound of a wrong majority vote is negligible for a sufficient size of the population. To do so, we consider a population of size ζ. For a single challenge-response pair (c, r), we can, therefore, define a set of identically and independently distributed (i.i.d) random variables $\{\mathcal{Z}_1, \mathcal{Z}_2, \mathcal{Z}_3, ..., \mathcal{Z}_\zeta\}$ such that for any arbitrary $\mathcal{Z}_i$, the following holds:

- $\mathcal{Z}_i = 1$ iff the i-th member of the population correctly predicts response r for challenge c.
- $\mathcal{Z}_i = 0$ otherwise.

Assume that the average fitness of the population is f_{avg}. We now define another random variable that captures the combined action of the entire population.

$$\mathcal{Z} = \mathcal{Z}_1 + \mathcal{Z}_2 + ... + \mathcal{Z}_\zeta$$

By linearity of expectation, we can express the overall expected majority vote as $E[\mathcal{Z}] = \sum_{x=1}^{\zeta} E[\mathcal{Z}_i] = \zeta \times f_{avg}$. On similar lines of argument, the majority vote $\mathcal{Z}$ fails if less than $\frac{\zeta}{2}$ members of the population vote in favour of the correct response bit. Combining both $E[\mathcal{Z}]$ and $\mathcal{Z}$ gives us the bound for amplification of success in case of majority voting [11]:

$$E[\mathcal{Z}] - \mathcal{Z} \geq \zeta \times (f_{avg} - \frac{1}{2}) \implies E[\mathcal{Z}] - \mathcal{Z} \geq 0 \qquad (11)$$

This result clearly achieves objective ①: as the population gets fitter (i.e. f_{avg} increases), majority voting is expected to perform better than the individual members of the population. Moreover, choosing a high ζ, although causes longer run-time, also allows for a larger accuracy amplification when using majority vote to predict a PUF's response. Furthermore, to achieve objective ②, we upper bound the probability of *failure* of majority voting by invoking Chernoff-Hoeffding theorem [11]:

$$Pr[\mathcal{Z} \leq \frac{\zeta}{2}] \leq e^{-2((f_{avg}-0.5)\times\sqrt{\zeta})^2} \qquad (12)$$

which is a negligible function in the size of the population as well as the overall fitness. More precisely, the probability that a majority vote over a large population will fail is negligible, as the population gets fitter over time.

7 EXPERIMENTAL RESULTS AND ANALYSIS

In this section, we present the experimental details and results.

Experimental Setup and Hyperparameter tuning. Our experiments include simulations on *PyPUF* [36] as well as validation on actual hardware data. As the procedure in `CalyPSO` (Algo. 1) depicts, we start by providing a challenge-response set (which may originate from either PyPUF's challenge generator or may come from actual hardware runs) as input. We divide this set into training and validation sets, with the latter never being used in `COMPUTE_POPULATION_FITNESS` at any point in the run of the algorithm.

The next important function in the algorithm is the `STEP` function, which is responsible to move the population towards the global optimum. We implement the `STEP` function by a *round-robin updation scheme*. For instance, if `target_puf_arch` is a 4 XOR PUF and the size of challenges is 64 bits, then we have 256 learnable delay parameters $\delta = \{\delta_1, \delta_2, \delta_3, \delta_4, ..., \delta_{256}\}$. For one `GENERATION`, `STEP` will randomly pop one parameter δ_i from this list, and add a normal noise $\mathcal{N}(0, \frac{1}{4})$. Further generations will repeat this process, but on the parameter set $\delta^{'} = \{\delta_1, \delta_2, \delta_3, \delta_4, ..., \delta_{i-1}, \delta_{i+1}..., \delta_{256}\}$. This strategy allows all delay parameters of δ to get an equal chance at evolution. Next important hyper-parameter is `le_parameter` that controls landscape evolution (c.f. Sec 4.2.1). Too low value of `le_parameter` will change (C, R) too fast for the algorithm to learn anything useful. And too high a value of `le_parameter` risks the population getting caught in a local optimum. Through empirical

Table 1: Experimental results for different PUF architectures from simulations on *PyPUF*. Here $K = 10^3$ and $M = 10^6$. The table captures three different independent experiments.

PUF arch	Train CRPs	Time taken Time taken	Run 1 Acc.	#generations (Run 1)	Run 2 Acc.	#generations (Run 2)	Run 3 Acc.	#generations (Run 3)
APUF	5K	~ 15 min.	99.13 %	363	98.86 %	462	99.36 %	638
2 XOR	30K	~ 1 hour	97.81 %	2043	97.13 %	2714	98.05 %	1953
3 XOR	30K	~ 3 hours	98.38 %	8953	97.42 %	9373	96.63 %	8737
4 XOR	100K	~ 8 hours	94.50 %	17213	94.62 %	17742	96.37 %	18253
5 XOR	100K	~ 8 hours	98.21 %	16843	99.42 %	18935	95.83 %	14948
6 XOR	300K	~ 17 hours	74.38 %	3736	71.52 %	4828	76.42 %	5836
8 XOR	200K	~ 1.5 days	92.61 %	24253	89.63 %	27284	94.61 %	28352
10 XOR	2M	~ 2 days	76.22 %	2575	68.35 %	4156	71.25 %	5831
13 XOR	2M	~ 2 days	78.10 %	2118	72.52 %	2418	75.35 %	2527
15 XOR	2M	~ 2 days	77.13 %	1898	75.12 %	2073	74.72 %	2462
20 XOR	2M	~ 2 days	78.80 %	1548	79.71 %	1824	80.42 %	1743
3-3 IPUF	500K	~ 4 hours	94.27 %	10057	91.42 %	9734	97.21 %	13152
1 LP-PUF	50K	~ 6 hours	97.42 %	9935	96.62 %	12157	98.24 %	9731
2 LP-PUF	100K	~ 5 hours	74.36 %	3842	78.52 %	3616	76.53 %	5623

Table 2: Results of the downgrade attack. A $x \rightarrow y$ entry means the results for an experiment where a PUF architecture x was modelled with architecture y (downgrade attack). Here, $K = 10^3$ and $M = 10^6$.

PUF arch	Train CRPs	Time taken Time taken	Run 1 Acc.	#generations (Run 1)	Run 2 Acc.	#generations (Run 2)	Run 3 Acc.	#generations (Run 3)
2 XOR $\rightarrow$ 1 XOR	50K	~ 6 hours	78.42 %	7630	73.25 %	7261	75.45 %	7541
1 LP-PUF $\rightarrow$ APUF	50K	~ 3 hours	75.97 %	8701	69.47 %	8159	73.72 %	8562
2 LP-PUF $\rightarrow$ 2 XOR	100K	~ 9 hours	82.74 %	2767	81.35 %	2525	84.25 %	2953
4 LP-PUF $\rightarrow$ 4 XOR	2M	~ 3 days	66.03 %	29284	59.21 %	27361	63.74 %	31527

evidence, we placed the value of this parameter's value at 50 generations. That is, if the algorithm fails to find a new solution for 50 consecutive generations, we invoke landscape evolution and give new paths to the global optimum by changing (C, R).

Simulations on PyPUF. For simulation of underlying PUFs, we used *PyPUF* [36] (which is a python based package for analysis of PUFs). Our implementation is thus written to be compatible with Python3. Each experiment was spread across 4 physical cores through Python's *multiprocessing.Pool*. Rest of the implementation has no dependence on any high-level evolutionary algorithm Python3 package. All our experiments were conducted on Intel(R) Xeon(R) Gold 6226 CPU @ 2.70 GHz with 96 cores, 2 threads per core, 12 cores per socket and 256*GB* DRAM.

Tab. 1 summarizes our results on *PyPUF* simulations of actual PUF architectures, while Tab. 2 summarizes the results of downgrade attacks. Each experiment's accuracy is reported on a test set of 1 million challenge-response pairs from the target PUF. Note that this test set is newly sampled every time `CalyPSO` finds a new fittest member in the population; the table captures the latest captured accuracy. This allows a better evaluation of `CalyPSO`'s convergence since a newly sampled test set prevents an overly optimistic view of `CalyPSO`'s ability because of a fixed test set.

One point to note is that in all cases, the simulations were noiseless. This means all PUF instances created by *PyPUF* had 100% reliability and 50% uniqueness/uniformity. This is because we wanted to evaluate `CalyPSO`'s ability without having any aid from external sources. For example, in the case of noisy simulations, *PyPUF* might generate target PUF instances having a significant deviation from 50% uniformity, making it easier to model

Validations on hardware data. . We also performed validations of our approach on hardware data, collected from both in-house constructions of PUFs. In addition to that, our in-house PUF constructions involved recreating a 4 XOR PUF in four instances on four different Nexys 4 DDR boards. The challenges were randomly sampled from within *PyPUF* and fed to each Nexys board in turn. This process was repeated 15 times, and the final response set was generated from the temporal majority voting over all measurements. The metrics of the created hardware are summarized in Tab. 3. As evident, all Nexys instances are behaving like PUFs. Our validations on challenge-response data from these hardware boards as well as with publicly available PUF datasets [19] correlate with the successful convergence of our algorithm (c.f. Tab. 1).

On average, we observed accuracy as high as 97.43% in, on average, 3000 generations. Note that this convergence is much faster than the convergence of 4-`XOR` PUF sampled from *PyPUF*. This is expected. All *PyPUF* simulations are overly idealistic. For noiseless simulations, the reliability value reported by *PyPUF* is 100 %, while uniqueness and uniformity are almost 50%. We also validated `CalyPSO`'s convergence on publicly available hardware datasets [19, 26]. The results for the same are summarized in Tab. 4.

Table 3: Performance metrics of in-house generated 4-XOR PUF hardware used to validate practicality of our algorithm's convergence. The number of CRPs is same as considered in Tab. 1.

Instance	Uniformity	Uniqueness	Reliability	Accuracy
1	50.332 %	50.13 %	89.15 %	93.13 %
2	49.798 %		87.79 %	95.12 %
3	50.814 %		89.43 %	92.35 %
4	49.946 %		88.90 %	90.13 %

Table 4: Performance metrics of publicly available hardware data from [19, 26].

PUF arch	Accuracy	#generations	Time taken
1-XOR	96.13 %	215	~ 24 mins
2-XOR	93.63 %	1475	~ 42 mins
3-XOR	94.13 %	4739	~ 1 hour
4-XOR	93.85 %	15396	~ 4 hours
5-XOR	89.41 %	14262	~ 6 hours

8 COMPARISON WITH STATE-OF-THE-ART ATTACKS

In Tab. 5, we compare the PUF modelling accuracy and required number of training challenge-response pairs (CRPs) for our proposed attack with existing neural networks (NNs) and reliability based attacks. It can be seen that our amoebic swarm optimization attack strategy requires a lower number of CRPs to perform a successful attack in contrast to reliability based attack requiring multiple measurements and NN attack striving to learn all the PUF representational parameters. Therefore, our proposed approach can be applied for approximating the delay parameters even on higher complexity XOR-PUFs with $k > 11$, which hasn't been demonstrated before in the literature. Furthermore, we also successfully attack (3-3) I-PUFs despite its increased non-linearity with respect to (1-4) I-PUF which have been demonstrated to break using reliability attacks [2, 31]. One must note that we do not use the PUF reliability information in our attacks and yet achieve a high modelling accuracy.

Lastly, in regard to the much coveted LP-PUF [2] construction that claims to have high security [34], we see that our proposed downgrade attack strategy obtains an accuracy of 75.97%, 82.74% and 66.03% for 1, 2 and 4 LP-PUF construction respectively. This is due to the fact that our attack strategy successfully nullifies the impact of input transformation in the case of LP-PUFs and thereby achieves better than random prediction for LP-PUFs.

9 IMPLICATION OF THE RESULTS AND FUTURE OF PUFS

The analysis in the paper shows very powerful tools which can annihilate the present day delay based PUFs by modeling the challenge response behaviors of the known designs. The work also brings in evolutionary algorithms as a powerful attack vector, at par and in fact much stronger than conventional machine learning based attacks. The fact that 20-XOR PUFs and the recently proposed LP-PUFs can be modeled using these methods raise the question: *How should we design the next generation delay PUFs?* A prudent approach would be to study the information acquisition done in the evolutionary learning process and develop PUF instances which ensure that the fitness of the evolution does not grow beyond a threshold [8]. The work also motivates and in fact compels to search for compositions of PUFs which are not closed in its set. This would imply that when two PUFs are composed together the resultant PUF would never realize Boolean mappings which would belong to the set of Boolean functions realized by the individual PUFs. Such kinds of questions have been addressed in the literature of block ciphers [5, 23], while defining compositions, we need to readdress them in the literature of PUFs intended to be resilient against modeling. We leave that as exciting future directions of research.

[2]It is to be noted that LP-PUFs are secure against the recently proposed cryptanalytic attack strategies [13] due to the hardware derived randomness induced in the challenge transformation). Therefore, such techniques do not apply to LP-PUF directly.

Table 5: Comparison Table for Modelling Accuracy across several PUF designs

Attack Type		Accuracy (%)			CRPs (×1000)		
		NN	Reliability	Proposed	NN	Reliability	Proposed
		[34, 37]	[2, 31]		[34, 37]	[2, 31]	
APUF		99.9	98.3	99.36	18.0.5	160	5
XOR-APUF	k=4	97	90	96.37	150	120	100
	k=5	97.3	-	99.42	200	-	100
	k=6	97.5	85	76.42	2000	240	300
	k=8	95.5	82	94.61	600	360	200
	k=10	97.9	78	76.22	119000	1200	2000
	k=13	-	-	78.10	-	-	2000
	k=15	-	-	77.13	-	-	2000
	k=20	-	-	80.42	-	-	2000
I-PUF	(3-3)	-	-	97.21	-	-	500
	(1-4)	-	93	-	-	160	-
LP-PUF	k=1	-	-	75.97	-	-	50
	k=2	80	-	84.25	500	-	100
	k=4	-	-	66.03	-	-	2000

10 CONCLUSION AND FUTURE DIRECTIONS

In this work, we presented a new perspective of modelling delay-based Physically Unclonable Functions (or PUFs). Instead of learning the challenge-response functional mapping as machine learning does, we formulated the problem of modelling a PUF by estimating the individual delay parameters belonging to the PUF. We analyzed the textbook evolutionary algorithms and established failure probabilities for textbook genetic algorithms. Consequently, we developed a novel variant of Particle Swarm Optimization suited to the context of PUFs. Through this algorithm, we were able to attack higher order k-XOR PUFs (as high as 20-XOR PUF) as well as LP-PUF instances (on which no prior attack has been reported). Additionally, we also provide a new class of attacks wherein we allow cross-architectural modelling of target PUF. We give mathematical proofs of two kinds of downgrade attacks: ① reducing the security of a k-XOR PUF to a $(k-1)$-XOR PUF, and ② input transformation bypass attack wherein we entirely bypass learnability of the input transformation and target the underlying PUF itself. To the best of our knowledge, *this work is the first of its kind to model delay-based PUFs in this particular way, and propose a new class of modelling attacks on delay-based PUFs.*

So far, this particular attack vector focuses sorely on delay-based PUFs. There are other classes of PUFs like optical PUFs and weak

PUFs wherein the response of input challenge is not typically exposed. Moreover, a precise mathematical modelling for such PUFs is usually not created. An interesting direction of future work shall be to extend the current attack vector on delay-based PUFs onto other kinds of PUFs.

REFERENCES

[1] Thomas Bäck and Hans-Paul Schwefel. 1993. An overview of evolutionary algorithms for parameter optimization. *Evolutionary computation* 1, 1 (1993), 1–23.

[2] Georg T Becker. 2015. The gap between promise and reality: On the insecurity of XOR arbiter PUFs. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 535–555.

[3] Christoph Böhm, Maximilian Hofer, and Wolfgang Pribyl. 2011. A microcontroller SRAM-PUF. In *2011 5th International Conference on Network and System Security*. 269–273. https://doi.org/10.1109/ICNSS.2011.6060013

[4] Qingqing Chen, György Csaba, Paolo Lugli, Ulf Schlichtmann, and Ulrich Rührmair. 2011. The bistable ring PUF: A new architecture for strong physical unclonable functions. In *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*. IEEE, 134–141.

[5] Joan Daemen and Vincent Rijmen. 2002. *The design of Rijndael*. Vol. 2. Springer.

[6] Jeroen Delvaux and Ingrid Verbauwhede. 2013. Side channel modeling attacks on 65nm arbiter PUFs exploiting CMOS device noise. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, 137–142.

[7] Stephanie Forrest. 1996. Genetic algorithms. *ACM Computing Surveys (CSUR)* 28, 1 (1996), 77–80.

[8] Stephanie Forrest and Melanie Mitchell. 1993. What makes a problem hard for a genetic algorithm? Some anomalous results and their explanation. *Machine Learning* 13, 2 (1993), 285–319.

[9] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. 2002. Controlled physical random functions. In *18th Annual Computer Security Applications Conference, 2002. Proceedings.* 149–160. https://doi.org/10.1109/CSAC.2002.1176287

[10] Blaise Gassend, Daihyun Lim, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. 2004. Identification and authentication of integrated circuits. *Concurrency and Computation: Practice and Experience* 16, 11 (2004), 1077–1098.

[11] Boyapally Harishma, Paulson Mathew, Sikhar Patranabis, Urbi Chatterjee, Umang Agarwal, Manu Maheshwari, Soumyajit Dey, and Debdeep Mukhopadhyay. 2020. Safe is the new smart: PUF-based authentication for load modification-resistant smart meters. *IEEE Transactions on Dependable and Secure Computing* 19, 1 (2020), 663–680.

[12] Hector Hung and Vladislav Adzic. 2006. Monte carlo simulation of device variations and mismatch in analog integrated circuits. *Proc. NCUR 2006* (2006), 1–8.

[13] Liliya Kraleva, Mohammad Mahzoun, Raluca Posteuca, Dilara Toprakhisar, Tomer Ashur, and Ingrid Verbauwhede. 2022. Cryptanalysis of Strong Physically Unclonable Functions. *IEEE Open Journal of the Solid-State Circuits Society* (2022).

[14] Raghavan Kumar and Wayne Burleson. 2015. Side-channel assisted modeling attacks on feed-forward arbiter PUFs using silicon data. In *International Workshop on Radio Frequency Identification: Security and Privacy Issues*. Springer, 53–67.

[15] Abhranil Maiti, Jeff Casarona, Luke McHale, and Patrick Schaumont. 2010. A large scale characterization of RO-PUF. In *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, 94–99.

[16] Mehrdad Majzoobi, Farinaz Koushanfar, and Srinivas Devadas. 2010. FPGA PUF using programmable delay lines. In *2010 IEEE International Workshop on Information Forensics and Security*. 1–6. https://doi.org/10.1109/WIFS.2010.5711471

[17] Mehrdad Majzoobi, Farinaz Koushanfar, and Miodrag Potkonjak. 2008. Lightweight secure PUFs. In *2008 IEEE/ACM International Conference on Computer-Aided Design*. 670–673. https://doi.org/10.1109/ICCAD.2008.4681648

[18] Debdeep Mukhopadhyay and Rajat Subhra Chakraborty. 2014. *Hardware security: design, threats, and safeguards*. CRC Press.

[19] Khalid T Mursi, Bipana Thapaliya, Yu Zhuang, Ahmad O Aseeri, and Mohammed Saeed Alkatheiri. 2020. A fast deep learning method for security vulnerability study of XOR PUFs. *Electronics* 9, 10 (2020), 1715.

[20] Phuong Ha Nguyen, Durga Prasad Sahoo, Chenglu Jin, Kaleel Mahmood, Ulrich Rührmair, and Marten van Dijk. 2018. The interpose PUF: Secure PUF design against state-of-the-art machine learning attacks. *Cryptology ePrint Archive* (2018).

[21] Sikhar Patranabis and Debdeep Mukhopadhyay. 2018. *Fault tolerant architectures for cryptography and hardware security*. Springer.

[22] Alain Pétrowski and Sana Ben-Hamida. 2017. *Evolutionary algorithms*. John Wiley & Sons.

[23] Josef Pieprzyk, Thomas Hardjono, and Jennifer Seberry. 2013. *Fundamentals of computer security*. Springer Science & Business Media.

[24] Riccardo Poli, James Kennedy, and Tim Blackwell. 2007. Particle swarm optimization. *Swarm intelligence* 1, 1 (2007), 33–57.

[25] Kuheli Pratihar, Urbi Chatterjee, Manaar Alam, Rajat Subhra Chakraborty, and Debdeep Mukhopadhyay. 2022. Birds of the Same Feather Flock Together: A Dual-Mode Circuit Candidate for Strong PUF-TRNG Functionalities. *IEEE Trans. Comput.* (2022), 1–14. https://doi.org/10.1109/TC.2022.3218986

[26] pyPUF. [n. d.]. pyPUF data. https://pypuf.readthedocs.io/en/latest/data/datasets.html

[27] Vladimir Rožić, Bohan Yang, Jo Vliegen, Nele Mentens, and Ingrid Verbauwhede. 2017. The Monte Carlo PUF. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–6. https://doi.org/10.23919/FPL.2017.8056780

[28] Ulrich Rührmair, Frank Sehnke, Jan Sölter, Gideon Dror, Srinivas Devadas, and Jürgen Schmidhuber. 2010. Modeling attacks on physical unclonable functions. In *Proceedings of the 17th ACM conference on Computer and communications security*. 237–249.

[29] Pranesh Santikellur, Aritra Bhattacharyay, and Rajat Subhra Chakraborty. 2019. Deep learning based model building attacks on arbiter PUF compositions. *Cryptology ePrint Archive* (2019).

[30] G Edward Suh and Srinivas Devadas. 2007. Physical unclonable functions for device authentication and secret key generation. In *2007 44th ACM/IEEE Design Automation Conference*. IEEE, 9–14.

[31] Johannes Tobisch, Anita Aghaie, and Georg T Becker. 2021. Combining Optimization Objectives: New Modeling Attacks on Strong PUFs. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), 357–389.

[32] Arunkumar Vijayakumar, Vinay C Patil, Charles B Prado, and Sandip Kundu. 2016. Machine learning resistant strong PUF: Possible or a pipe dream?. In *2016 IEEE international symposium on hardware oriented security and trust (HOST)*. IEEE, 19–24.

[33] Darrell Whitley. 2001. An overview of evolutionary algorithms: practical issues and common pitfalls. *Information and software technology* 43, 14 (2001), 817–831.

[34] Nils Wisiol. 2021. Towards Attack Resilient Arbiter PUF-Based Strong PUFs. *Cryptology ePrint Archive* (2021).

[35] Nils Wisiol, Christoph Graebnitz, Marian Margraf, Manuel Oswald, Tudor AA Soroceanu, and Benjamin Zengin. 2017. Why attackers lose: Design and security analysis of arbitrarily large XOR arbiter PUFs. *Cryptology ePrint Archive* (2017).

[36] Nils Wisiol, Christoph Gräbnitz, Christopher Mühl, Benjamin Zengin, Tudor Soroceanu, Niklas Pirnay, Khalid T. Mursi, and Adomas Baliuka. 2021. *pypuf: Cryptanalysis of Physically Unclonable Functions*. https://doi.org/10.5281/zenodo.3901410

[37] Nils Wisiol, Bipana Thapaliya, Khalid T Mursi, Jean-Pierre Seifert, and Yu Zhuang. 2022. Neural network modeling attacks on arbiter-PUF-based designs. *IEEE Transactions on Information Forensics and Security* 17 (2022), 2719–2731.

[38] Kan Xiao, Md Tauhidur Rahman, Domenic Forte, Yu Huang, Mei Su, and Mohammad Tehranipoor. 2014. Bit selection algorithm suitable for high-volume production of SRAM-PUF. In *2014 IEEE international symposium on hardware-oriented security and trust (HOST)*. IEEE, 101–106.

[39] Xinjie Yu and Mitsuo Gen. 2010. *Introduction to evolutionary algorithms*. Springer Science & Business Media.