IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2024, No. 1, pp. 501–526. DOI:10.46586/tches.v2024.i1.501-526

CalyPSO: An Enhanced Search Optimization based Framework to Model Delay-based PUFs

Nimish Mishra[1], Kuheli Pratihar[1], Satota Mandal[2], Anirban Chakraborty[1], Ulrich Rührmair[3] and Debdeep Mukhopadhyay[1]

[1] Indian Institute of Technology, Kharagpur, India.
{kuheli.pratihar,anirban.chakraborty}@iitkgp.ac.in
nimish.mishra@kgpian.iitkgp.ac.in,debdeep@cse.iitkgp.ac.in
[2] IIEST, Shibpur, India. 2021itb014.satota@students.iiests.ac.in
[3] TU Berlin, Berlin, Germany, and University of Connecticut, Storrs, USA. ruehrmair@ilo.de

Abstract.
Delay-based Physically Unclonable Functions (PUFs) are a popular choice for "keyless" cryptography in low-power devices. However, they have been subjected to modeling attacks using Machine Learning (ML) approaches, leading to improved PUF designs that resist ML-based attacks. On the contrary, evolutionary search (ES) based modeling approaches have garnered little attention compared to their ML counterparts due to their limited success. In this work, we revisit the problem of modeling delay-based PUFs using ES algorithms and identify drawbacks in present state-of-the-art genetic algorithms (GA) when applied to PUFs. This leads to the design of a new ES-based algorithm called CalyPSO, inspired by Particle Swarm Optimization (PSO) techniques, which is fundamentally different from classic genetic algorithm design rationale. This allows CalyPSO to avoid the pitfalls of textbook GA and mount successful modeling attacks on a variety of delay-based PUFs, including k-XOR APUF variants. Empirically, we show attacks for the parameter choices of k as high as 20, for which there are no reported ML or ES-based attacks without exploiting additional information like reliability or power/timing side-channels. We further show that CalyPSO can invade PUF designs like interpose-PUFs (i-PUFs) and (previously unattacked) LP-PUFs, which attempt to enhance ML robustness by obfuscating the input challenge. Furthermore, we evolve CalyPSO to CalyPSO++ by observing that the PUF compositions do not alter the input challenge dimensions, allowing the attacker to investigate cross-architecture modeling. This allows us to model a k-XOR APUF using a $(k-1)$-XOR APUF as well as perform cross-architectural modeling of BR-PUF and i-PUF using k-XOR APUF variants. CalyPSO++ provides the first modeling attack on 4 LP-PUF by reducing it to a 4-XOR APUF. Finally, we demonstrate the potency of CalyPSO and CalyPSO++ by successfully modeling various PUF architectures on noisy simulations as well as real-world hardware implementations.

Keywords: Physically Unclonable Functions · Modeling Attacks · Evolutionary Algorithms · Particle Swarm Optimization

1 Introduction

Physically Unclonable Functions. Over the last two decades, the advent of ubiquitous computing has led to significant growth and pervasiveness of resource-constrained devices in modern networks. These devices, deployed in-the-wild on edge platforms, face numerous challenges in terms of security and integrity. They are vulnerable to various attack vectors like fault injection and side-channel attacks [BHB19, CAF20]. In such scenarios, generic cryptographic primitives, which rely on the assumption of the explicit secret key generation

Licensed under Creative Commons License CC-BY 4.0.
Received: 2023-07-15 Accepted: 2023-09-15 Published: 2023-12-04

and management procedures on the victim device, fail their security guarantees when secret key recovery is performed using fault injection or side-channel attacks. In this context, *Physically Unclonable Functions (PUFs)* [SD07, GCvDD02] have garnered interest from the security research community. PUFs are appealing due to their inherent feature of being "keyless", thereby mitigating the risk of physical attacks that could potentially leak the key. A PUF can be conceptualized as a physical system that relies on intrinsic hardware randomness as the source of entropy. When given a challenge $\mathbf{c}$ (which is essentially a n-bit bitstring, for some security parameter n) as external stimulus, a PUF essentially behaves as an activated hardware component that depends on nanoscale structural variances to produce an output response r. These nanoscale structural variances can originate from a variety of sources like multiplexer delays [SD07, GCvDD02, MKD10], ring oscillators [MCMS10], start-up values of a static random access memory (SRAM) [BHP11, XRF+14].

Modeling attacks on PUFs. PUFs, by definition, are designed to be "unclonable"; however, in practice, they are not impervious to modeling attacks [RSS+10, RSS+13]. In such attacks, the adversary attempts to create an algorithmic model that can predict the PUF's response to arbitrary challenges with high probability. If such a model is successfully created, it essentially undermines the security guarantee of the PUF, and consequently, any protocols built on top of it. Therefore, developing PUFs resistant to modeling attacks is a significant and intriguing challenge in the security research community. We now note different tactics employed in literature to approach the problem of modeling PUFs. Machine Learning (ML) is one of the most powerful tools for this task due to its ability to learn multi-dimensional hyperplanes, thereby enabling the modeling of the non-linear relationship between challenges and their corresponding responses. While *classical ML attacks* use only challenge-response pairs (CRPs) to train ML models, which are simpler to obtain for adversaries [RSS+10, RSS+13], works like [Bec15a, TAB21, Bec15b, RXS+14, MRMK13] aid the learning process with additional information (like *reliability* or *power/timing side-channel information*) to model higher-order `XOR` APUFs. Although ML-based attacks dominate the literature, there are a few notable works [VPPK16, KB15, RSS+10, RSS+13] that take an alternative approach by employing different types of Evolutionary Search Algorithms (ES) for modeling. While works like [VPPK16, RSS+10, RSS+13] have demonstrated successful ES-based modeling attacks on a specific class of PUFs (namely the Feed-Forward Arbiter PUFs (`FF`-PUF)), to the best of our knowledge, there are no known works that have been able to model higher `XOR` APUFs and other delay-based PUFs that utilize input transformations (like LP-PUF [Wis21b]).

1.1 Related Works

ML attacks on PUFs have been a major line of research that led to the introduction of different attack strategies as well as ingenious countermeasures to thwart such modeling attacks. Classical Arbiter PUF [GLC+04] (APUFs) and its lower order `XOR` variants (up to 6-`XOR`) can be modelled using simple logistic regression techniques [RSS+10, RSS+13]. For the higher order variants, more advanced ML techniques like multi-layer perceptron (MLP) and long short-term memory (LSTM) based neural networks have been used to successfully model 11-`XOR` [WTM+22] and 8-`XOR` APUFs [FKMK22]. Similarly, in [SLZ19], artificial neural networks (ANNs) along with global approximations were used to model 5-`XOR` APUFs. While these attacks have been demonstrated on APUFs and their `XOR`-variants, there have been attacks proposed on variants of delay-based PUFs that use input transformations for additional resilience. For example, interpose-PUFs [NSJ+18] that use input transformations have been successfully modelled using a divide-and-conquer ML strategy [WMP+20]. In addition to classical ML techniques, works such as [TAB21, Bec15a] have incorporated relaxed assumptions on the adversary, such as availability of reliability information, to achieve high accuracy for modeling up to 10-`XOR` APUFs. Likewise, [RXS+14] used side-channel information, like power consumption, to

achieve high accuracies in modeling up to 16-XOR APUFs. However, it's important to note that these side channels require physical access to the PUF device and advanced physical attack capabilities, making them challenging to implement [MRMK13, RXS+14].

The successful modeling of different delay-based PUFs provided a major impetus to consider two major design strategies for these PUFs - ① increasing non-linearity and ② introducing complex input transformations. Such design choices have, to an extent, been able to resist known ML attacks beyond 12-XOR APUFs without usage of additional side-channel information (as in [KB15, BK14]).

1.2 Motivation

Although machine learning has been successful in modeling a wide variety of delay-based PUFs, it is still limiting in the sense that either ① there are no reported works in literature reporting successful ML modeling attacks on PUFs of certain complexities (such as k-XOR APUFs for $k > 12$ or LP-PUF), or ② has additional side-channel requirements (for instance, availability of reliability information). Moreover, due to the predominance of ML attacks in literature, the strategies used to design modeling-resilient delay-based PUFs have mainly accounted for state-of-the-art ML-based techniques only.

However, there exist an entirely different class of attack vectors than ML which has received limited attention from the community. While there certainly have been prior works [VPPK16, KB15, RSS+10] investigating the efficacy of applying evolutionary search (ES) algorithms to model PUFs, the overall implications are still limiting. This is because, first, these works have focused only on a small sub-class of ES algorithms (i.e. genetic algorithms), ignoring a wide array of other ES algorithms. And secondly, such works have generally reported negative results on the modeling accuracy for modeling *stronger* PUFs, as observed in [VPPK16] where the maximum success rate is only 60% against XOR APUFs. This lack of exploration of ES based approaches in the context of PUFs prompted us to delve deeper into the intricacies of modeling techniques of known ML and ES techniques. We believe that a deeper inspection can lead to a better *search optimization* approach for modeling higher order XOR APUFs and other well-known ML-resilient delay-based PUF variants. These insights lead us to ask the following questions:

1. *Does there exist sub-classes of Evolutionary Algorithms other than the well investigated genetic algorithms which are possibly more efficient than modern Machine Learning techniques at modeling non-linear delay-based PUFs?*
2. *Is it possible to devise an Evolutionary Search Algorithm that can efficiently model higher order XOR APUFs and other delay-based PUFs like LP-PUF that are shown to be resilient against state-of-the-art ML techniques?*
3. *Given the architectural topologies of PUF chains, is it possible to formulate a cross-architectural modeling attack such that a particular class of PUFs can be approximated by a PUF belonging to another class?*

1.3 Contributions

In this work, we answer the aforementioned questions in the affirmative. In particular, we make the following contributions:

❶ **An alternative modeling strategy for delay-based PUFs:** The current state-of-the-art Machine Learning algorithms have their limitations in modeling higher order XOR PUFs and certain classes of complex PUF architectures like LP-PUF. Recognizing these limitations and the lack of adequate exploration into the use of evolutionary search (ES) algorithms outside of genetic algorithms (GA), this work demonstrates how ES algorithms other than GAs can outperform machine learning in the modeling of delay-based PUFs.

Specifically, a new model of attacking delay-based PUFs is introduced, which focuses not on learning the decision hyperplane (as ML does), but rather on learning the cumulative effect of the delay parameters (parameters which drive the behaviour of delay-based PUFs). This allows us to construct modeling strategies for PUF architectures that reportedly show stronger resilience against ML based attacks.

❷ **CalyPSO - novel modeling framework for PUFs:** As part of our investigation into evolutionary search (ES) algorithms, we review the *negative* results in literature wherein a rather small sub-class of ES algorithms (i.e. genetic algorithms) failed to model complex delay-based PUFs, and develop a causal understanding of the same. We shed light on why certain properties of textbook genetic algorithms are fundamentally unsuitable for modeling PUFs. Consequently, we propose a unique *genotype representation* that is specifically tailored for delay-based PUFs, and introduce a new variant of Particle Swarm Optimization (PSO) algorithm called CalyPSO, inspired by the *natural process of amoebic asexual reproduction.*

❸ **Demonstrate modeling of k-XOR APUF (upto $k = 20$) and other delay PUFs:** Empirically, CalyPSO outperforms both machine learning as well as prior ES based attack vectors on PUFs. CalyPSO, 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 than reported in literature. We also show high modeling accuracy on noisy simulations as well as hardware implementations of different variants of k-XOR APUFs (hardware implementations for 4-XOR to 12-XOR and noisy simulations for 13-XOR to 20-XOR). We further demonstrate attacks on delay-based PUFs that derive their security from input transformations. One prime example is LP-PUF, which has not been successfully modelled yet in literature.

❹ **CalyPSO++ - enhanced framework for cross-architecture modeling:** Additionally, the *genotype* representation we propose for PUFs, combined with the specification of CalyPSO, allows us to investigate a previously unexplored class of attacks on PUFs: cross-architectural attacks. We propose an enhanced version of our framework, called CalyPSO++, which introduces a novel attack strategy that allows us to model target PUFs of one architecture using the mathematical model of other PUF architectures. Specifically, we demonstrate the reduction of security of a k-XOR APUF to a $(k-1)$-XOR APUF as well as bypass complex input transformations such as substitution permutation networks, enabling us to successfully model k-LP-PUF by reducing it to a k-XOR APUF. Using CalyPSO++, we demonstrate successful modeling of hardware implementations of BR-PUF, $(11, 11)$ i-PUF and 4 LP-PUF using different variants of k-XOR APUFs.

1.4 Organisation

The rest of the paper is organised as follows. We provide a brief background on APUF delay model, evaluation metric, ML modeling and newly proposed LP-PUF in Sec. 2. Next, in Sec. 3 we provide a deeper insight into learning approaches of ML attacks and Sec. 4 discussed the pitfalls of known ES algorithms in context of PUFs. In Sec. 5, we introduce a novel evolutionary search algorithm named CalyPSO to model delay based PUFs. We enhance the framework in Sec. 6 and propose CalyPSO++ that shows PUF modeling through cross-architectural learning. Sec. 7 shows the experimental results across different families of delay PUFs. Finally, we provide a discussion on the future of delay-based PUFs and conclude in Sec. 8.

2 Preliminaries

In this section, we present the background information on Arbiter PUFs, including their delay model. We discuss the non-linear design approaches undertaken in literature to make

APUFs resistant against ML-based modeling attacks.

2.1 Additive delay model: Case study of an Arbiter PUF

Additive Delay Model. Arbiter Physical Unclonable Function (APUF), originally proposed in [GLC+04], is the first delay-based PUF design upon which many modern delay-based PUF designs are based. An APUF takes a single challenge $\mathbf{c} = (c_1, c_2, c_3, ..., c_n) \in \{-1, 1\}^n$ as input (where n is the challenge length) and outputs a single bit response $r \in \{-1, 1\}$. In digital logic, the algebraic representation of 1 and -1 corresponds to the `LOW` and `HIGH` states respectively. Structurally, each stage of APUF receives two input pulses from its previous stage, propagates those pulses through a challenge dependent path, and adds additional "*stage delay*" depending on stage delay parameter δ (which mathematically quantifies the stage's internal nanoscale level variations). The final response r is the outcome of the "*additive delay*" Δ, which is the cumulative *delay* effect from all n stage delays. The *additive delay* model is defined as

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

where c_i and c_j denote the challenge bits, and $\delta_1^{(i)}$ and $\delta_{-1}^{(i)}$ represent the delay parameters for the i-th stage when $c_i = 1$ and $c_i = -1$, respectively. The arbiter in this process is the `D` flip-flop which generates the final response. The behavior of the flip-flop is abstracted using the `sgn` function $\mathcal{G} : \mathcal{G}(\Delta) = r$. This notation is consistently used throughout the rest of the paper.

Unintentional linearity. From a ML perspective, the additive delay model of the APUF (and its behavior) has a vulnerability that can be exploited. Note that δ (c.f. Eq 1) captures the stage delay of the i-th stage given input challenge bit c_i. As a result, the cumulative delay at the i-th stage depends on the *cumulative* delays up to $(i-1)$-th stage and the stage delay of the i-th stage [WGM+17]. Mathematically therefore, Eq. 1 can be rewritten as $\sum_{i=1}^{n} \omega_i \Phi_i$, where $\omega_i = \frac{1}{2}(\delta_{-1}^{i} + \delta_1^{i} - \delta_{-1}^{i-1} + \delta_{-1}^{i-1})$. Here δ_y^x represents the delay parameter corresponding to the x-th stage and input challenge bit y (for $y \in \{-1, 1\}$). Additionally, $\Phi_i = \prod_{j=i}^{n} c_j$ is termed as the *parity* vector and is derived from the publicly known input challenge $\mathbf{c}$. It should be noted that $\Phi_i = \prod_{j=i}^{n} c_j$ is publicly known to the adversary, making Eq. 1 *linear* in $\boldsymbol{\omega}$, thereby leading to successful ML modeling.

2.2 Non-linear design variants: defences against ML modeling

An effective defense strategy against such attacks would involve "de-linearizing" $\Delta = \sum_{i=1}^{n} \omega_i \Phi_i$, i.e., breaking the linear relationship between Δ and $\boldsymbol{\omega}$ to make it more difficult for machine learning algorithms to learn a separable decision hyperplane on $\mathcal{G}(\Delta)$. This can be achieved through two approaches: ① introducing explicit non-linearity, and ② input transformation.

Explicit introduction of non-linearity. To introduce explicit non-linearity [NSJ+18], `XOR` APUFs are designed by combining several APUF chains with a non-linear `XOR` function, which is known to be challenging for machine learning algorithms. Specifically, a k-`XOR` APUF consists of k arbiter chains that compute the `XOR` of k individual responses. Each APUF in the chain is given an input challenge $\mathbf{c} = (c_1, c_2, c_3, ..., c_n) \in \{-1, 1\}^n$, and the XORed output from each chain is the response r. Mathematically, a k-`XOR` APUF can be represented by the following equation: $r = \mathcal{G}(\Delta_1) \oplus \mathcal{G}(\Delta_2) \oplus \mathcal{G}(\Delta_3) \oplus \cdots \oplus \mathcal{G}(\Delta_k)$. The equation can be interpreted as a series of k-`XOR` chains, where the i-th chain is represented

by delay Δ_i and the corresponding response generated is given by $\mathcal{G}(\Delta_i)$. To further elaborate, substitution of Eq. 1 (additive delay model Δ) gives the following arrangement:

$$r = \mathcal{G}\left(\sum_{i=1}^{n}\left((\delta_1)_{c_i}^{(i)}\prod_{j=i}^{n}c_j\right)\right) \oplus \cdots \oplus \mathcal{G}\left(\sum_{i=1}^{n}\left((\delta_k)_{c_i}^{(i)}\prod_{j=i}^{n}c_i\right)\right) \tag{2}$$

It is worth noting that each individual APUF in the k-XOR APUF has its own delay parameter set $\boldsymbol{\delta}_i$, where $1 \leq i \leq k$. From a ML perspective, a higher value of k makes it more challenging for ML models due to the increased non-linearity introduced by the XOR operation among the responses.

Input transformations. An alternative approach to achieve the same objective is to adopt design principles from block ciphers to *hide* the publicly available challenge bits through input transformations [MKP08, SAS+19, Wis21b, PCA+22]. Specifically, these defense mechanisms employ a one-way function $f_s(\mathbf{c})$ that is parameterized by a secret key s, which transforms the original challenge $\mathbf{c} = \{c_1, c_2, \cdots, c_n\}$ into a *private* challenge $\mathbf{c}^{'} = \{c_1^{'}, c_2^{'}, c_3^{'}, ..., c_n^{'}\}$. This transformed challenge is then used as input to the PUF. In a generic k-XOR equation with input transformations, we can write it as follows:

$$r = \mathcal{G}(\sum_{x=1}^{n}(\delta_1)_{f_s(c)_x}^{(x)}\prod_{y=x}^{n}f_s(c)_y) \oplus ... \oplus \mathcal{G}(\sum_{x=1}^{n}(\delta_k)_{f_s(c)_x}^{(x)}\prod_{y=x}^{n}f_s(c)_y) \tag{3}$$

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

2.3 PUF Evaluation Metrics

Given a set of challenge-response pairs, there exist metrics which allow quantifying the behaviour correctness of the PUF. We define the most important of such metrics here:

1. *Uniformity:* It estimates the distribution of response bits (either 1 or -1) against a challenge set. Ideally, for a PUF, the likelihood of either response bit should be no better than an random unbiased coin toss.
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) [RYV+17].

3 A deeper look into ML attacks and the learning process

We note that certain delay-based PUFs such as higher XOR APUFs (more than 12-XOR) and PUFs utilizing input transformations (like LP-PUF) have been effective in resisting machine learning modeling attacks[1]. Consequently, in this section, we summarize the ML attack strategy for better comparison with alternative modeling strategies, particularly evolutionary search techniques. We begin with the following observation:

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

[1]For the rest of the paper, we exclude discussions on use of additional side-channel information.

The observation **O1** directly motivates the two design approaches employed to counter machine learning (ML) attacks (c.f. Sec. 2.2). Literature indicates that beyond a certain threshold, the ① induced non-linearity with increasing k-XORs and ② input transformations distort the decision hyperplane (i.e. make the decision hyperplane increasingly inseparable) to a point where classic as well as state-of-the-art ML attacks struggle. This observation raises an intriguing question: *can alternative approaches, such as evolutionary search, be optimized for modeling delay-based PUFs in order to explore PUF families known to be resilient against ML attacks in existing literature?* To address this, we conduct a closer examination of the intrinsic design and architectural principles of delay-based PUFs, and formulate a dedicated modeling strategy.

An alternate modeling strategy

It is well-established in literature that the nanoscale structural variances of a delay-based PUF can be *approximated* by a normal distribution with an appropriate variance, which includes both the inherent delays of CMOS circuitry [HA06] and additional noise that arises in hardware [DV13]. We leverage this information to adopt a new perspective towards modeling delay-based PUFs. Instead of treating the problem of modeling PUFs as a decision hyperplane learnability problem, as commonly done in machine learning (ML), we focus on the *combined effect* of the individual stage delays in the PUF.

Our representation of PUFs is a set of normal random variables $\boldsymbol{\delta} = \{\delta_1, \delta_2, \delta_3, \ldots, \delta_n\}$ where $\delta_i \sim \mathcal{N}(0, \sigma^2) \, \forall \, 1 \leq i \leq n$ for standard deviation σ. This vector $\boldsymbol{\delta}$ characterizes the individual stage delays in a PUF (c.f. Eq 1). The core idea is that every delay-based PUF generates responses based 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 represents the *cumulative delay* arising from the operations of the individual delay parameters $\boldsymbol{\delta} = \{\delta_1, \delta_2, \delta_3, \ldots, \delta_n\}$. This observation forms the basis of our attack strategy proposed 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 for $\delta_i^{'} \sim \mathcal{N}(0, \sigma^2) \, \forall \, 1 \leq i \leq n$, and thus another *combined delay* $\Delta^{'}$ such that $r = \mathcal{G}(\Delta) = \mathcal{G}(\Delta^{'})$. Since this strategy does not explicitly require finding out the decision hyperplane solving the binary classification problem on the PUF's responses, it is able to subvert the popular defence strategies in design of traditional delay-based PUFs, which rely on convoluting the decision hyperplane.

A logical claim from observation **O2** is that $\mathcal{G}(\Delta^{'})$ serves as a *model* of the target PUF $\mathcal{G}(\Delta)$ due to the relationship $r = \mathcal{G}(\Delta) = \mathcal{G}(\Delta^{'})$. Hence, instead of learning the decision hyperplane, our strategy is to *search* through the space of all PUFs parameterized by $\boldsymbol{\delta}^{'}$ to find a PUF with a cumulative delay $\Delta^{'}$ such that $\mid \Delta - \Delta^{'} \mid \leq \epsilon$ (where ϵ represents an acceptable error). Furthermore, approaching the problem of modeling PUFs from evolutionary search perspective allows us to launch innovative *cross-architectural* attacks- approximating a target delay Δ in a search space of *simpler* PUFs (see Sec. 6). In this study, we demonstrate cross-architectural attacks on two use-cases: ① reducing the security of a k-XOR APUF to a $(k-1)$-XOR APUF, thereby reducing the degrees of non-linearity; and ② reducing the security of a k-LP-PUF to an equivalent k-XOR APUF, thereby nullifying the input transformations. These *cross-architectural* attacks highlight our perspective of viewing PUF modeling as a *search problem* on the delay parameter set $\boldsymbol{\delta}$, which allows for a more extensive search space. Thereby, we make the following observation, which serves as the foundation for designing our attack:

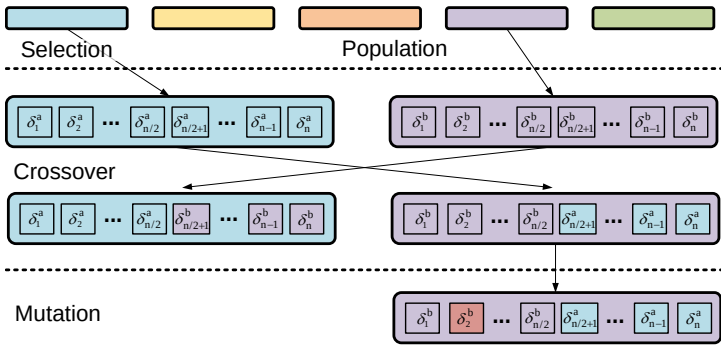

Figure 1: An iteration of the genetic algorithm on the genotype suggested in this work.

✓ **O3**. Specifically, as the ① non-linearity increases or as ② input transformations achieve more diffusion (we use the term *diffusion* in the same context as block ciphers), the decision hyperplane becomes more and more convoluted. However, both the defences ① and ② have no effect whatsoever on Δ, which is where we attack.

To summarize, in this section we suggest an alternative modeling strategy for *searching through the victim PUF's parameter space*. In the subsequent sections, we further develop observation **O2** to create an evolutionary search algorithm based modeling framework called CalyPSO, while observation **O3** forms the basis for CalyPSO++ for cross-architecture modeling. However, for brevity, we first discuss the limitations and pitfalls of prior ES-based works on PUFs in the next section. This analysis helps us conceive an improved strategy to eventually culminate our search-based modeling framework.

4 Towards PUF-aware evolutionary algorithm design

While the literature reports a significant number of ML based modeling attacks, the use of Evolutionary Search (ES) techniques as an alternate strategy has been relatively less explored. Although some attempts have been made in the literature using a small subset of evolutionary algorithms, specifically genetic algorithms [VPPK16, KB15], the reported modeling accuracy did not exceed 60% when defense mechanisms like increased non-linearity and input transformations were introduced. Furthermore, recent literature that uses evolutionary search algorithms has not been successful in modeling higher order XOR APUFs or PUFs with input transformations, such as LP-PUF. Therefore, it is crucial to analyze the approaches and strategies employed in prior works in order to develop better strategies for the search objective based on the observation **O3** (cf. Sec. 3).

In general, Evolutionary Search (ES) algorithms are a class of algorithms used to solve computational problems with well-defined search spaces [BS93, YG10, PBH17]. Theoretically, with infinite computational resources, it would be possible to exhaustively search the entire search space using brute force or randomized search strategies. However, ES algorithms prevent exploration of the entire search space by making intelligent choices using a defined fitness function that quantifies the progress of the search with respect to a global optimum. The algorithm balances two opposing forces [Whi01]: ① *exploration*, and ② exploitation. Too low exploration can result in the algorithm getting stuck in local optima, while overly large exploration is akin to a completely random search. Too much exploitation can lead to the algorithm getting trapped in local optima, while too little exploitation is essentially a random search of the search space.

Previous research on using evolutionary algorithms for modeling Physical Unclonable Functions (PUFs) has primarily focused on a limited subset of ES algorithms, specifically genetic algorithms, to attack delay-based PUFs such as APUF, 4-XOR APUF, Feed-Forward

APUFs, and analog PUFs [VPPK16, KB15, RSS+10]. Genetic algorithms are inspired by the process of gene evolution in nature and are based on the principle of *survival of the fittest*, aiming to mimic nature's strategy. Every genetic algorithm is a composition of the following four sub-parts [For96]: ① **Genotype**, ② **Selection operator**, ③ **Crossover operator**, and ④ **Mutation operator**. The genotype represents the genetic encoding of the problem being studied. The selection operator determines which members of the population will reproduce in a given generation. The crossover operator controls how the genotypes of two individuals in the population are mixed to create offspring, mainly influencing the **exploitation** phase of the genetic algorithm. Lastly, the mutation operator introduces random mutations into the population as it evolves, controlling the **exploration** phase of the genetic algorithm. 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 [VPPK16, KB15, RSS+13] on standard genetic algorithms choose the hyperparameters based on the default settings of genetic algorithms. However, the genotype representation requires careful consideration when applied to PUFs. In [VPPK16, KB15], the genotype representation is a set of table entries. While [KB15, RSS+13] focuses on Feed-Forward APUF designs and do not explore XOR-based designs, [VPPK16] explicitly acknowledges the limitations of genetic algorithms in breaking PUFs.

Drawback in textbook genotype representation

We now summarize the associated problems with genetic algorithms (GA) for PUFs. As shown in Fig. 1, the GA chooses two PUF instances (technically called the *parents*) from the population of all available instances and performs a *crossover* [Hil04]. The newly formed PUF instance (or the *child* instance) undergoes mutation in order to evolve. Finally, the *fittest* members of the population undergo similar evolution in later iterations of the algorithm. Concretely, the *fitness* of every member of the population is computed as: $fitness = accuracy - bias$, where *accuracy* abstracts the % of correct predictions that the *child* PUF does for the victim PUF's responses (for the same challenge set). Likewise, *bias* abstracts the victim PUF's likelihood to generate a certain response (either 1 or -1) more than the other bit.

Traditional GA's approach of *combining* two PUF parameters to create new PUF parameters does not necessarily guarantee a *fitter* PUF than the previous population. This is because the delay parameters $\boldsymbol{\delta}^a = \{\delta_1^a, \delta_2^a, \cdots, \delta_n^a\}$ and $\boldsymbol{\delta}^b = \{\delta_1^b, \delta_2^b, \cdots, \delta_n^b\}$ have been learnt based on the *collective behaviour* of the *parent* PUFs. Mixing the halves of each *parent* into new children does not take into account the aggregate effect of the other half in the parent PUF instance. Moreover, such a child instance effectively destroys the relationship between the different stages in the parent instances.

A PUF-aware genotype representation

In this work, we propose a novel genotype representation for PUFs that eliminates the need for crossover operations in evolutionary algorithms. We also extend our search beyond traditional genetic algorithms (GAs), which are just one type of evolutionary search (ES) algorithm, to explore other classes of ES algorithms that do not require genotype crossover. Specifically, we leverage the concept of *de-linearization* defenses and input challenge transformations to enhance the security of PUFs. However, even in the presence of such defenses, the functionality of delay-based PUFs still relies on the $\boldsymbol{\omega}$ vector (as discussed in Sec. 2.1 and observation **O3**) that is used to evaluate Δ. In our work, we utilize an evolutionary search technique with our improved genotype representation to directly

estimate Δ and 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 $\boldsymbol{\omega}$.

5 CalyPSO: A PUF-aware evolutionary optimizer

With the newfound intuition for an improvised genotype representation, we now proceed to introduce a novel Particle Swarm Optimization variant named CalyPSO. CalyPSO is inspired by the biomimicry [HYZ23] of *amoebic reproduction* as it allows for an intuitive evolutionary strategy on the PUF-aware genotype representation introduced in Sec. 4. We note that CalyPSO is based on the ideas of *swarm* optimizers, which are fundamentally different from the previously explored genetic algorithms in PUF literature. Unlike gene intermixing based genetic algorithms, swarm algorithms [PKB07] draw upon the efficiency of various available swarms in the animal kingdom that allow them to achieve an objective of collective interest, thereby relying on *collective behaviour* rather than genetic intermixing. One textbook example is classical 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.

Following the rationale of swarm optimizers, CalyPSO also 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, while overdoing exploitation risks being caught in local extremums. CalyPSO incorporates the evolutionary approach of amoebic reproduction into the traditional PSO strategy using our improvised genotype representation. In the rest of the section, we elaborate on the design intuition and provide a detailed description of our algorithm.

5.1 Algorithm design intuition

For brevity, let us consider the PUF's delay parameters $\boldsymbol{\delta}^a = \{\delta_1^a, \delta_2^a, \delta_3^a, \cdots, \delta_n^a\}$ learnt over a period of several generations of the evolutionary learning algorithm. Unlike the changes a genetic algorithm does, we do not want to discard half of the parameters learnt and replace them with parameters of another PUF (this 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. 2), 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 overall PUF response, leading to a change in 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 $\boldsymbol{\delta}^a$

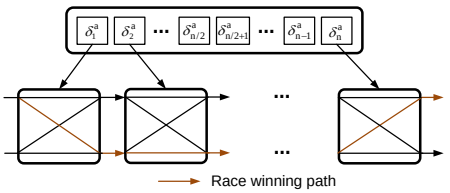

Figure 2: Each δ_i^a affects the i-th stage. The **combined effect** Δ is obtained by the combined effects of each of δ_i which allow one path to win the race.

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 $(\boldsymbol{\delta}^{'})^a$ to be still influenced by parameters from the previous generation (i.e. $\{\delta_1^a, \delta_2^a, \delta_3^a, \cdots, \delta_{i-1}^a, \delta_{i+1}^a, \cdots, \delta_n^a\}$), while also focusing on improving overall accuracy on target response set. This allows the algorithm to evaluate the effect of this *mutation* on the *combined delay model* $\Delta = \sum_{i=1}^{n}(\delta_i^a \Phi_i)$ by changing a single parameter δ_i^a to $(\delta^{'})_i^a$ and preserving all other parameters in $\boldsymbol{\delta}^a$. 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. It is noteworthy that when modeling a PUF instance, it may not be necessary to learn the individual delay parameters for all the stages. If a modeling strategy can accurately capture the behavior of each stage, such as the probability of the cross or straight path winning, the overall behavior of the PUF for any given challenge can be effectively modelled with high probability.

5.2 Algorithm design decisions

In this work, we develop a novel evolutionary search algorithm named CalyPSO. We derive the motivation and design decision of our algorithm by mapping the PUF search problem (cf. Sec. 3) to the following problem in nature: *how does a population of amoeba move towards a food source (i.e. an objective)?* Consider the adjoining Fig. 3. 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 its *fitness*, each member of the population takes one step towards the direction which takes that member closer to the food source. Here the fitness of an individual member of the population can be adjudged as the remaining distance from the food source. Intuitively, the higher the peak is in the landscape, the more *fit* an amoeba becomes when it reaches there.

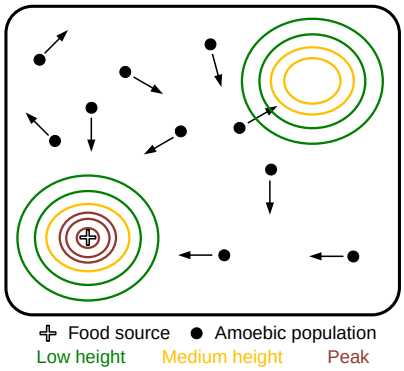

Figure 3: Initial amoebic population with a food source in the landscape defined by contours. It is assumed that the food source is at the highest peak in the landscape.

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 PSO 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, such a broadcast does not benefit the search process in the context of PUFs. As discussed in Sec. 4, any two instances of PUF must evolve independently without influencing one another. In other words, every member of the population shall 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:

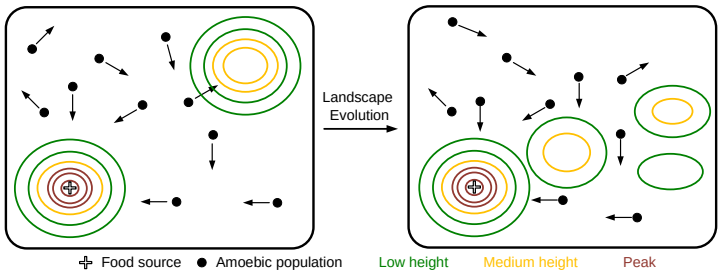

Figure 4: An example of landscape evolution. Arrows indicate the directions in which the population advances. Note how the contours change as the landscape evolves.

- **C2**. A generic PSO approach is not much helpful in context of PUFs as we cannot use one PUF instance to evolve another PUF instance.

5.2.1 Solving C1: Landscape evolution

The generic landscape (as depicted in Fig. 3) represents the initial configuration of the search space and is dependent on the particular search problem and the initial input parameters/conditions. In our context, this landscape is the target PUF mapping $\mathcal{P}$ (c.f. Sec. 1.2). Ideally, a landscape with a large amount of data extracted from $\mathcal{P}$ would be smooth, allowing particles to converge towards the final objective. However, in practical scenarios, the available data for the algorithm is often limited, resulting in a more challenging landscape for the particles to navigate.

Solution to C1. We adapt the concept of evolving landscapes [Jon95, Pai11] into our algorithm. Given limited data from $\mathcal{P}$, we construct partial landscapes from subsets of the overall data. Over a sufficient number of generations, the members of the population will have evolved over varying landscapes. Since a *false* optimum will not occur in *all* subsets used in landscape evolution (otherwise it would not be a *false* optimum), a member of the population stuck in such a *false* optimum in *one* generation will become unstuck in subsequent generations. Eventually, using landscape evolution, the *false* optimums will smooth out, leaving the global optimum visible for convergence.

Consider the illustration in Fig. 4. Since we are utilizing only a subset of the total available data, it is possible for *false* contours to emerge. Such contours may contain *false* optimums that would trap the algorithm's convergence. However, as the landscape evolves during the algorithm's execution, these *false* contours will not occur in every sampled subset. Therefore, over multiple runs of the algorithm, the population members that may have become trapped in *false* contours will also gradually converge towards the global optimum. Hence the solution to challenge **C1** is:

- **S1**. Using landscape evolution as an essential portion of the algorithm allows it to prevent from being caught up in local extremums.

5.2.2 Solving C2: Asexual reproduction

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 in case of PUFs because that would require mixing genotypes from two members of the population (which does not perform any better than the crossover operator in genetic algorithms). This means that *search space exploration* is no longer possible without altering the generic PSO as every member of the population will simply keep on doing a random search in their own specific directions.

Solution to C2: We merge the generic concept of a PSO with *amoebic reproduction*. 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 avoiding the pitfalls that genetic algorithm has in the context of PUFs (c.f. Sec. 4). 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 over where in the landscape the populated children will spawn [2], thereby risking bad solutions in the search process. Consider Fig. 5. 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. In short, we solve challenge **C2** as:

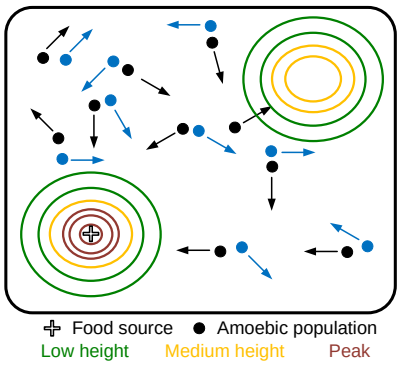

Figure 5: An example of amoebic reproduction applied in the algorithm.

- **S2**. Using amoebic reproduction in place of swarm optimisation helps reproduce *fitter progeny* through a more *PUF-aware* evolutionary strategy.

5.3 Algorithm description

With attack intuition and solutions **S1** and **S2** in place, we now proceed to develop our framework CalyPSO, as detailed in Algo. 1. The algorithm is invoked with `ATTACK_WRAPPER` which does essential initializations: ① the *victim* PUF's challenge-response tuple (C, R) that needs to be modelled (line 20), ② the `target_puf_arch` which abstracts the details of the architecture of the victim PUF (line 21), ③ `le_parameter` which dictates the frequency of landscape evolution (line 22), ④ `delay_param` in line 23 which controls the number of delays in the set $\boldsymbol{\delta}$ to perturb in one generation (c.f. Sec. 5.1), and ⑤ `population_list` in (line 24) which holds an initial population of 500 randomly sampled elements of the genotype detailed in Sec. 4. The *random* sampling of initial population is abstracted in line 6 by `RANDOM(target_puf_arch)`, wherein we randomly sample n normally distributed stage delays to construct the set $\boldsymbol{\delta}$ for each PUF in the initialized population.

Then begins the evolutionary search. In every iteration (alternatively, in every *generation*), three main operations take place. First, ① `compute_population_fitness` (line 26) is invoked on the entire population to compute *fitness* of every member of the population (c.f. Sec. 4). Intuitively, *fitness* quantifies how *close* a member of the population is to the objective of successfully modeling the victim PUF's behaviour. Secondly, ② `AMOEBIC_REPRODUCTION` (line 29) is invoked which kicks in the amoebic reproduction part. Finally, ③ `STEP` (line 30) uses `delay_param` to perturb the delay set $\boldsymbol{\delta}$ of every member of the population. Apart from these, we use helper functions `SORT` (line 27) and `TRIM` (line 28) to remove the less fitter members of the population and maintain the size of `population_list` at `population_size`.

[2] because the children are produced by intermixing two parent genotypes

Algorithm 1 CalyPSO

```
1: procedure INITIALIZE_POPULATION(target_puf_arch)
2:     set population_size ← 500
3:     set population_list ← NULL
4:     while population_list.size() != population_size do
5:         /* Randomly generate an instance of target_puf_arch*/
6:         population_list.append(RANDOM(target_puf_arch))
7:     return population_list
8: procedure COMPUTE_POPULATION_FITNESS(C, R, population_list)
9:     for member in population_list do
10:        predicted_responses = member.evaluate(C)
11:        member.fitness = /*compare similarity between R and predicted_responses*/
12:    return population_list
13: procedure AMOEBIC_REPRODUCTION(population_list)
14:    population_list.append(CLONE(population_list))
15: procedure STEP(population_list, delay_param)
16:    for member in population_list do
17:        /*Add normal noise to each member's delay parameters contained in the list delay_param*/
18: procedure ATTACK_WRAPPER
19:    set GENERATION ← 0
20:    (C, R) ← challenge-response tuple of the target PUF
21:    set target_puf_arch
22:    le_parameter ← landscape evolution hyperparameter
23:    set delay_param ← 1                          ▷ delays to perturb in one generation.
24:    population_list ← initialize_population(target_puf_arch)
25:    while True do
26:        call compute_population_fitness(C, R, population_list)
27:        population_list.sort()                                  ▷ Sort based on fitness
28:        population_list.trim()                          ▷ Trim population to size 500
29:        call amoebic_reproduction(population_list)
30:        call step(population_list, delay_param)
31:        if not GENERATION % le_parameter then
32:            /*randomly sample a new challenge set C'*/
33:            R ← target PUF response on C'
```

CalyPSO unveils the power of search algorithms by adopting two design decisions - ① instead of learning delay parameters, it tries to model the behaviour of the target PUF and ② uses bio-mimicry to solve the inherent challenges of ES algorithms in the context of PUFs. Using CalyPSO, we have been able to model different variants of delay PUFs, including higher order XOR APUF ($k > 12$) and LP-PUF with reasonable number of CRPs (more details in Sec. 7). One important point to note here is that the framework relies on a mathematical model (c.f. Sec. 2.1) of same family of PUF in order to model the victim PUF. In other words, to model a specific PUF $\mathcal{P}$ belonging to a certain PUF architecture family, CalyPSO requires a mathematical model to simulate PUF instances that belong to the same architecture and eventually converge into one instance P', such that both P and P' behave in a similar fashion (albeit with negligible error margin). At this juncture, we ask a fascinating question: *can a PUF belonging to a certain architectural family be modelled using instances belonging to a different architectural family?* In the next section, we answer this question in the affirmative. We enhance our framework as CalyPSO++ that, for the first time in PUF literature, demonstrate cross-architecture modeling using ES.

6 CalyPSO++: Cross-Architectural modeling of PUFs

The success of ML attacks against certain classes or families of APUFs (and lower-order XOR variants) can be attributed to the linear to lower-order non-linear complexity of the function $\Delta = f(\boldsymbol{\omega})$ which is exploited by ML models. By introducing non-linearity and input transformations, researchers have been able to resist state-of-the-art ML attacks on complex PUF architectures [WTM+22, Wis21b]. Whereas, CalyPSO has been able to bypass both the defence strategies by transforming PUF modeling problem into a search problem and using a novel evolutionary algorithm and crafted genotype representation. However, it is important to highlight that CalyPSO has its own limitations as it requires a mathematical model of the targeted family. In this section, we alleviate this restriction by introducing CalyPSO++, a cross-architectural PUF modeling framework that allows modeling of PUFs belonging to an architectural family by approximating simulations from another architectural family. The approach of perceiving PUF modeling as a search problem presents a unique advantage. The PUF search space (cf Sec. 5.2) not being defined by the mathematical model of the target PUF allows cross-architectural modeling where instances from a different family can be used to search for the approximate behaviour of the target PUF.

6.1 Modeling k-XOR APUF using $(k-1)$-XOR APUF

To model a k-XOR APUF using a $(k-1)$-XOR variant, we use the idea of FORMULA-SATISFIABILITY. A FORMULA-SATISFIABILITY problem instance is composed of the following:

- n boolean variables: $x_1, x_2, \ldots, x_n$
- m boolean connectives: $\wedge, \vee$

Given a formula $\mathcal{F}$ composed of the aforementioned components, FORMULA-SATISFIABILITY asks whether there is an assignment to $\{x_1, x_2, ..., x_n\}$ such that $\mathcal{F} = 1$. It is well established that FORMULA-SATISFIABILITY is an NP-complete problem [CLRS22]. XOR APUFs achieve de-linearization of the relationship between Δ and $\boldsymbol{\omega}$ (cf. Eqn. 1 in Sec. 2.1) by increasing the number of XORs; thereby, increasing the non-linearity in modern PUF designs (c.f. Sec 2.2). In order to reduce *degrees of freedom*, we model the problem of learning a k-XOR APUF as a FORMULA-SATISFIABILITY problem. Concretely, at the hardware level k APUF outputs $(0, 1)$ are XORed, which can be represented by the function:

$$R = f_1(\mathbf{C}, \boldsymbol{\delta}_1) \oplus f_2(\mathbf{C}, \boldsymbol{\delta}_2) \oplus \cdots \oplus f_k(\mathbf{C}, \boldsymbol{\delta}_k) \tag{4}$$

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. Eq. 4 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, Eq. 4 can be re-written as:

$$R = f_1(\mathbf{C}, \boldsymbol{\delta}_1) \oplus f_2(\mathbf{C}, \boldsymbol{\delta}_2) \oplus \ldots f_{k-1}(\mathbf{C}, \boldsymbol{\delta}_{k-1}) \oplus b \tag{5}$$

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. 5 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:

✓ **O4**. Any k-XOR APUF 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 APUF to a $(k-1)$-XOR APUF.

The observation **O4** forms the basis of our cross-architectural framework CalyPSO++. While the core algorithm uses the idea of Algorithm. 1, CalyPSO++ does not require the mathematical model of the target PUF architecture as one of the inputs. It starts with one of the known mathematical models (for eg. additive delay model of $(k-1)$-XOR APUF to model k-XOR APUF) and proceeds as usual. Using CalyPSO++, we have been able to model 1 LP-PUF from APUF and 2-XOR from 1-XOR APUF (APUF). Other that XOR APUFs, which are basically variants of APUF, CalyPSO++ is able to model entirely different architecture, like Bistable Ring (BR) PUF [CCL+11, XRHB15] from 4-XOR APUF, which demonstrate true cross-architectural prowess (more details in Sec. 7).

6.2 Bypassing input transformations

One of the benefits of viewing PUF modeling as a search problem is that the input transformation used in some PUF architectures to protect against ML attacks does not impact the evolutionary search process within the PUF search landscape. As an example, we leverage the input transformation of LP-PUF, which utilizes a substitution-permutation network (SPN) for diffusion [Wis21b]. LP-PUF is a delay-based PUF that applies an SPN to transform the input challenge set C into $C^{'}$ before passing it to a standard k-XOR APUF. The SPN's parameters are generated by a series of k APUFs, making the SPN's security tied to the hardware itself. This transformation hides the actual challenge input to the innermost k-XOR layer from adversaries and ML models, providing the required non-linearity to resist ML-based attacks. To the best of our knowledge, no ML-based or evolutionary algorithm (ES)-based attack has been successful against LP-PUF, as mapping C directly to the response set R would require learning the SPN without knowledge of $C^{'}$, which is a challenging problem. However, CalyPSO++ is capable of attacking LP-PUF's SPN because it considers both the structure of the SPN and the k-XOR APUF components of LP-PUF as part of its genotype, allowing the PUF population to converge towards a solution that models both components. We make the following observation with respect to PUFs that employ input transformations:

✓ **O5**. Any input transformation-based PUF (like LP-PUF) converts the actual challenge C into C'. The PUF operates on the transformed tuple (C', R). CalyPSO++ 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 $C^{''} \rightarrow R$.

Using observation **O5**, CalyPSO++ makes 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.

7 Experimental results and analysis

In this section, we provide details about our experimental setup, hyperparameters used for CalyPSO and CalyPSO++ and modeling results of different PUF architectures, along with a comparison with state-of-the-art.

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	100K	∼ 1 day	96.72 %	12695	91.27 %	10538	94.35 %	11464
7-XOR	100K	∼ 1 day	97.12 %	11442	93.73 %	9583	87.37 %	10547
8-XOR	200K	∼ 1.5 days	92.61 %	24253	89.63 %	27284	94.61 %	28352
9-XOR	200K	∼ 2 days	83.12 %	24142	81.24%	21321	76.43%	19631
10-XOR	2M	∼ 2 days	84.21%	15413	81.52%	17312	82.84%	14251
11-XOR	2M	∼ 2 days	81.44 %	8376	81.85 %	7866	80.84 %	7115
12-XOR	2M	∼ 2 days	81.14 %	7827	81.17 %	7273	80.57 %	6928
13-XOR	2M	∼ 2 days	82.5 %	7414	82.5 %	6988	81.8 %	6340
14-XOR	2M	∼ 2 days	83.04 %	6962	82.54 %	6343	83.17 %	6050
15-XOR	2M	∼ 2 days	83.28 %	6608	82.80 %	5868	83.49 %	5709
16-XOR	2M	∼ 2 days	83.58 %	5282	83.69 %	4900	83.13 %	4761
17-XOR	2M	∼ 2 days	83.10 %	5968	83.23 %	5714	82.57 %	5487
18-XOR	2M	∼ 2 days	83.44 %	5690	83.08 %	5138	83.54 %	4954
19-XOR	2M	∼ 2 days	83.59 %	5423	83.75 %	5053	83.18 %	4787
20-XOR	2M	∼ 2 days	84.83 %	5205	84.60 %	4865	85.02 %	4573
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
FF-APUF (10 loops)	200K	∼ 1 day	93.13 %	9629	95.21 %	9315	91.24 %	8941

7.1 Experimental Setup and Hyperparameter tuning

Our experiments include simulations on *PyPUF* [WGM+21] (both noisy and noiseless) 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` APUF 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 5.2.1). 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 evidence, we placed the value of this parameter's value at 500 generations. That is, if the algorithm fails to find a new solution for 500 consecutive generations, we invoke landscape evolution and give new paths to the global optimum by changing (C, R). 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 256GB DRAM. 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 package. The source code for CalyPSO/CalyPSO++ and CRP dataset for hardware implementations are available at https://github.com/SEAL-IIT-KGP/calypso.

7.2 Noiseless simulations on *PyPUF*

In order to evaluate the functional capability of our attack, we first mount both versions of the attack (i.e. CalyPSO and CalyPSO++) on noiseless versions of respective PUF

Table 2: Results for cross-architectural attack. A $x \rightarrow y$ entry signifies an experiment where a PUF architecture x is modelled with architecture y. 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 → 1-XOR	50K	∼ 1 day	78.42 %	7630	73.25 %	7261	75.45 %	7541
3-XOR → 2-XOR	50K	∼ 1 day	77.72 %	12704	77.59 %	12306	77.68 %	11937
4-XOR → 3-XOR	100K	∼ 1 day	77.5 %	11452	77.47 %	10808	77.15 %	9929
5-XOR → 4-XOR	100K	∼ 1 day	79.65 %	10479	79.59 %	9870	79.53 %	9583
6-XOR → 5-XOR	100K	∼ 1 day	79.04 %	7859	78.932 %	7639	78.76 %	7239
1 LP-PUF → APUF	50K	∼ 5 hours	79.97 %	15421	80.21 %	13452	81.75 %	13773
2 LP-PUF → 2-XOR	100K	∼ 9 hours	82.74 %	2767	81.35 %	2525	84.25 %	2953
4 LP-PUF → 4-XOR	2M	∼ 3 days	66.03 %	29284	59.21 %	27361	63.74 %	31527

architectures. Table 1 summarizes our results on *PyPUF* simulations of actual PUF architectures. Likewise, Table 2 summarizes the results of cross-architectural 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. Interestingly, as the non-linearity increases (with increasing value of k-XOR), the accuracy obtained by CalyPSO in modeling the PUFs decreases in Table 1. A similar trend is observed in Table 2 as input transformations become more involved. This is because as PUF architectures become more and more complex, the modeling algorithms need to deal with increasingly expanded search space. However, one must also note that CalyPSO and CalyPSO++ are essentially randomized algorithms with random sampling of PUF instances and random decision-making undertaken at every iteration. Therefore, the accuracy achieved, CRPs required and number of generations reported in each run for any particular PUF architecture signify the random choice of parameters selected by the algorithm for that particular run. This randomization explains certain deviation contrary to this general decreasing trend in accuracy with increased complexity, like lesser achieved accuracy for 11-XOR and 12-XOR than that of higher XOR PUFs in Table 1, or better accuracy for cross-architectural attack on 2 LP-PUF than on 1 LP-PUF in Table 2.

7.3 Exploring effect of noise on CalyPSO and CalyPSO++

In order to better evaluate the performance of CalyPSO and CalyPSO++ in presence of noise, we perform a number of experiments on real-world hardware implementations (both on publicly available dataset [MTZ+20] as well as in-house constructions) and noisy simulations of various PUF architectures.

We first detail results on hardware instantiations. For 4-XOR to 9-XOR APUF variants, we use the publicly available CRP dataset from hardware implementations on Artix-7 FPGA [MTZ+20, Wis21a]. The dataset is collected for 1 million challenges for 4, 5, and 6 XOR variants and for 5 million challenges in case of 7, 8, and 9 XOR variants, with challenge length of 64 bits. In addition, we also created in-house hardware designs on FPGAs. It is worth mentioning that creating hardware designs for all PUF variants is impractical and not necessary. Moreover, since we already test our attack algorithm on publicly available hardware data for 4 to 9 XOR variants, we picked one smaller XOR variant (i.e. 4-XOR APUF) and three large XOR variants (i.e. 10, 11, and 12-XOR APUFs) for our hardware designs. This, in turn, ensures that we have tested our framework on hardware-based data for 4 to 12 XOR APUFs. We implemented the designs across *four* Nexys-4 DDR boards (Artix-7 FPGA) for 300K challenges of length 64-bits each with *five* measurements for each

Table 3: Performance evaluation of CalyPSO/CalyPSO++ against various PUF architectures in the presence of noise. A $x \to y$ entry signifies an experiment where a PUF architecture x is modelled with architecture y.

Instance	Data Source	CRPs	Uniformity	Uniqueness	Reliability	Accuracy	Generations	Algorithm
4-XOR	In-house hardware	100K	49.798 %	50.13 %	87.79 %	85.12 %	12632	CalyPSO
4-XOR	Hardware data [MTZ+20]	500K	48.5%	•	•	92.942 %	23310	CalyPSO
5-XOR	Hardware data [MTZ+20]	500K	49.84%	•	•	92.893 %	22970	CalyPSO
6-XOR	Hardware data [MTZ+20]	1M	49.8%	•	•	87.327%	23185	CalyPSO
7-XOR	Hardware data [MTZ+20]	1M	50.13%	•	•	92.655 %	22895	CalyPSO
8-XOR	Hardware data [MTZ+20]	5M	49.92%	•	•	89.41 %	20474	CalyPSO
9-XOR	Hardware data [MTZ+20]	5M	50.02%	•	•	85.93 %	18489	CalyPSO
10-XOR	In-house hardware	300K	48.415%	43.69%	95.854%	82.58%	27731	CalyPSO
11-XOR	In-house hardware	300K	50.09%	50.68%	85.27%	81.52%	25784	CalyPSO
12-XOR	In-house hardware	300K	49.93%	46.94%	79.53%	77.85%	19352	CalyPSO
12-XOR	PyPUF ⊗	5M	50.05%	49.96%	78.26%	75.54%	23512	CalyPSO
13-XOR	PyPUF †	5M	49.89%	49.94%	87.21%	82.62%	34612	CalyPSO
14-XOR	PyPUF †	5M	50.01%	49.91%	86.61%	83.85%	39636	CalyPSO
15-XOR	PyPUF †	5M	49.96%	50.01%	85.882%	81.87%	34842	CalyPSO
16-XOR	PyPUF †	5M	49.97%	49.94%	85.21%	79.63%	35713	CalyPSO
17-XOR	PyPUF †	5M	49.92%	49.92%	84.78%	78.83%	37527	CalyPSO
18-XOR	PyPUF †	5M	50.05%	50.11%	83.99%	79.83%	35285	CalyPSO
19-XOR	PyPUF †	5M	50.02%	50.07%	83.57%	77.62%	35527	CalyPSO
20-XOR	PyPUF †	5M	49.92%	50.05%	83.18%	75.62%	34587	CalyPSO
20-XOR	PyPUF ⊗	5M	49.83%	50.02%	75.92%	73.84%	19157	CalyPSO
BR-PUF → 4-XOR	In-house hardware	200K	46.88 %	52.52 %	91.77 %	75.38%	36752	CalyPSO++
4 LP-PUF → 4-XOR	In-house hardware	300K	49.81 %	47.81%	80.57 %	74.29 %	36742	CalyPSO++
(11, 11) i-PUF → 11-XOR	In-house hardware	300K	46.227%	56.24%	91.8304 %	84.83 %	34168	CalyPSO++

•: Not possible to compute since the public data had *single* measurement for a *single* PUF instance.
†: The noise for the noisy simulation was drawn from a normal distribution of mean 0 and standard deviation 0.03.
⊗: The noise for the noisy simulation was drawn from a normal distribution of mean 0 and standard deviation 0.09.

challenge. Finally, we used temporal majority voting[3] to create the overall golden responses for each PUF architecture. The PUF metrics and their corresponding modeling accuracy are summarized in Table 3. Our validations on challenge-response data from these in-house implementations on FPGAs as well as with publicly available PUF datasets [MTZ+20] corroborate with the results on noiseless simulations (c.f. Table 1).

For higher XOR PUFs ($\geq$ 13-XOR variants), we note that these designs show a marked decrease in reliability ($< 75\%$) when instantiated on hardware. Hence, for $\geq$ 13-XOR variants, we tested CalyPSO against noisy simulations. *PyPUF* simulates noise by incorporating an additional Gaussian variable $\mathcal{N}(0, \sigma)$ of mean 0 and user-defined standard deviation σ. However, the choice of this *user-defined* standard deviation must be such that the simulation behaves functionally as close to the hardware as possible. To explore this further, we instantiated over 1000 simulated PUF instances of each architecture mentioned in Table 4, and compared the standard deviation for the noise for which the software instances produced almost similar distributions of responses as the hardware. Concretely, we varied the standard deviation of simulation noise from 0.01 to 0.15 in steps of 0.01 and instantiated 1000 PUF simulations for each standard deviation. In Table 4, we then noted the observed noise distribution for each XOR PUF architecture which gave response distribution as close as possible to the response distribution observed in hardware. We note that the software simulations were run against the same set of challenges for which the hardware was instantiated. The golden response set from software simulations was computed by temporal majority voting over 15 measurements. From our experiments, we observed the simulation noise's standard deviation to be in range 0.3-0.9. Hence, in our simulations of XOR PUF variants for which we do not have hardware implementation (i.e. $\geq$ 13-XOR), we ran the simulations (from 13-XOR to 20-XOR) with noise standard deviation set to 0.03. We also chose two variants (one *medium* 12-XOR and one *high* 20-XOR) for which we also attack simulations with higher end (i.e. 0.9) of the noisy spectrum reported in Table 4. It is important to note that the randomized nature of CalyPSO/CalyPSO++ prevents any direct correlation between achieved accuracies for a k-XOR and a $(k+1)$-XOR PUF (similar to noiseless simulations in Sec. 7.2). For instance, in Table 3, the accuracy

[3] A single challenge is repeated for multiple measurements on the same board, and the response of the PUF (named *golden* response) is given as the majority of responses obtained over all measurements.

Table 4: Exploration of simulation noise for different XOR variants which produces almost similar distribution of responses as the hardware. Here, $\mathbf{h}$ and $\mathbf{s}$ represent the biases of responses from the hardware and the software simulations respectively, while $\#\mathbf{C}$ represents the number of challenges [MTZ+20] for which software simulations were run. The equation ($|\ \mathbf{h}-\mathbf{s}\ | \times \#\mathbf{C}$) quantifies the difference in response distribution between the software and the hardware for a certain standard deviation of simulation noise.

Instance	Noise standard deviation	Bias (Hardware data) : $\mathbf{h}$	Bias (Software simulation) : $\mathbf{s}$	$\#C$	$\lvert\ \mathbf{h}-\mathbf{s}\ \rvert \times \#C$
4-XOR	0.08	0.03	0.02	100000	999
5-XOR	0.09	0.003164	0.003184	100000	2
6-XOR	0.03	0.002576	0.002568	100000	1
7-XOR	0.03	-0.002716	-0.00276	100000	4
8-XOR	0.06	0.001432	0.001464	100000	3
9-XOR	0.04	-0.000456	-0.000436	100000	2

trends for simulations of 12-XOR and 13-XOR PUFs cannot be directly compared because of their different noise levels. For 12-XOR PUF, we chose noise level 0.09 that gave approximately the same reliability as 12-XOR hardware, while 13-XOR PUF was simulated with gentler noise level 0.03 (c.f. Table 4). Moreover, we note that in experiments on either noisy simulations or hardware data, reliability plays an essential role in determining convergence. By definition, reliability refers to the % of times, the PUF responses are reproducible for an identical challenge over time, under varying operating conditions. More specifically, the reliability value provides the theoretical upper limit achievable by any modeling approach for any given PUF. Therefore, we consider a PUF to be successfully modeled once the accuracy metric reported by the modeling algorithm reaches close to the target PUF's reliability value.

Finally, to evaluate the resilience of CalyPSO++ in presence of noise, we implemented three different kinds of PUF architectures on hardware: BR-PUF, 4 LP-PUF, and $(11, 11)$-i-PUF. Table 3 shows cross-architectural attacks of CalyPSO++ on these architectures. We implemented each design onto *three* Nexys DDR 4 boards for 300K CRPs and performed temporal majority voting upon *five* measurements for each challenge. We note that for BR-PUF, *PyPUF* mandates the passing of predetermined weights because of its inability to represent physical intrinsics of bistable rings. In other words, BR-PUF does not have a functional mathematical model which can be simulated accurately in *PyPUF*. However, as noted in Table 3, CalyPSO++ is still able to show significant cross-architectural learning capability, thereby showing its attack potency. Furthermore, we also note that this is the first attack in literature on a hardware implementation of LP-PUF. From an adversarial point of view, the success of CalyPSO++ in modeling hardware instantiations of these PUF variants notes the promise of cross-architectural attacks on strong PUFs.

7.4 A note on rate of convergence of CalyPSO

CalyPSO/CalyPSO++ are randomized algorithms in the sense that the initialization of the population pool and the generational mutations are driven by random decisions. As such, for the same target PUF instance, two separate runs of CalyPSO are expected to converge differently (i.e. differ in the number of generations CalyPSO takes to model the target PUF). However, the fact that the mutation strategy is round-robin ensures that every arbiter stage of every member of the population gets at least one chance of evolution in number of generations upper bounded by the total number of stages. This places a tight upper bound on the likelihood of evolution of one delay stage. For instance, in a k-XOR PUF working on n-bit challenges, each delay stage evolves once in at most $(k \times n)$ generations. Moreover, the rate of convergence is not constant across all generations. Initially, *exploration* phase (c.f. Sec. 5) dominates, and the algorithm takes more abrupt random steps, thereby showing a higher rate of convergence. However, as the population matures, *exploitation* phase (c.f. Sec. 5) starts to dominate. Finally, the algorithm converges as the *exploitation* phase saturates when the accuracy achieved is near the value of reliability of the target PUF.

Table 5: Comparison Table for modeling accuracy across several PUF designs

PUF architecture		State of the Art Works			Our Work (simulation)		Our Work (Hardware)	
		Accuracy (%)	**CRPs** (×1000)	**Additional Information**	**Accuracy** (%)	**CRPs** (×1000)	**Accuracy** (%)	**CRPs** (×1000)
APUF		98.3 [Bec15a]	20	Reliability			-	-
		99 [RSS+13]	2.5	NA	99.36	5		
XOR-APUF	k=2	99.3 [SBC19]	32	NA	98.05	30	-	-
	k=3	99.2 [SBC19]	36.8	NA	98.38	30	-	-
	k=4	94.6 [Bec15a]	150	Reliability	96.37	100	95.12	500
		99 [RSS+13]	12	NA				
		98.27 [SC20]	40	NA				
		95 [MRMK13]	40	Power side channel				
	k=5	99 [RSS+13]	80	NA	99.42	100	92.893	500
		98.09 [SC20]	80	NA				
		95 [MRMK13]	80	Power side channel				
	k=6	84 [TAB21]	40	Reliability	96.72	100	87.33	1000
		99 [RSS+13]	200	NA				
		97.39 [SC20]	320	NA				
		95 [MRMK13]	200	Power side channel				
	k=7	99 [RSS+13]	500	NA	97.12	100	92.65	1000
		97.78 [SC20]	560	NA				
		95 [MRMK13]	500	Power side channel				
	k=8	89 [Bec15a]	300	Reliability	94.61	200	89.41	5000
		99 [FKMK22]	2000	NA				
		97.42 [SC20]	2700	NA				
		98.5 [RXS+14]	26	Timing side channel				
		98.1 [RXS+14]	26	Power side channel				
	k=9	98.1 [WTM+22]	45000	NA	83.12	200	85.95	5000
	k=10	89 [TAB21]	200	Reliability	84.21	2000	82.58	300
		97.9 [WTM+22]	119000	NA				
	k=11	98.1 [WTM+22]	325000	NA	81.85	2000	81.52	300
	k=12	98.1 [RXS+14]	39	Timing side channel	81.17	2000	77.85	300
		98.3 [RXS+14]	39	Power side channel				
	k=13	NR		NA	82.5	2000	82.62	5000
	k=14	NR		NA	83.17	2000	83.85	5000
	k=15	NR		NA	83.49	2000	81.87	5000
	k=16	80.2 [Bec15a]	500	Reliability	83.69	2000	79.63	5000
		98 [RXS+14]	52	Timing side channel				
		98 [RXS+14]	52	Power side channel				
	k=17	NR		NA	83.23	2000	78.83	5000
	k=18	NR		NA	83.54	2000	79.83	5000
	k=19	NR		NA	83.75	2000	77.62	5000
	k=20	NR		NA	85.02	2000	75.62	5000
***i*-PUF**	(3-3)	98 [TAB21]	300	NA	97.21	500	-	-
LP-PUF	k=1	NR		NA	98.24	50	-	-
	k=2	80 [Wis21b]	500	NA	84.25*	100	-	-
	k=4	50 [Wis21b]	50000	NA	66.03*	2000	74.29*	200
FF-APUF (10 loops)	k=1	93 [WTM+22]	630	NA	95.21	200	-	-
BR-PUF	k=1	98.32 [GTFS16]	1	NA	†	†	75.38*	200

* Accuracy values are reported with CalyPSO + +; NR: Not Reported; NA: Not Applicable (works use ML model)
† Does not have an accurate simulation in *PyPUF*.

7.5 Comparison with State-of-the-art attacks

In Table 5, we compare the PUF modeling accuracy using CalyPSO and CalyPSO++ along with the required number of training challenge-response pairs (CRPs) with state-of-the art approaches. It can be seen that CalyPSO requires a lower number of CRPs to perform a successful attack in contrast to attacks requiring additional information and neural network (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 APUFs with $k > 16$, 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 [TAB21, Bec15b]. One must note that we do not use the PUF reliability information in our attacks and yet achieve a high modeling accuracy. Lastly, in regard to the much coveted LP-PUF [4] construction that claims to have high security [Wis21b], we see that our proposed cross-architectural attack strategy obtains an accuracy of 98.24%, 84.25% and 66.03% for 1, 2 and 4 LP-PUF construction respectively (in a noiseless setting). This is due to the fact that our attack strategy successfully nullifies the impact of input

[4] LP-PUFs are secure against the recently proposed cryptanalytic attack strategies [KMP+22] due to the hardware derived randomness induced in the challenge transformation.

transformation in the case of LP-PUFs and thereby achieves better than random prediction for LP-PUFs. Furthermore, one can also see in Table 5, that a FF-APUF with 10 loops can be easily modelled with much lesser CRPs than the state-of-the-art approaches using CalyPSO. Lastly, we are also the first ones to propose successful cross-architectural modeling of BR-PUFs with an accuracy of 75.38%.

Limitations of ML approaches: One might observe that in comparison to approaches where no additional information (like reliability values or power side-channel traces) were used, our approach requires significantly less number of CRPs than traditional Machine Learning approaches. This is due to the fundamentally different approach we take as compared to ML techniques in the context of modeling PUFs. For instance, increasing non-linearity causes ML algorithms to struggle at separating hyperplanes. As such, the modeling performed by ML becomes difficult (without additional information like reliability or side-channel traces) with increasing values of k. Empirically, it is observed in the literature (cf. Table 5) that state-of-the-art ML techniques struggle to model PUFs beyond $k = 12$. We attribute this observation to the fundamental attack principle of ML (i.e. separable hyperplanes), wherein for higher k values, the hyperplane becomes too convoluted to be linearly separable. Briefly, these results reinforce the fact that by *not* relying on the separation of convoluted decisional hyperplanes, our evolutionary-based approach is able to model various PUF architectures without the need for additional (side channel or reliability) information.

8 Conclusion

This work proposes an alternative approach for modeling delay-based PUFs by developing a novel evolutionary algorithm named CalyPSO instead of using machine learning. CalyPSO successfully modelled k-XOR APUFs (with k as high as 20), as well as LP-PUF instances where no prior attacks have been reported (on both noiseless and noisy versions). We also propose CalyPSO++ to mount novel cross-architectural modeling attacks on PUFs. Concretely, we ① reduce the security of a k-XOR APUF to a $(k-1)$-XOR APUF, and ② successfully model PUFs relying on input transformations for security. To the best of our knowledge, this work is the first of its kind to propose a new class of cross-architectural modeling attacks on delay-based PUFs. The novel attack vectors introduced in this work raise the question: *How should we design the next generation delay PUFs?* One way would be to study the information acquisition done in the evolutionary learning process and develop PUF instances for which the fitness remains lesser than a threshold [FM93]. This work motivates the search for PUF compositions which are not closed in its set. Concretely, when two PUFs are composed together, the resultant PUF would never realize Boolean mappings which belong to the set of Boolean functions realized by the individual PUFs (an idea already explored in context of block ciphers [PHS13, DR02]). Borrowing such ideas while defining PUF compositions could be an exciting future direction of research.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions for improving the paper. They would also like to thank the Department of Science and Technology (DST), Govt of India, IHUB NTIHAC Foundation, C3i Building, Indian Institute of Technology Kanpur, and Centre on Hardware-Security Entrepreneurship Research and Development, Meity, India, for partially funding this work.

References

[Bec15a] Georg T Becker. The gap between promise and reality: On the insecurity of xor arbiter pufs. In *Cryptographic Hardware and Embedded Systems–CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings 17*, pages 535–555. Springer, 2015.

[Bec15b] Georg T Becker. The gap between promise and reality: On the insecurity of XOR arbiter PUFs. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 535–555. Springer, 2015.

[BHB19] Jakub Breier, Xiaolu Hou, and Shivam Bhasin. *Automated Methods in Cryptographic Fault Analysis*. Springer, 2019.

[BHP11] Christoph Böhm, Maximilian Hofer, and Wolfgang Pribyl. A microcontroller sram-puf. In *2011 5th International Conference on Network and System Security*, pages 269–273, 2011.

[BK14] Georg T Becker and Raghavan Kumar. Active and passive side-channel attacks on delay based puf designs. *Cryptology ePrint Archive*, 2014.

[BS93] Thomas Bäck and Hans-Paul Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary computation*, 1(1):1–23, 1993.

[CAF20] Sujit Rokka Chhetri and Mohammad Abdullah Al Faruque. *Data-Driven Modeling of Cyber-Physical Systems using Side-Channel Analysis*. Springer Nature, 2020.

[CCL+11] Qingqing Chen, György Csaba, Paolo Lugli, Ulf Schlichtmann, and Ulrich Rührmair. The bistable ring puf: A new architecture for strong physical unclonable functions. In *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 134–141. IEEE, 2011.

[CLRS22] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[DR02] Joan Daemen and Vincent Rijmen. *The design of Rijndael*, volume 2. Springer, 2002.

[DV13] Jeroen Delvaux and Ingrid Verbauwhede. Side channel modeling attacks on 65nm arbiter pufs exploiting cmos device noise. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 137–142. IEEE, 2013.

[FKMK22] Sina Soleimani Fard, Masoud Kaveh, Mohammad Reza Mosavi, and Seok-Bum Ko. An efficient modeling attack for breaking the security of xor-arbiter pufs by using the fully connected and long-short term memory. *Microprocessors and Microsystems*, 94:104667, 2022.

[FM93] Stephanie Forrest and Melanie Mitchell. What makes a problem hard for a genetic algorithm? some anomalous results and their explanation. *Machine Learning*, 13(2):285–319, 1993.

[For96] Stephanie Forrest. Genetic algorithms. *ACM Computing Surveys (CSUR)*, 28(1):77–80, 1996.

[GCvDD02] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. Controlled physical random functions. In *18th Annual Computer Security Applications Conference, 2002. Proceedings.*, pages 149–160, 2002.

[GLC+04] Blaise Gassend, Daihyun Lim, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Identification and authentication of integrated circuits. *Concurrency and Computation: Practice and Experience*, 16(11):1077–1098, 2004.

[GTFS16] Fatemeh Ganji, Shahin Tajik, Fabian Fäßler, and Jean-Pierre Seifert. Strong machine learning attack against pufs with no mathematical model. In *Cryptographic Hardware and Embedded Systems–CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings 18*, pages 391–411. Springer, 2016.

[HA06] Hector Hung and Vladislav Adzic. Monte carlo simulation of device variations and mismatch in analog integrated circuits. *Proc. NCUR 2006*, pages 1–8, 2006.

[Hil04] Kenneth J Hillers. Crossover interference. *Current Biology*, 14(24):R1036–R1037, 2004.

[HYZ23] Kathryn Hinkelman, Yizhi Yang, and Wangda Zuo. Design methodologies and engineering applications for ecosystem biomimicry: An interdisciplinary review spanning cyber, physical, and cyber-physical systems. *Bioinspiration & Biomimetics*, 2023.

[Jon95] Terry Jones. Evolutionary algorithms, fitness landscapes and search. *PhD Thesis*, 1995.

[KB15] Raghavan Kumar and Wayne Burleson. Side-channel assisted modeling attacks on feed-forward arbiter pufs using silicon data. In *International Workshop on Radio Frequency Identification: Security and Privacy Issues*, pages 53–67. Springer, 2015.

[KMP+22] Liliya Kraleva, Mohammad Mahzoun, Raluca Posteuca, Dilara Toprakhisar, Tomer Ashur, and Ingrid Verbauwhede. Cryptanalysis of strong physically unclonable functions. *IEEE Open Journal of the Solid-State Circuits Society*, 2022.

[MCMS10] Abhranil Maiti, Jeff Casarona, Luke McHale, and Patrick Schaumont. A large scale characterization of ro-puf. In *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 94–99. IEEE, 2010.

[MKD10] Mehrdad Majzoobi, Farinaz Koushanfar, and Srinivas Devadas. Fpga puf using programmable delay lines. In *2010 IEEE International Workshop on Information Forensics and Security*, pages 1–6, 2010.

[MKP08] Mehrdad Majzoobi, Farinaz Koushanfar, and Miodrag Potkonjak. Lightweight secure pufs. In *2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 670–673, 2008.

[MRMK13] Ahmed Mahmoud, Ulrich Rührmair, Mehrdad Majzoobi, and Farinaz Koushanfar. Combined modeling and side channel attacks on strong pufs. *Cryptology ePrint Archive*, 2013.

[MTZ+20] Khalid T Mursi, Bipana Thapaliya, Yu Zhuang, Ahmad O Aseeri, and Mohammed Saeed Alkatheiri. A fast deep learning method for security vulnerability study of xor pufs. *Electronics*, 9(10):1715, 2020.

[NSJ+18] Phuong Ha Nguyen, Durga Prasad Sahoo, Chenglu Jin, Kaleel Mahmood, Ulrich Rührmair, and Marten van Dijk. The interpose puf: Secure puf design against state-of-the-art machine learning attacks. *Cryptology ePrint Archive*, 2018.

[Pai11] Kyungrock Paik. Optimization approach for 4-d natural landscape evolution. *IEEE transactions on evolutionary computation*, 15(5):684–691, 2011.

[PBH17] Alain Pétrowski and Sana Ben-Hamida. *Evolutionary algorithms*. John Wiley & Sons, 2017.

[PCA+22] Kuheli Pratihar, Urbi Chatterjee, Manaar Alam, Rajat Subhra Chakraborty, and Debdeep Mukhopadhyay. Birds of the same feather flock together: A dual-mode circuit candidate for strong puf-trng functionalities. *IEEE Transactions on Computers*, pages 1–14, 2022.

[PHS13] Josef Pieprzyk, Thomas Hardjono, and Jennifer Seberry. *Fundamentals of computer security*. Springer Science & Business Media, 2013.

[PKB07] Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization. *Swarm intelligence*, 1(1):33–57, 2007.

[RSS+10] Ulrich Rührmair, Frank Sehnke, Jan Sölter, Gideon Dror, Srinivas Devadas, and Jürgen Schmidhuber. Modeling attacks on physical unclonable functions. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 237–249, 2010.

[RSS+13] Ulrich Rührmair, Jan Sölter, Frank Sehnke, Xiaolin Xu, Ahmed Mahmoud, Vera Stoyanova, Gideon Dror, Jürgen Schmidhuber, Wayne Burleson, and Srinivas Devadas. Puf modeling attacks on simulated and silicon data. *IEEE transactions on information forensics and security*, 8(11):1876–1891, 2013.

[RXS+14] Ulrich Rührmair, Xiaolin Xu, Jan Sölter, Ahmed Mahmoud, Mehrdad Majzoobi, Farinaz Koushanfar, and Wayne Burleson. Efficient power and timing side channels for physical unclonable functions. In *Cryptographic Hardware and Embedded Systems–CHES 2014: 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings 16*, pages 476–492. Springer, 2014.

[RYV+17] Vladimir Rožić, Bohan Yang, Jo Vliegen, Nele Mentens, and Ingrid Verbauwhede. The Monte Carlo PUF. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6, 2017.

[SAS+19] Nimesh Shah, Manaar Alam, Durga Prasad Sahoo, Debdeep Mukhopadhyay, and Arindam Basu. A 0.16 pj/bit recurrent neural network based puf for enhanced machine learning attack resistance. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pages 627–632, 2019.

[SBC19] Pranesh Santikellur, Aritra Bhattacharyay, and Rajat Subhra Chakraborty. Deep learning based model building attacks on arbiter puf compositions. *Cryptology ePrint Archive*, 2019.

[SC20] Pranesh Santikellur and Rajat Subhra Chakraborty. A computationally efficient tensor regression network-based modeling attack on xor arbiter puf and its variants. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(6):1197–1206, 2020.

[SD07] G Edward Suh and Srinivas Devadas. Physical unclonable functions for device authentication and secret key generation. In *2007 44th ACM/IEEE Design Automation Conference*, pages 9–14. IEEE, 2007.

[SLZ19] Junye Shi, Yang Lu, and Jiliang Zhang. Approximation attacks on strong pufs. *IEEE transactions on computer-aided design of integrated circuits and systems*, 39(10):2138–2151, 2019.

[TAB21] Johannes Tobisch, Anita Aghaie, and Georg T Becker. Combining Optimization Objectives: New Modeling Attacks on Strong PUFs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 357–389, 2021.

[VPPK16] Arunkumar Vijayakumar, Vinay C Patil, Charles B Prado, and Sandip Kundu. Machine learning resistant strong puf: Possible or a pipe dream? In *2016 IEEE international symposium on hardware oriented security and trust (HOST)*, pages 19–24. IEEE, 2016.

[WGM+17] Nils Wisiol, Christoph Graebnitz, Marian Margraf, Manuel Oswald, Tudor AA Soroceanu, and Benjamin Zengin. Why attackers lose: Design and security analysis of arbitrarily large xor arbiter pufs. *Cryptology ePrint Archive*, 2017.

[WGM+21] Nils Wisiol, Christoph Gräbnitz, Christopher Mühl, Benjamin Zengin, Tudor Soroceanu, Niklas Pirnay, Khalid T. Mursi, and Adomas Baliuka. pypuf: Cryptanalysis of Physically Unclonable Functions, 2021.

[Whi01] Darrell Whitley. An overview of evolutionary algorithms: practical issues and common pitfalls. *Information and software technology*, 43(14):817–831, 2001.

[Wis21a] Nils Wisiol. pypuf data, August 2021. https://doi.org/10.5281/zenodo.5221305.

[Wis21b] Nils Wisiol. Towards attack resilient arbiter puf-based strong pufs. *Cryptology ePrint Archive*, 2021.

[WMP+20] Nils Wisiol, Christopher Mühl, Niklas Pirnay, Phuong Ha Nguyen, Marian Margraf, Jean-Pierre Seifert, Marten van Dijk, and Ulrich Rührmair. Splitting the interpose puf: A novel modeling attack strategy. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 97–120, 2020.

[WTM+22] Nils Wisiol, Bipana Thapaliya, Khalid T Mursi, Jean-Pierre Seifert, and Yu Zhuang. Neural network modeling attacks on arbiter-puf-based designs. *IEEE Transactions on Information Forensics and Security*, 17:2719–2731, 2022.

[XRF+14] Kan Xiao, Md Tauhidur Rahman, Domenic Forte, Yu Huang, Mei Su, and Mohammad Tehranipoor. Bit selection algorithm suitable for high-volume production of sram-puf. In *2014 IEEE international symposium on hardware-oriented security and trust (HOST)*, pages 101–106. IEEE, 2014.

[XRHB15] Xiaolin Xu, Ulrich Rührmair, Daniel E Holcomb, and Wayne Burleson. Security evaluation and enhancement of bistable ring pufs. In *International Workshop on Radio Frequency Identification: Security and Privacy Issues*, pages 3–16. Springer, 2015.

[YG10] Xinjie Yu and Mitsuo Gen. *Introduction to evolutionary algorithms*. Springer Science & Business Media, 2010.