pypuf

Release 3.2.1

Nils Wisiol

May 29, 2023

SIMULATION

pypuf is a toolbox for simulation, testing, and attacking Physically Unclonable Functions. Some functionality is only implemented in the archived version of pypuf v1.

CHAPTER

ONE

PUF SIMULATIONS IN PYPUF

Modelling of PUFs plays a central role in their analysis. Without a theoretical model, we can make no predictions on the behavior of PUFs and hence cannot study its security. On the other hand, often a model can be used to simulate PUF behavior to facilitate the analysis and avoid expensive studies with hardware.

1.1 Included PUF Desings

pypuf currently features simulation of the following strong PUFs:

1. *Arbiter PUF-based designs* utilizing the *additive delay model*, including simulations of the Arbiter PUF [GCvDD02], XOR Arbiter PUF [SD07], Lightweight Secure PUF [MKP08], Permutation PUF [WBMS19], and Interpose PUF [NSJM19].
2. *Feed-Forward Arbiter PUFs* and XORs thereof [GLCDD04].
3. *PUF designs based on bistable rings* [CCLSR11] [XRHB15].
4. *Integrated Optical PUFs* [RHUWDFJ13].

1.2 Technicalities

Simulations of PUFs in pypuf are build for performance and hence *use batch evaluation* of challenges: given a list of challenges, a list of responses will be returned.

CHAPTER

TWO

ARBITER PUFS AND COMPOSITIONS

Simulation of Arbiter PUFs and compositions of (several) Arbiter PUFs, such as in the XOR Arbiter PUF [SD07] or Interpose PUF [NSJM19] as well as designs that use *input transformations* (e.g., Lightweight Secure PUF [MKP08] are simulated in pypuf using the *additive delay model*.

Note: pypuf uses $\{-1, 1\}$ to represent bit values in Arbiter PUF simulations, for both challenges and responses.

Note: All simulations based on the additive delay model in pypuf use `numpy.int8` to represent bit values, i.e. for each challenge or response bit, one byte of memory is allocated. While inefficient in terms of required memory, this provides faster evaluation performance than storing one logical bit in one bit of memory.

2.1 Arbiter PUF

The Arbiter PUF appeared first in an article proposing a electrical Strong PUF that can be implemented cheaply using the regular CMOS manufacturing process [GCvDD02]. It can be easily modeled by several machine learning attacks.

pypuf supports simulation of Arbiter PUFs with and without noise. To instantiate a random Arbiter PUF instance, the challenge length `n` must be chosen and a `seed` for reproducible results must be provided. To evaluate challenges, provide a *list* of challenges to the `eval` method. For example, to evaluate an Arbiter PUF on three randomly chosen challenges:

```
>>> from pypuf.simulation import ArbiterPUF
>>> puf = ArbiterPUF(n=64, seed=1)
>>> from pypuf.io import random_inputs
>>> puf.eval(random_inputs(n=64, N=3, seed=2))
array([ 1,  1, -1], dtype=int8)
```

For noisy simulations, a noise level needs to be given. Basically, the higher the number, the more likely responses are to be disturbed by noise; however the influence of noise also depends on the challenge given and the concrete instance. For details, please refer to the *noise model used in the additive delay model*.

The noise level is specified as the *noisiness* parameter, `.1` is a realistic value; `1.0` will result in purely random responses. `0.0` will disable noise simulation, which is the default.

```
>>> from pypuf.simulation import ArbiterPUF
>>> puf = ArbiterPUF(n=64, seed=1, noisiness=.2)
>>> from pypuf.io import random_inputs
>>> puf.eval(random_inputs(n=64, N=3, seed=2))
```

(continues on next page)

(continued from previous page)

```
array([-1, -1,  1], dtype=int8)
>>> puf.eval(random_inputs(n=64, N=3, seed=2))
array([-1, -1, -1], dtype=int8)
```

In above example, note that the response to the *same* third challenge evaluates to a different response due to noise. This is reproducible with new instantiations using the same seed.

2.2 XOR Arbiter PUF

The XOR Arbiter PUF was introduced as mitigation for the vulnerability of the Arbiter PUF to machine learning attacks [SD07]. It consists of `k` Arbiter PUFs with all receive the same input, the response is defined as the parity (XOR) of the individual responses. The security of XOR Arbiter PUF was first studied by Rührmair et al. [RSSD10], who successfully attacked XOR Arbiter PUFs using the responses of random challenges. Their attack scales polynomially in the challenge length and exponentially in the number of parallel arbiter chains. High-performance results of this attack were studied by Tobisch and Becker [TB15]. The must successful attack on XOR Arbiter PUFs is based on the noise level of challenges, as measured by the attacker [Bec15].

To simulate an 8-XOR 64-bit XOR Arbiter PUF using relatively stable Arbiter PUF implementations, use

```
>>> from pypuf.simulation import XORArbiterPUF
>>> puf = XORArbiterPUF(n=64, k=8, seed=1, noisiness=.05)
>>> from pypuf.io import random_inputs
>>> puf.eval(random_inputs(n=64, N=3, seed=2))
array([-1, -1,  1], dtype=int8)
```

Note: The *noisiness* parameter in the XOR Arbiter PUF is directly referring to the noise of *each* Arbiter PUF in the simulation. This means that for equal *noisiness*, noise level will increase for higher *k*.

2.3 Feed-Forward (XOR) Arbiter PUF

Feed-Forward Arbiter PUFs are an attempt to increase resistance to modeling attacks [GLCDD04] compared to traditional Arbiter PUFs. In an Feed-Forward XOR Arbiter PUF, many Feed-Forward Arbiter PUFs are evaluated in parallel, and the XOR of the individual response bits is returned. The feed-forward loops may be homogeneous, i.e. the same for all participating Arbiter PUFs, or inhomogeneous.

Even if using the noisy simulation, all involved arbiter elements are assumed to work entirely noise-free with infinite precision.

To simulate an 4-XOR 128-bit Feed Forward Arbiter PUF in which each Feed-Forward Arbiter PUF will have a feed-forward loop after the 33nd stage that will determine the challenge bit to the 69th stage, use

```
>>> from pypuf.simulation import XORFeedForwardArbiterPUF
>>> puf = XORFeedForwardArbiterPUF(n=128, k=4, ff=[(32, 68)], seed=1)
>>> from pypuf.io import random_inputs
>>> puf.eval(random_inputs(n=128, N=6, seed=2))
array([-1, -1,  1,  1, -1,  1])
```

The full Feed-Forward Arbiter PUF simulation API is displayed below.

`FeedForwardArbiterPUF.__init__`(*n: int*, *ff: List[Tuple[int, int]]*, *seed: Optional[int] = None*, *noisiness: float = 0*) → None

Initialize a Feed-Forward Arbiter PUF Simulation.

Parameters

- **n** – Number of challenge bits.
- **ff** – List of forward connections in the Feed-Forward Arbiter PUFs. Forward connections are defined by two-tuples (i, j), where i defines the *arbiter position*, and j defines the *feed position* of the feed forward loop. For each feed forward loop given, an arbiter element is simulated after the i-th stage, with the result inserted as the challenge bit to the j-th stage. Note that the Feed-Forward Arbiter PUF has *n + len(ff)* stages. It is required that $i < j$, and no feed position may appear more than once per Arbiter PUF, however, it is allowed to use the arbiter position multiple times.
- **seed** – Seed for random weight generation.
- **noisiness** – Noise-level of the simulation.

`XORFeedForwardArbiterPUF.__init__`(*n: int*, *k: int*, *ff: Union[List[List[Tuple[int, int]]], List[Tuple[int, int]]]*, *seed: Optional[int] = None*, *noisiness: float = 0*) → None

Initialize an XOR Feed-Forward Arbiter PUF Simulation.

Parameters

- **n** – Number of challenge bits.
- **k** – Number of Feed-Forward PUFs in this XOR Feed-Forward Arbiter PUF.
- **ff** – List of `k` lists of forward connections in the Feed-Forward Arbiter PUF. The l-th Arbiter PUF in the XOR Feed-Forward Arbiter PUF will use the l-th list of forward connections. If a list of tuples is given instead, it is used for all `k` Feed-Forward Arbiter PUFs. See *`pypuf.simulation.FeedForwardArbiterPUF.__init__()`* for how to define feed-forward connections.
- **seed** – Seed for random weight generation.
- **noisiness** – Noise-level of the simulation.

2.4 Lightweight Secure PUF

The Lightweight Secure PUF [MKP08] was introduced to mitigate the vulnerability of the Arbiter PUF to machine learning attacks, and is the first PUF that uses *different* challenges to each arbiter chain, all generated from a *master challenge*. The Lightweight Secure PUF makes naive attacks harder [RSSD10], however does not increase overall attack resilience [WBMS19].

Todo: Add information on how the input transformation is defined and implemented.

To simulate an 8-XOR 64-bit XOR Arbiter PUF using relatively stable Arbiter PUF implementations, use

```
>>> from pypuf.simulation import LightweightSecurePUF
>>> puf = LightweightSecurePUF(n=64, k=8, seed=1, noisiness=.05)
>>> from pypuf.io import random_inputs
>>> puf.eval(random_inputs(n=64, N=3, seed=2))
array([-1, -1, -1], dtype=int8)
```

2.5 Permutation PUF

The Permutation PUF is an iteration of the idea behind the Lightweight Secure PUF, which is to feed different challenges to the arbiter chains in an XOR Arbiter PUF. After it was shown that the specific way the Lightweight Secure PUF modifies the individual challenges, the Permutation PUF was introduced to simplify implementation and remove the attack surface the Lightweight Secure PUF introduced [WBMS19].

To generate the individual challenges, the Permutation PUF applies a predetermined set of *k* permutations of the *master* challenge, one for each individual challenge. The permutations are chosen in a way such that no two permutations permute a bit the same way, i.e. from the same position to the same position, and additionally such that no permutation has a fix point.

To simulate an 8-XOR 64-bit XOR Arbiter PUF using relatively stable Arbiter PUF implementations, use

```
>>> from pypuf.simulation import PermutationPUF
>>> puf = PermutationPUF(n=64, k=8, seed=1, noisiness=.05)
>>> from pypuf.io import random_inputs
>>> puf.eval(random_inputs(n=64, N=3, seed=2))
array([ 1, -1,  1], dtype=int8)
```

2.6 Interpose PUF

The Interpose PUF [NSJM19] was designed to mitigate the well-performing reliability-based attack on the XOR Arbiter PUF [Bec15]. It consists of two XOR Arbiter PUFs, called *upper* and *lower* layer. The upper layer has k_{up} parallel arbiter chains and challenge length n, the lower layer k_{down} and challenge lenght $n + 1$. To determine the response of the Interpose PUF, the challenge is input into the upper layer and evaluated. The response of the upper layer is then *interposed* in the middle of the challenge; the resulting $n + 1$ bit long challenge is then input in the lower layer. The resulting response is the final response of the PUF.

A security analysis of showed the Interpose PUF to be immune against known attacks in the literature [NSJM19]. However, the Logistic Regression attack [RSSD10], originally designed for attacking the XOR Arbiter PUF, can be modified to “split” the Interpose PUF and model it with effort only slightly above what is needed to attack XOR Arbiter PUFs of similar size [WMPN19].

To simulate an (8,8) 64-bit Interpose PUF using relatively stable Arbiter PUF implementations, use

```
>>> from pypuf.simulation import InterposePUF
>>> puf = InterposePUF(n=64, k_up=8, k_down=8, seed=1, noisiness=.05)
>>> from pypuf.io import random_inputs
>>> puf.eval(random_inputs(n=64, N=3, seed=2))
array([-1,  1, -1], dtype=int8)
```

Note that the `noisiness` parameter applies to both upper and lower layer.

CHAPTER

THREE

ADDITIVE DELAY MODEL

The Additive Delay Model [GCvDD02] can be used to model Arbiter PUFs as well as Arbiter PUF-based constructions, such as the XOR Arbiter PUF, the Lightweight Secure PUF, the Permutation PUF, and the Interpose PUF. It can be extended by a noise model based on Gaussian random variables, which has been shown to highly accurate [DV13].

This module contains `LTFArray`, a simulation implementation of the Additive Delay Model, as well as `NoisyLTFArray`, an implementation of the Additive Delay Model with Gaussian noise.

Note: pypuf's implementation of the Additive Delay Model uses $\{-1, 1\}$ to represent bit values both for challenges and responses.

Note: All simulations based on the Additive Delay Model in pypuf use `numpy.int8` to represent bit values, i.e. for each challenge or response bit, one byte of memory is allocated. While inefficient in terms of required memory, this provides faster evaluation performance than storing one logical bit in one bit of memory.

3.1 Modeling of Delay Values

Todo: Add derivation of Additive Delay Model from circuit

Implementation of the Additive Delay Model

class `pypuf.simulation.base.LTFArray`(*weight_array: ndarray*, *transform: Union[str, Callable]*, *combiner: Union[str, Callable] = 'xor'*, *bias: Optional[ndarray] = None*)

Highly optimized, numpy-based implementation that can batch-evaluate functions as found in the Additive Delay Model of Arbiter PUFs, i.e. functions f of the form

$$f : \{-1,1\}^n \to \{-1,1\}, f(c) = \text{sgn}\, \hat{f}(c),$$
$$\hat{f} : \{-1,1\}^n \to \mathbb{R}, \hat{f}(x) = \prod_{l=1}^{k} (b_l + \sum_{i=1}^{n} W_{l,i} \cdot x_i),$$

where n, k are natural numbers (positive `int`) specifying the size of the *LTFArray*, $W \in \mathbb{R}^{k \times n}$ are the *weights* of the linear threshold functions, and $b_i, i \in [k]$ are the biases of the LTFs.

For performance reasons, the evaluation interface of `LTFArray` is specialized for evaluating a list of challenges with each call, returning a list of responses of same length; this length will often be referred to as N.

Todo: Detail on weight distribution.

Two generalizations of f are supported. First, modifying the challenge input has been extensively studied in the literature [Lightweight Secure PUF, Permutation PUF]. In `LTFArray`, this is implemented as *input transformation*. Second, using a function different from XOR to combine the individual results into a final result, which is less studied and is known to `LTFArray` as *combiner function* (in reference to LFSRs). We detail on both below.

Input Transformations. An input transformation can be understood as a list of functions $\lambda_1, ..., \lambda_k$, where $\lambda_i : \{-1,1\}^n \to \{-1,1\}^n$ generates the input for the i-th LTF in the `LTFArray`. In order to use input transformations with `LTFArray`, implement a Python function that maps a list of challenges `challenges` given as `ndarray` of shape (N, n) and the number of LTFs given as a positive `int k` to a list of list of sub-challenges to the individual LTFs (`sub_challenges`), returned as `ndarray` of shape (N, k, n), where for all $i \in [N]$ and $l \in [k]$

$$\texttt{sub\_challenges}_{i,l} = \lambda_l(\texttt{challenges}_i).$$

Given such a function, `LTFArray` will evaluate

$$f : \{-1,1\}^n \to \{-1,1\}, f(c) = \text{sgn}\, \hat{f}(c),$$
$$\hat{f} : \{-1,1\}^n \to \mathbb{R}, \hat{f}(x) = \prod_{l=1}^{k} (b_l + \langle W_l, \lambda_l(x) \rangle).$$

Warning: In above definition, the λ_i are different from preprocessing functions implemented in hardware. This is due to the nature of the Arbiter PUF Additive Delay Model: in order to accurately model the behavior of an Arbiter PUF without additional challenge processing on a given hardware input $c \in \{-1,1\}^n$, the Arbiter PUF needs ot be modeled as function

$$f(c) = \text{sgn} \prod_{l=1}^{k} b_l + \sum_{i=1}^{n} W_{l,i} \cdot c_i c_{i+1} \cdots c_n,$$

i.e. $\lambda_1 = \cdots = \lambda_k$ with $\lambda_1(c)_i = c_i c_{i+1} \cdots c_n$. This function is implemented in `LTFArray` as `att`; its inverse is `att_inverse`.

Combiner Function. The combiner function replaces the product of linear thresholds in the definition of $\hat{f}$. Setting the *combiner* parameter to a *Callable my_combiner* taking a shape (N, k) array of *sub-responses* and returning an array of shape $(N,)$, will compute the function

$$f(c) = \text{sgn}\, \texttt{my\_combiner} \left(b_l + \sum_{i=1}^{n} W_{l,i} \cdot c_i c_{i+1} \cdots c_n \right).$$

To instanciate an `LTFArray`, provide the following parameters:

Parameters

- **`weight_array`** – *weights* of the LTFs in this array, *ndarray* of floats with shape (k, n)
- **`transform`** – *input transformation* as described above, given as callable function or string *s* identifying a member function of this class with name *s* or *f'transform_{s}'*.
- **`combiner`** – optional *combiner function* as described above, given as callable function or string *s* identifying a member function of this class with name *s* or *f'combiner_{s}'*, defaults to *xor*, i.e. the partity function.
- **`bias`** – optional *ndarray* of floats with shape $(k,)$, specifying the bias values b_l for $l \in [k]$, defaults to zero bias.

To create an `LTFArray` containing just one function, the majority vote function of four inputs, use

```
>>> from pypuf.simulation import LTFArray
>>> from numpy import array, ones
>>> my_puf = LTFArray(ones(shape=(1, 4)), transform='id')
>>> my_puf.eval(array([[1, 1, -1, 1]]))
array([1])
>>> my_puf.eval(array([[1, -1, -1, -1]]))
array([-1])
```

eval(*challenges: ndarray*, *block_size: Optional[int] = 1000000*) → ndarray

Evaluate the PUF on a list of given challenges.

Parameters

challenges – List of challenges to evaluate on. Challenges must be given as `ndarray` of shape (N, `challenge_length`), where N is the number of challenges to be evaluated. Evaluating many challenges at once may have performance benefits, to evaluate a single challenge, provide an `ndarray` with shape (1, `challenge_length`). In cases where `challenge_length` = 0, an empty array with shape (N, 0) needs to be provided to determine the number of responses requested.

Returns

`ndarray`, shape (N, `response_length`), listing the simulated responses to the challenges in order they were given.

3.2 Modeling of Noise

Todo: Add overview of `NoisyLTFArray`.

CHAPTER

FOUR

BISTABLE RING PUFS AND COMPOSITIONS

Simulation of all flavors of Bistable Ring PUFs in pypuf are based on a model based in linear threshold functions. This model was found to describe Bistable Rings and Twisted Bistable Rings quite well [XRHB15]. Notice that the Bistable Ring PUF simulation in pypuf is based on a model derived from successful modeling attacks [XRHB15] rather than a physically motivated model.

Warning: pypuf is currently unaware of how physical intrinsics of Bistable Rings are generated in the PUF manufacturing process, hence passing of predetermined weights is mandatory for instantiating any bistable ring PUF.

Note: pypuf uses $\{-1, 1\}$ to represent bit values in Bistable Ring PUF simulations, for both challenges and responses.

Note: All bistable ring PUF simulations in pypuf use `numpy.int8` to represent bit values, i.e. for each challenge or response bit, one byte of memory is allocated. While inefficient in terms of required memory, this provides faster evaluation performance than storing one logical bit in one bit of memory.

4.1 Bistable Ring PUF

The Bistable Ring PUF was proposed as an FPGA-friendly Strong PUF design [CCLSR11].

pypuf supports simulation of Bistable Rings only without noise, due to the lack of a noise model in the literature. To instantiate a Bistable Ring PUF instance, the challenge length `n` must be chosen and appropriate weight parameters must be provided. To evaluate challenges, provide a *list* of challenges to the `eval` method. For example, to obtain a couple of Challenge-Response-Pairs of an Bistable Ring PUF:

```
>>> from numpy.random import default_rng
>>> from pypuf.simulation import BistableRingPUF
>>> n = 64
>>> weights = default_rng(1).normal(size=(n+1))  # instead, data should be derived from␣
↪experimental setup
>>> puf = BistableRingPUF(n=64, weights=weights)
>>> from pypuf.io import ChallengeResponseSet
>>> crps = ChallengeResponseSet.from_simulation(puf, N=3, seed=1)
>>> crps.responses[:, 0, 0]  # first response bit, first query
array([ 1., -1., -1.])
```

4.2 XOR Bistable Ring PUF

After successful modeling attacks on the Bistable Ring PUF [SH14], an XORed version of the Bistable Ring PUF was suggested [XRHB15]. Much like the XOR Arbiter PUF, it is a design variant where many Bistable Ring PUFs are instantiated and the parity of the individual responses is given as the final output of the XOR Bistable Ring PUF.

To simulate an 8-XOR 64-bit Bistable Ring PUF, use

```
>>> from numpy.random import default_rng
>>> from pypuf.simulation import XORBistableRingPUF
>>> k, n = 8, 64
>>> weights = default_rng(1).normal(size=(k, n+1))  # instead, data should be derived␣
↪from experimental setup
>>> puf = XORBistableRingPUF(n=64, k=8, weights=weights)
>>> from pypuf.io import random_inputs
>>> puf.eval(random_inputs(n=64, N=4, seed=2))
array([ 1,  1,  1, -1], dtype=int8)
```

CHAPTER

FIVE

INTEGRATED OPTICAL PUFS

pypuf ships a very basic simulation of integrated optical PUFs [RHUWDFJ13].

class pypuf.simulation.**IntegratedOpticalPUF**(*n: int*, *m: int*, *seed: int*)

Very basic simulation of an integrated optical PUF [RHUWDFJ13]. This simulation is derived from the corresponding successful modeling attack [RHUWDFJ13].

To compute a response to a given challenge c, the simulation evaluates the value of

$$\left|c \cdot A \cdot e^{\phi i}\right|^2,$$

where $A \in \mathbb{R}^{n \times m}$ and $\phi \in \mathbb{R}^{n \times m}$; n is the challenge length, m the response length.

By default, the values for A are chosen mutually independent and uniformly random in $[0, 1)$; the values for ϕ are chosen mutually independent and uniformly random in $[0, 2\pi)$.

Warning: This simulation only gives a very rough, idealized idea of how integrated optical PUFs may behave. In particular, the *pypuf.metrics.uniqueness()* may be quite different from the behavior real-world implementations. Also, the behavior of response pixels is mutually independent, which differs from behavior reported in the literature [RHUWDFJ13].

__init__(*n: int*, *m: int*, *seed: int*) → None

Initializes a simulation for an Integrated Optical PUF with n challenge bits, m response bits, with randomness based on the given `seed`.

eval(*challenges: ndarray*) → ndarray

Evaluate the PUF on a list of given challenges.

Parameters

challenges – List of challenges to evaluate on. Challenges must be given as `ndarray` of shape (N, `challenge_length`), where N is the number of challenges to be evaluated. Evaluating many challenges at once may have performance benefits, to evaluate a single challenge, provide an `ndarray` with shape (1, `challenge_length`). In cases where `challenge_length` = 0, an empty array with shape (N, 0) needs to be provided to determine the number of responses requested.

Returns

`ndarray`, shape (N, `response_length`), listing the simulated responses to the challenges in order they were given.

CHAPTER

SIX

BASE CLASS FOR SIMULATIONS

6.1 Batch Evaluation

pypuf PUF simulations are build to evaluate several given challenges at once, to provide better performance. The interfaces are defined accordingly, see below.

6.2 Adding Simulations

Simulations should extend the base class `simulation.Simulation` and provide a instantiation function that takes a `seed` argument for reproducible results. Weak PUFs can be added by setting `challenge_length` to zero.

6.3 API

class `pypuf.simulation.`**`Simulation`**

A PUF simulation that can be evaluated on challenges.

property challenge_length: int

The expected challenge length of this simulation, `int`.

property response_length: int

The length of responses generated by this simulation, `int`.

`eval`(*challenges: ndarray*) $\rightarrow$ ndarray

Evaluate the PUF on a list of given challenges.

Parameters

challenges – List of challenges to evaluate on. Challenges must be given as `ndarray` of shape (N, `challenge_length`), where N is the number of challenges to be evaluated. Evaluating many challenges at once may have performance benefits, to evaluate a single challenge, provide an `ndarray` with shape (1, `challenge_length`). In cases where `challenge_length` = 0, an empty array with shape (N, 0) needs to be provided to determine the number of responses requested.

Returns

`ndarray`, shape (N, `response_length`), listing the simulated responses to the challenges in order they were given.

r_eval(*r: int*, *challenges: ndarray*) → ndarray

Evaluates the Simulation `r` times on the list of N `challenges` given and returns an array of shape (`r`, N, `self.response_length`) of all responses.

```
>>> from pypuf.simulation import XORArbiterPUF
>>> from pypuf.io import random_inputs
>>> puf = XORArbiterPUF(n=64, k=4, noisiness=.02, seed=1)
>>> responses = puf.r_eval(5, random_inputs(N=2, n=64, seed=4))
>>> responses[0, :, :]  # unstable example
array([[ 1.,  1., -1., -1., -1.]])
>>> responses[1, :, :]  # stable example
array([[1., 1., 1., 1., 1.]])
```

Note: To approximate the expected respones value, use average along the last axis:

```
>>> from numpy import average
>>> average(responses, axis=-1)
array([[-0.2],
       [ 1. ]])
```

static seed(*description: str*) → int

Helper function that turns a string into an integer that can be consumed as seed by a pseudo-random number generator (PRNG). Usage scenario: create a descriptive seed and use it to initialize the PRNG.

```
>>> from numpy.random import default_rng
>>> from pypuf.simulation import Simulation
>>> seed = 'parameter seed for my PUF instance'
>>> prng = default_rng(seed=Simulation.seed(seed))
>>> parameters = prng.normal(size=(3, 4))
>>> parameters
array([[ 1.64917478, -1.28702893,  0.17287684, -1.69475886],
       [-1.74432269,  1.59592227,  1.12435243, -0.23488442],
       [-0.74190059,  0.95516568, -2.25170753, -0.22208081]])
```

CHAPTER

SEVEN

OVERVIEW

7.1 Generating Uniform Random Challenges

`pypuf.io.random_inputs`(*n: int*, *N: int*, *seed: int*) $\rightarrow$ ndarray

Generates N uniformly random challenges of length n and returns them as a *numpy.ndarray* of shape (N, n). The randomness is based on the provided `seed`.

Note: pypuf uses $\{-1, 1\}$ to represent bit values for both challenges and responses. To convert from $\{-1, 1\}$ notation to the more traditional $\{0, 1\}$ encoding, use `x = (1 - x) // 2`. For the reverse conversion, use `x = 1 - 2*x`

```
>>> import numpy as np
>>> import pypuf.io
>>> challenges = pypuf.io.random_inputs(n=64, N=10, seed=1)
>>> np.unique(challenges)
array([-1,  1], dtype=int8)
>>> challenges01 = (1 - challenges) // 2
>>> np.unique(challenges01)
array([0, 1], dtype=int8)
>>> challenges11 = 1 - 2 * challenges01
>>> (challenges11 == challenges).all()
True
```

7.2 Storing Challenge-Response Data

pypuf stores challenge-response data in objects of the class *`pypuf.io.ChallengeResponseSet`*. These objects contain two numpy arrays with the challenges and responses, respectively and provide a number of auxiliary functions for convenient management.

To create a instance of *`pypuf.io.ChallengeResponseSet`* for given challenges and responses, use

```
>>> import numpy as np
>>> challenges = np.array([[-1, -1, -1, -1], [-1, -1, -1, 1]])
>>> responses = np.array([1, 1])
>>> import pypuf.io
>>> crp = pypuf.io.ChallengeResponseSet(challenges, responses)
```

To create an instance of *`pypuf.io.ChallengeResponseSet`* from a *`pypuf.simulation.Simulation`*, use

```
>>> import pypuf.simulation
>>> puf = pypuf.simulation.ArbiterPUF(n=64, seed=1)
>>> import pypuf.io
>>> crp = pypuf.io.ChallengeResponseSet.from_simulation(puf, N=1000, seed=2)
```

pypuf CRP data can be stored on disk and loaded back into pypuf:

```
>>> crp = pypuf.io.ChallengeResponseSet.from_simulation(puf, N=1000, seed=2)
>>> crp.save('crps.npz')
>>> crp_loaded = pypuf.io.ChallengeResponseSet.load('crps.npz')
>>> crp == crp_loaded
True
```

pypuf.io.ChallengeResponseSet can also be sliced to obtain a single challenge-response pair or to get a subset of CRPs:

```
>>> crp = pypuf.io.ChallengeResponseSet.from_simulation(puf, N=1000, seed=2)
>>> crp[0]
(array([-1,...), array([[1.]]))
>>> crp[:10]
<10 CRPs with challenge length 64 and response length 1, each response measured 1␣
↪time(s)>
```

class pypuf.io.**ChallengeResponseSet**(*challenges: ndarray*, *responses: ndarray*)

__init__(*challenges: ndarray*, *responses: ndarray*) → None

Create a challenge-response object containing the given challenges and responses of a PUF token.

Parameters

- **challenges** (*numpy.ndarray*) – Challenges to the PUF token organized as array of shape (N, n), where N is the number of challenges and n is the challenge length.
- **responses** (*numpy.ndarray*) – Responses of the PUF token organized as array of shape (N, m, r), where N is the number of challenges, m is the response length, and r is the number of measurements per challenge. The last axis is optional.

property challenge_length: int

The length n of the recorded challenges.

classmethod load(*f: str*) → object

Loads CRPs from the given file `f`.

property response_length: int

The length m of the recorded responses.

save(*f: str*) → None

Saves the CRPs to the given file `f`.

CHAPTER

EIGHT

PUF DATA SETS

pypuf includes a number of data sets which have been published in the scientific literature on PUFs. The data is not downloaded along with pypuf when it is installed, instead, it is automatically fetched on first access.

For example, pypuf includes an FPGA-implemented 64-bit 4-XOR Arbiter PUF [MTZAA20]:

```
>>> import pypuf.io
>>> pypuf.io.MTZAA20.xor_arbiter_puf_4_xor
<CRP Set available from https://zenodo.org/record/5215875/files/MTZAA20_4XOR_64bit_LUT_
↪2239B_attacking_1M.txt.npz?download=1, not fetched yet>
```

The data is automatically fetched on first use, e.g. to compute the bias of the responses:

```
>>> import pypuf.metrics
>>> pypuf.metrics.bias_data(pypuf.io.MTZAA20.xor_arbiter_puf_4_xor.responses)
array([[0.03495]])
```

As CRP sets may have large size and significnat download time, pypuf logs a warning message when fetching CRPs, which include URL and file size (if known):

```
Fetching CRPs (9.9MiB) from https://zenodo.org/record/5215875/files/MTZAA20_4XOR_64bit_
↪LUT_2239B_attacking_1M.txt.npz?download=1
```

8.1 Hybrid Boolean Network [CCPG21]

Hybrid Boolean Networks can be used as PUFs as described by Charlot et al. [CCPG21]. In pypuf, there are CRPs included for 8 different PUF instances. Each instance is measured on the same set of 1002 challenges, where each challenge is measured 100 times. The measurement time is chosen as optimal (cf. discussion in the paper).

pypuf Object	PUF Type	Implementation	Amount	Challenge Length	Response Length	Repeated Measurements
`pypuf.io.CCPG21.hbn_board_1`	Hybrid Boolean Network	FPGA	1002	256	256	100
`pypuf.io.CCPG21.hbn_board_2`	Hybrid Boolean Network	FPGA	1002	256	256	100
`pypuf.io.CCPG21.hbn_board_3`	Hybrid Boolean Network	FPGA	1002	256	256	100
`pypuf.io.CCPG21.hbn_board_4`	Hybrid Boolean Network	FPGA	1002	256	256	100
`pypuf.io.CCPG21.hbn_board_5`	Hybrid Boolean Network	FPGA	1002	256	256	100
`pypuf.io.CCPG21.hbn_board_6`	Hybrid Boolean Network	FPGA	1002	256	256	100
`pypuf.io.CCPG21.hbn_board_7`	Hybrid Boolean Network	FPGA	1002	256	256	100
`pypuf.io.CCPG21.hbn_board_8`	Hybrid Boolean Network	FPGA	1002	256	256	100

8.2 64-bit Interpose PUF [AM21]

The Interpose PUF provided consists of one 64-bit Arbiter PUF in the top layer and five 65-bit Arbiter PUFs in the bottom layer. The challenges of `pypuf.io.AM21.interpose_puf` and `pypuf.io.AM21.arbiter_puf_top` are identical; the challenges to the bottom Arbiter PUFs correspond to the original challenges interposed with the responses of the top Arbiter PUF.

The `pypuf.io.AM21.interpose_puf` CRP data is provided for convenience only, it can also be derived from the other data sets. Of course, also (1,x)-Interpose PUFs can be build for $x < 5$.

pypuf Object	PUF Type	Implementation	Amount	Challenge Length	Response Length	Repeated Measurements
`pypuf.io.AM21.arbiter_puf_top`	Arbiter PUF	FPGA	1M	64	1	1
`pypuf.io.AM21.arbiter_puf_bottom_0`	Arbiter PUF	FPGA	1M	65	1	1
`pypuf.io.AM21.arbiter_puf_bottom_1`	Arbiter PUF	FPGA	1M	65	1	1
`pypuf.io.AM21.arbiter_puf_bottom_2`	Arbiter PUF	FPGA	1M	65	1	1
`pypuf.io.AM21.arbiter_puf_bottom_3`	Arbiter PUF	FPGA	1M	65	1	1
`pypuf.io.AM21.arbiter_puf_bottom_4`	Arbiter PUF	FPGA	1M	65	1	1
`pypuf.io.AM21.interpose_puf`	(1,5)-Interpose PUF	FPGA	1M	64	1	1

8.3 64-bit XOR Arbiter PUF [MTZAA20]

pypuf Object	PUF Type	Implementation	Amount	Challenge Length	Response Length	Repeated Measurements
`pypuf.io.MTZAA20.xor_arbiter_puf_4_xor`	4-XOR Arbiter PUF	FPGA	1M	64	1	1
`pypuf.io.MTZAA20.xor_arbiter_puf_5_xor`	5-XOR Arbiter PUF	FPGA	1M	64	1	1
`pypuf.io.MTZAA20.xor_arbiter_puf_6_xor`	6-XOR Arbiter PUF	FPGA	1M	64	1	1
`pypuf.io.MTZAA20.xor_arbiter_puf_7_xor`	7-XOR Arbiter PUF	FPGA	5M	64	1	1
`pypuf.io.MTZAA20.xor_arbiter_puf_8_xor`	8-XOR Arbiter PUF	FPGA	5M	64	1	1
`pypuf.io.MTZAA20.xor_arbiter_puf_9_xor`	9-XOR Arbiter PUF	FPGA	5M	64	1	1

CHAPTER

NINE

BIAS

The response bias is one of the fundamental metrics for the response behavior of PUFs. Large response bias enable trivial modeling attacks on the PUF as the attacker can just guess the more likely of the two responses.

PUF bias is in the literature also known as *balance* or *uniformity*.

Note that per bit encoding in pypuf, a bias of *zero* means a perfectly unbiased behavior, whereas bias of -1 and 1 means that the PUF under test always returns -1 and 1, respectively. To convert the pypuf bias value b into the more traditional 0-1-scale, use $1/2 - b/2$.

`pypuf.metrics.`**`bias`**(*instance: Simulation*, *seed: int*, *N: int = 1000*) → ndarray

Approximates the bias of a given simulation by generating `N` random challenges using seed `seed` and computing the bias for each of the $m \geq 1$ response bits given by the `instance`,

$$b_l = E_x\left[f(x)_l\right],$$

where f is the function computed by `instance` and $f(x)_l, 1 \leq l \leq m$ is the l-th response bit.

Arbiter PUF simulations in pypuf per additive delay model almost unbiased:

```
>>> from pypuf.simulation import ArbiterPUF
>>> from pypuf.metrics import bias
>>> bias(ArbiterPUF(n=128, seed=42), seed=1)
-0.004
```

On the other hand, 2-XOR Arbiter PUFs can have relatively large bias [WP20].

```
>>> from pypuf.simulation import XORArbiterPUF
>>> bias(XORArbiterPUF(n=64, k=2, seed=2), seed=2)
-0.086
```

`pypuf.metrics.`**`bias_data`**(*responses: ndarray*) → ndarray

Given an arrays of responses of shape (N, m), returns the m bias values of the response bits,

$$b_l = E_x\left[f(x)_l\right],$$

where f is the function given by `responses` and $f(x)_l, 1 \leq l \leq m$ is the l-th response bit.

If response length m is greater than 1, the bias is given for each bit seperately. To obtain the general bias, average the response.

Returns an array of shape $(m,)$.

CHAPTER

TEN

RELIABILITY

Reliability is a measure for the noise in challenge evaluation of a single PUF instance. Most PUF implementations have challenge evaluation mechanisms influenced by noise, hence the question of reproducible responses arises. In the literature, the concept of reliability is also referred to as *intra-distance*, *stability* and *reproducibility*. pypuf defines the reliability on challenge x of a PUF with responses $\texttt{eval}(x)$ to be

$$\Pr_{\texttt{eval}}\left[\texttt{eval}(x) = \texttt{eval}(x)\right],$$

where the probability is taken over the noise in the evaluation process `eval`. The general reliability of a PUF is the average over all challenges,

$$E_x\left[\Pr_{\texttt{eval}}\left[\texttt{eval}(x) = \texttt{eval}(x)\right]\right].$$

The definition is applied separately for each output bit.

pypuf ships approximations of PUF reliability based on both an instance of *`pypuf.simulation.Simulation`* and on response data given. In some PUFs, the reliability depends on the challenge given, i.e. in one instance, some challenges have hard-to-reproduce responses, while others have very stable response behavior. pypuf hence reports reliability information seperatly for each challenge. To obtain a general notion of reliability, results can be averaged along the first axis.

`pypuf.metrics.`**`reliability`**(*instance: Simulation*, *seed: int*, *N: int = 10000*, *r: int = 17*) → ndarray

For a given simulation, estimates simulated reliability.

Randomly generates N challenges using the `seed`, then queries each challenge r times, obtaining a response array of shape (N, m, r), where m is the response length of the given simulation.

Then applies `reliability_data()` to determine the reliability of each respones bit on each challenge.

Returns a float-array of shape (N, m), giving the reliability of each response bit on each challenge. To obtain the general reliability for each response bit, average along the first axis. To obtain the total reliability of this instance, average across all axes. Note that bad reliability on single response bits may be problematic.

```
>>> from pypuf.simulation import ArbiterPUF, XORArbiterPUF
>>> from pypuf.metrics import reliability
>>> from numpy import average
>>> puf = XORArbiterPUF(n=128, k=2, seed=1, noisiness=.2)
>>> average(reliability(puf, seed=2), axis=0)
array([0.84967612])
```

An example of an extremely noisy PUF:

```
>>> average(reliability(XORArbiterPUF(n=32, k=12, seed=1, noisiness=1), seed=2),␣
↪axis=0)
array([0.52975917])
```

An example of a very reliable PUF:

```
>>> average(reliability(ArbiterPUF(n=32, seed=1, noisiness=.01), seed=2), axis=0)
array([0.99582561])
```

An example of a perfectly reliable PUF:

```
>>> average(reliability(ArbiterPUF(n=64, seed=1, noisiness=0), seed=2), axis=0)
array([1.])
```

`pypuf.metrics.`**`reliability_data`**`(`*responses: ndarray*`)` → ndarray

Computes the reliability of response data of a PUF instance.

Given responses of a PUF instance in an array of shape (N, m, r), where N is the number of challenges queries, m is the response length of the PUF, r is the number of repeated queries for each challenge, this function computes the empirical reliability of the PUF responses, i.e. an approximation of

$$\Pr_{\text{eval}}\left[\text{eval}(x) = \text{eval}(x)\right]$$

where $\text{eval}(x)$ is the response of the (noisy) PUF when evaluated on input x, and the probability is taken over the noise in the evaluation process.

The approximation is derived from an approximation of $E_{\text{eval}}\left[\text{eval}(x)\right]$, which is given by the mean of `responses`,

$$\Pr_{\text{eval}}\left[\text{eval}(x) = \text{eval}(x)\right] = \frac{1}{2} E_{\text{eval}}\left[\text{eval}(x)\right]^2 + \frac{1}{2}.$$

The formula can be obtained by observing $\Pr_{\text{eval}}\left[\text{eval}(x) = \text{eval}(x)\right] = \Pr\left[\text{eval}(x) = 1\right]^2 + \Pr\left[\text{eval}(x) = -1\right]^2$ and $\Pr_{\text{eval}}\left[\text{eval}(x) = 1\right] = \frac{1}{2} E_{\text{eval}}\left[\text{eval}(x)\right] + \frac{1}{2}$.

An array of shape (N, m) is returned, estimating above probability for each challenge and each response bit. To obtain a the reliability for each response bit, average along the first axis, to obtain the general reliability, average over all axes.

```
>>> from pypuf.metrics import reliability_data
>>> from numpy import average, array
>>> responses = array([[[1, 1, -1]], [[1, 1, 1]], [[1, 1, 1]], [[1, 1, 1]]])
>>> average(reliability_data(responses), axis=0)
array([0.88888889])
```

CHAPTER ELEVEN

UNIQUENESS

Uniqueness is a measure for how differently PUF instances of the same class behave. As security is build on the individual behavior, a high uniqueness is a minimum security requirement for any PUF. The concept of uniqueness is also often called *inter-distance*. pypuf can approximate the uniqueness based on given instances of `Simulation` or based on response data, see below.

Uniqueness is estimated on a per-challenge basis, as low uniqueness on a small number of challenges can be problematic. To obtain a general uniqueness measure for each response bit, average results along the first axis.

`pypuf.metrics.`**`uniqueness`**`(`*instances: List[*Simulation*], seed: int, N: int = 10000*`)` → ndarray

Estimates the uniqueness of a list of given Simulations.

Randomly generates N challenges using seed `seed`, then queries each simulation each challenge, obtaining an array with shape (l, N, m) of all responses, where l is the number of simulations, and m is the response length of the given simulations. (All given simulations must have same challenge and response lenght.)

Then applies `uniqueness_data()` to determine the uniqueness of each response bit.

Returns a float-array of shape $(m,)$, giving the uniqueness of each output bit. To obtain total uniqueness, average along all axes, but be aware that low uniqueness on individual response bits may be problematic.

```
>>> from numpy import average
>>> from pypuf.simulation import XORArbiterPUF
>>> from pypuf.metrics import uniqueness
>>> instances = [XORArbiterPUF(n=64, k=1, seed=seed) for seed in range(5)]
>>> uniqueness(instances, seed=31415, N=1000)
array([0.924])
```

`pypuf.metrics.`**`uniqueness_data`**`(`*responses: ndarray*`)` → ndarray

Estimates the uniqueness of responses of a set of PUF instances per challenge.

Given PUF responses in $\{-1, 1\}^m$ in an array of shape (l, N, m), where $l \in \mathbb{N}$ is the number of PUF instances, $N \in \mathbb{N}$ is the number of challenges queried, and $m \in \mathbb{N}$ is the response length, returns an estimate of the uniqueness of PUFs on challenges in this set, i.e. an approximation of

$$1 - 2 \cdot E_{f,g;f \neq g} \left[\left| \frac{1}{2} - \Pr_x \left[f(x) = g(x) \right] \right| \right]$$

where the challenges x are from the (implicitly) provided challenges, $f(x)$ and $g(x)$ are the responses of PUFs f and g described in the provided data.

For perfectly unique PUFs and for any f and g, we expect $\Pr_x \left[f(x) = g(x) \right] = 1/2$, hence the uniqueness to be 1. On the other hand, if any two f and g have similarity higher (or lower) than $1/2$, then the uniqueness will be lower than 1.

Returns an array of shape $(m,)$ with uniqueness values per response bit.

```
>>> from numpy import random, array
>>> from pypuf.metrics import uniqueness_data, similarity_data
>>> prng = random.default_rng(seed=1)
>>> # generate **different** random responses using numpy's pseudo-random generator
>>> responses = array([2 * random.default_rng(seed).integers(0, 2, size=(1000, 3)) -
↪ 1 for seed in range(4)])
>>> uniqueness_data(responses)
array([0.97333333, 0.979     , 0.971     ])
>>> # generate **same** random responses using numpy's pseudo-random generator
>>> responses = array([2 * random.default_rng(1).integers(0, 2, size=(1000, 1)) - 1␣
↪for seed in range(4)])
>>> uniqueness_data(responses)
array([0.])
```

CHAPTER

TWELVE

DISTANCE / SIMILARITY

In the context of machine learning attacks on PUFs, it is often required to estimate the accuracy of predictions made by a PUF model. pypuf provides a metric to estimate the similarity of two PUFs, given either as response data or simulation, which can be used to compute the accuracy of predictions.

`pypuf.metrics.`**`accuracy`**`(`*simulation:* Simulation, *test_set:* ChallengeResponseSet`)` → ndarray

Given a simulation and a test set, returns the relative frequency of responses by the simulation that match responses recorded in the test set by response bit, i.e. an approximation of

$$\Pr_x \left[f(x) = g(x) \right],$$

where f is given by the provided simulation and g is defined by the `test_set`; x is from the set of challenges given in the `test_set`.

If response length is greater than 1, the similarity is given for each bit seperately. To obtain the general similarity, average the response.

Returns an array of shape $(m,)$.

```
>>> from pypuf.simulation import XORArbiterPUF
>>> from pypuf.io import ChallengeResponseSet
>>> from pypuf.metrics import accuracy
>>> puf = XORArbiterPUF(n=128, k=4, noisiness=.1, seed=1)
>>> test_set = ChallengeResponseSet.from_simulation(puf, N=1000, seed=2)
>>> accuracy(puf, test_set)
array([0.843])
>>> puf = XORArbiterPUF(n=64, k=4, noisiness=.3, seed=2)
>>> test_set = ChallengeResponseSet.from_simulation(puf, N=1000, seed=2, r=5)
>>> accuracy(puf, test_set)
array([0.69])
```

`pypuf.metrics.`**`similarity`**`(`*instance1:* Simulation, *instance2:* Simulation, *seed: int*, *N: int = 1000*`)` → ndarray

Approximate the similarity in response behavior of two simulations with identical challenge and response length, i.e. an approximation of

$$\Pr_x \left[f(x) = g(x) \right],$$

where f and g are given by the provided simulations, and the N challenges x are generated randomly using the provided `seed`.

If response length is greater than 1, the similarity is given for each bit seperately. To obtain the general similarity, average the response.

Returns an array of shape $(m,)$.

```
>>> from pypuf.simulation import XORArbiterPUF
>>> from pypuf.metrics import similarity
>>> similarity(XORArbiterPUF(n=128, k=4, seed=1), XORArbiterPUF(n=128, k=4, seed=1),
↪ seed=31415)  # same seed
array([1.])
>>> similarity(XORArbiterPUF(n=128, k=4, seed=1), XORArbiterPUF(n=128, k=4, seed=2),
↪ seed=31415)  # different seed
array([0.516])
```

pypuf.metrics.**similarity_data**(*responses1: ndarray*, *responses2: ndarray*) → ndarray

Given two arrays of responses of shape (N, m), returns the relative frequency of equal responses in the arrays for each response bit, i.e. an approximation of

$$\Pr_x [f(x) = g(x)] ,$$

where f and g are the functions given by `responses1` and `responses2`, respectively; x are the challenges (given implicitly by ordering of the response arrays).

If response length m is greater than 1, the similarity is given for each bit seperately. To obtain the general similarity, average the response.

Returns an array of shape $(m,)$.

```
>>> from pypuf.metrics import similarity_data
>>> from numpy import array
>>> similarity_data(array([[1, 1], [1, 1], [1, 1], [1, 1]]), array([[1, 1], [1, 1],␣
↪[1, 1], [-1, 1]]))
array([0.75, 1.  ])
```

CHAPTER THIRTEEN

CORRELATION

The correlation metric is useful to judge the prediction accuracy when responses are non-binary, e.g. when studying *Integrated Optical PUFs*.

```
>>> import pypuf.simulation, pypuf.io, pypuf.attack, pypuf.metrics
>>> puf = pypuf.simulation.IntegratedOpticalPUF(n=64, m=25, seed=1)
>>> crps = pypuf.io.ChallengeResponseSet.from_simulation(puf, N=1000, seed=2)
>>> crps_test = pypuf.io.ChallengeResponseSet.from_simulation(puf, N=1000, seed=3)
>>> feature_map = pypuf.attack.LeastSquaresRegression.feature_map_optical_pufs_reloaded_
↪improved
>>> model = pypuf.attack.LeastSquaresRegression(crps, feature_map=feature_map).fit()
>>> pypuf.metrics.correlation(model, crps_test).mean()
0.69...
```

Note that the correlation can differ when additionally, post-processing of the responses is performed, e.g. by thresholding the values such that half the values give -1 and the other half 1:

```
>>> import numpy as np
>>> threshold = lambda r: np.sign(r - np.quantile(r.flatten(), .5))
>>> pypuf.metrics.correlation(model, crps_test, postprocessing=threshold).mean()
0.41...
```

`pypuf.metrics.`**`correlation`**(*simulation: ~pypuf.simulation.base.Simulation*, *test_set: ~pypuf.io.ChallengeResponseSet*, *postprocessing: ~typing.Optional[~typing.Callable[~numpy.ndarray, ~numpy.ndarray]] = <function postprocessing_noop>*) → ndarray

Evaluates the given `simulation` on the challenges defined in the `test_set` and computes the `correlations` of the response pixels.

`pypuf.metrics.`**`correlation_data`**(*responses1: ~numpy.ndarray*, *responses2: ~numpy.ndarray*, *postprocessing: ~typing.Optional[~typing.Callable[~numpy.ndarray, ~numpy.ndarray]] = <function postprocessing_noop>*) → ndarray

Given two versions `responses1` and `responses2` of N responses of m pixels each, the m Pearson corrleations of the response pixels are returned. If any `postprocessing` function is given, it is applied to both `responses1` and `responses2` before the correlations are computed.

```
>>> import numpy as np
>>> import pypuf.metrics
>>> responses1 = np.array([[-1, -1, -1, 1], [1, 1, 1, -1]])
>>> responses2 = np.array([[-1, -1, -1, 1], [1, 1, 1, -1]])
>>> pypuf.metrics.correlation_data(responses1, responses2)
array([1., 1., 1., 1.], dtype=float32)
```

(continues on next page)

(continued from previous page)

```
>>> responses1 = np.array([[-1], [-1], [1], [1]])
>>> responses2 = np.array([[1], [1], [-1], [-1]])
>>> pypuf.metrics.correlation_data(responses1, responses2)
array([-1.], dtype=float32)
```

CHAPTER

FOURTEEN

FOURIER-BASED METRICS (PUFMETER)

An advanced method to study Boolean functions is harmonic analysis, also known as Fourier analysis [ODon14]. It can give information about the properties, including learnability, of a given Boolean function. pypuf formalizes (noise-free) PUFs as Boolean functions and can compute some of the metrics used in PUFMeter [GFS19].

`pypuf.metrics.`**`influence`**(*puf: Simulation*, *i: int*, *seed: int*, *N: int = 1000*) → float

Approximates the influence of the i-th input bit on the output of the given PUF simulation.

For a Boolean function $f : \{-1,1\}^n \to \{-1,1\}$, the influence of the i-th input coordinate is defined as the probability that the output of the function changes when the i-th input coordinate is changed, i.e.

$$\mathrm{Inf}_i(f) = \Pr_x \left[f(x) \neq f(x^{\oplus i}) \right],$$

where $x^{\oplus i}$ is the same as x, but with the i-th input coordinate flipped.

The value of $\mathrm{Inf}_i(f)$ is approximated on a sample of uniform random inputs to f.

Ideally, all input bits have an influence of approximately 50% on the response of a PUF function. In contrast, the Arbiter PUF is known to have input bits with extremely high and low influences:

```
>>> import pypuf.simulation, pypuf.metrics
>>> puf = pypuf.simulation.ArbiterPUF(n=128, seed=1)
>>> pypuf.metrics.influence(puf, i=0, seed=2)
0.03
>>> pypuf.metrics.influence(puf, i=127, seed=3)
0.971
```

The Lightweight Secure PUF [MKP08] removed this flaw of the Arbiter PUF, but did not succeed in created a more secure PUF [WBMS19].

```
>>> pypuf.metrics.influence(pypuf.simulation.LightweightSecurePUF(n=128, k=1,␣
↪seed=1), i=0, seed=2)
0.447
```

Parameters

- **`puf`** (*`pypuf.simulation.Simulation`*) – Function f.
- **`i`** (`int`) – The (zero-based) index of the coordinate whose influence will be approximated.
- **`seed`** (`int`) – Determines the seed for the PRNG that generates the uniform random inputs to f.
- **`N`** (`int`) – The number of uniform random inputs that is used to approximate the influence.

Returns

An approximation of the influence of the i-th input coordinate on the output of f, i.e. $\mathrm{Inf}_i(f)$.

Return type
float

`pypuf.metrics.`**`noise_sensitivity`**(*puf:* Simulation, *eps: float*, *seed: int*, *N: int = 1000*) → float

The noise sensitivity captures how a function reacts to noisy inputs. Note that noise here refers to flipped *input bits*, i.e. is different from the usual notion of noise in the context of PUFs. Formally, the noise sensitivity of a Boolean function f at noise-level ε is defined as the probability that the output flips if each input bit is flipped with probability ε (independently). I.e.,

$$\mathrm{NS}_\varepsilon(f) = \Pr_x\left[f(x) \neq f(x')\right],$$

where x' is a copy of x, but each bit is flipped independently with probability ε.

The noise sensitivity is approximated using a uniform random sample of inputs and corresponding with-probability-ε-flipped inputs to f.

A low noise sensitivity of f indicates that further testing is required, as f may be close to a junta and thus susceptible to a modeling attack [GFS19].

Usually, it is interesting to compute the noise sensitivity for small ε:

```
>>> import pypuf.simulation, pypuf.metrics
>>> puf = pypuf.simulation.ArbiterPUF(n=64, seed=1)
>>> pypuf.metrics.noise_sensitivity(puf, eps=.01, seed=2)
0.216
```

Parameters

- **puf** (*pypuf.simulation.Simulation*) – Function f.
- **eps** (`float`) – Noise-level ε of the inputs.
- **seed** (`int`) – Determines the seed for the PRNG that generates the inputs to f.
- **N** (`int`) – The number of uniform random inputs that is used to approximate the noise sensitivity.

Returns
The noise sensitivity of f at noise level ε, i.e. $\mathrm{NS}_\varepsilon(f)$.

Return type
float

`pypuf.metrics.`**`total_influence`**(*puf:* Simulation, *seed: int*, *N: int = 1000*) → float

The total influence, also known as average sensitivity [ODon14], is the sum of all coordinate-wise influences as computed by `influence()`, i.e.

$$\mathrm{I}(f) = \sum_{i=1}^{n} \Pr_x\left[f(x) \neq f(x^{\oplus i})\right].$$

We hence have $0 \leq \mathrm{I}(f) \leq n$.

The total influence can give information about the membership of the function f in certain classes and is used in PUFMeter to analyze PUFs with respect to modeling attacks [GFS19]. Low values of total influence are concerning as they indicate that a modeling attack using the LMN algorithm may be successful.

As an example, the total influence of the `pypuf.simulation.ArbiterPUF` and `pypuf.simulation.PermutationPUF` can be computed as follows:

```
>>> import pypuf.simulation, pypuf.metrics
>>> pypuf.metrics.total_influence(pypuf.simulation.ArbiterPUF(n=64, seed=1), seed=2)
19.358
>>> pypuf.metrics.total_influence(pypuf.simulation.PermutationPUF(n=64, k=1,␣
↪seed=1), seed=2)
32.263...
```

Parameters

- **puf** (*pypuf.simulation.Simulation*) – Function f.
- **seed** (int) – Determines the seed for the PRNG that generates the uniform random inputs to f.
- **N** (int) – The number of uniform random inputs that is used to approximate the influence.

Returns

An approximation of the total influence of f, i.e. $\mathrm{I}_i(f)$.

Return type

float

CHAPTER

FIFTEEN

PUF MODELING ATTACKS

One promise of Physically Unclonable Functions is an increase in hardware security, compared to non-volatile memory key storage solutions. Consequently, the attacker model for PUFs is broadly defined. Usually, the attacker gets physical access to the PUF token for a longer but limited amount of time [WMPN19].

Within this attacker model, it is possible to subject a PUF to *modeling attacks*, where the attacker tries to extrapolate the PUF token's behavior, i.e. to predict the behavior of the PUF when stimulated with unseen challenges. If an attacker is able to predict the PUF token's behavior, a remote party cannot distinguish between the PUF token and the predictive model, breaking the security of the PUF.

Currently implemented attacks in pypuf are all *offline*, i.e. they run on pre-recorded information (as opposed to having online access for adaptively chosen queries).

class `pypuf.attack.base.`**`OfflineAttack`**(*crps: ChallengeInformationSet*)

A modeling attack based on a prerecorded information about a PUF token's behavior on given challenges.

`__init__`(*crps: ChallengeInformationSet*) → None

Initialize the modeling attack. After initialization, the attack can be run using the `fit()` function.

Parameters

`crps` – Information about observed behavior of the PUF on known challenges.

`fit`() → *Simulation*

Runs the attack configured in this attack object. The obtained model is stored as `model()` property and provided as return value.

property model: Optional[*Simulation*]

The model that was obtained running the `fit()` method, `None` if `fit()` was not run yet.

CHAPTER SIXTEEN

LOGISTIC REGRESSION ATTACK

The Logistic Regression Attack (LR Attack) was introduced by Rührmair et al. [RSSD10] to model XOR Arbiter PUFs. In pypuf, an modernized and optimized version of the LR attack is implemented using tensorflow. The original implementation is also available from the authors. A study of the data complexity of the LR attack using a different unpublished implementation of the LR attack was done by Tobisch and Becker [TB15].

16.1 Example Usage

To run the attack, CRP data of the PUF token under attack is required. Such data can be obtained through experiments on real hardware, or using a simulation. In this example, we use the pypuf XOR Arbiter PUF simulator:

```
>>> import pypuf.simulation, pypuf.io
>>> puf = pypuf.simulation.XORArbiterPUF(n=64, k=4, seed=1)
>>> crps = pypuf.io.ChallengeResponseSet.from_simulation(puf, N=50000, seed=2)
```

To run the attack, we configure the attack object with the challenge response data and attack parameters. The parameters need careful adjustment for each choice of security parameters in the PUF. Then the attack is run using the *pypuf.attack.LRAttack2021.fit()* method.

```
>>> import pypuf.attack
>>> attack = pypuf.attack.LRAttack2021(crps, seed=3, k=4, bs=1000, lr=.001, epochs=100)
>>> attack.fit()
    Epoch 1/100
    ...
    50/50 [==============================] - ... - loss: 0.4... - accuracy: 0.9... - val_
↪loss: 0.4643 - val_accuracy: 0.9620
    <pypuf.simulation.base.LTFArray object at 0x...>
>>> model = attack.model
```

The model accuracy can be measured using the pypuf accuracy metric `pypuf.metrics.accuracy()`.

```
>>> import pypuf.metrics
>>> pypuf.metrics.similarity(puf, model, seed=4)
array([0.966])
```

16.2 Applicability

The Logistic Regression can be extended to other variants of the XOR Arbiter PUF [WBMS19]. This functionality is currently not yet included in pypuf, so that the user needs to perform the appropriate conversions before using *`pypuf.attack.LRAttack2021`*.

This implementation is also suitable to conduct the splitting attack on the Interpose PUF [WMPN19].

16.3 API

`class pypuf.attack.`**`LRAttack2021`**(*crps: ChallengeResponseSet*, *seed: int*, *k: int*, *bs: int*, *lr: float*, *epochs: int*, *stop_validation_accuracy: float = 0.95*)

Improved Logistic Regression modeling attack for XOR Arbiter PUFs.

Based on the attack of Rührmair et al. [RSSD10], this version uses tensorflow to model XOR Arbiter PUFs based on observed challenge-response pairs. Compared to the version used by the original authors, this version is based on tensorflow and uses some detail improvements.

Todo: A detailed description of the modifications used in pypuf is currently under consideration for publication. This section will be updated as soon as the manuscript is available to the public.

`__init__`(*crps: ChallengeResponseSet*, *seed: int*, *k: int*, *bs: int*, *lr: float*, *epochs: int*, *stop_validation_accuracy: float = 0.95*) → None

Initialize an improved Logistic Regression attack using the given parameters.

Parameters

- **`crps`** (*`pypuf.io.ChallengeResponseSet`*) – Challenge-response data observed from the PUF under attack. 99% of CRP data will be used as training data, 1% will be used as validation set.
- **`seed`** (`int`) – Random seed for model initialization. Success of the attack may depend on the seed, in particular when little challenge-response data is used.
- **`k`** (`int`) – Number of parallel arbiter chains used in the XOR Arbiter PUF.
- **`bs`** (`int`) – Number of training examples that are processed together. Larger block size benefits from higher confidence of gradient direction and better computational performance, smaller block size benefits from earlier feedback of the weight adoption on following training steps.
- **`lr`** (`float`) – Learning rate of the Adam optimizer used for optimization.
- **`epochs`** (`int`) – Maximum number of epochs performed.
- **`stop_validation_accuracy`** (`float`) – Training is stopped when this validation accuracy is reached. Set to 1 to deactivate.

`fit``()` → *Simulation*

Using tensorflow, runs the attack as configured and returns the obtained model.

Note: Tensorflow will write to stdout.

Todo: Currently, a copy of the challenges is created to compute the features for learning. This essentially doubles memory consumption. If the challenges can be overwritten, this can be performed in-situ to reduce memory footprint of the attack.

Returns
Model of the XOR Arbiter PUF under attack.

property history: Optional[dict]

After *fit()* was called, returns a dictionary that contains information about the training process. The dictionary contains lists of length corresponding to the number of executed epochs:

- `loss` the training loss,
- `val_loss` the validation loss,
- `accuracy` the training accuracy, and
- `val_accuracy` the validation accuracy.

property model: Optional[*Simulation*]

The model that was obtained running the *fit()* method, `None` if *fit()* was not run yet.

16.4 Performance

The pypuf implementation is tested using tensorflow 2.4 on Intel Xeon E5-2630 v4 attacking n-bit k-XOR Arbiter PUFs. The results below are compared with those of Tobisch and Becker [TB15], which have been obtained in 2015 using up to 16 cores.

n	k	CRPs	success rate	duration	cores	[TB15] / 16 cores
64	4	30k	10/10	<1 min	4	<1 min
64	5	260k	10/10	4 min	4	<1 min
64	6	2M	20/20	<1 min	4	1 min
64	7	20M	10/10	3 min	4	55 min
64	8	150M	10/10	28 min	4	391 min
64	9	350M				2266 min
64	9	500M	7/10	14 min	40	
64	10	1B	6/10	41 min	40	

CHAPTER

SEVENTEEN

MULTILAYER PERCEPTRON ATTACK

Multilayer Perceptron (MLP) was first used by Alkatheiri and Zhuang [AZ17] to model Feed-Forward Arbiter PUFs. In follow-up work, Aseeri et al. [AZA18] launched MLP-based modeling attacks against XOR Arbiter PUFs. Thereafter, Mursi et al. [MTZAA20] and Wisiol et al. [WMSZ21] modified the network and parameters used by Aseeri et al. to reduce data complexity of the attack.

pypuf contains two closely related MLP-based modeling attacks. The state-of-the-art attack by Wisiol et al. [WMSZ21] and a re-implementation of the attack by Aseeri et al. [AZA18] using tensorflow/Keras.

17.1 Example Usage [WMSZ21]

To run the attack, CRP data of the PUF token under attack is required. Such data can be obtained through experiments on real hardware, or using a simulation. In this example, we use the pypuf XOR Arbiter PUF simulator:

```
>>> import pypuf.simulation, pypuf.io
>>> puf = pypuf.simulation.XORArbiterPUF(n=64, k=5, seed=1)
>>> crps = pypuf.io.ChallengeResponseSet.from_simulation(puf, N=500000, seed=2)
```

To run the attack, we configure the attack object with the challenge response data and attack parameters. The parameters need careful adjustment for each choice of security parameters in the PUF. Then the attack is run using the *pypuf.attack.MLPAttack2021.fit()* method.

```
>>> import pypuf.attack
>>> attack = pypuf.attack.MLPAttack2021(
...     crps, seed=3, net=[2 ** 4, 2 ** 5, 2 ** 4],
...     epochs=30, lr=.001, bs=1000, early_stop=.08
... )
>>> attack.fit()
    Epoch 1/30
    ...
    495/495 [==============================] - ... - loss: 0.0... - accuracy: 0.9... -␣
↪val_loss: 0.0670 - val_accuracy: 0.9750
    <pypuf.attack.mlp2021.MLPAttack2021.Model object at 0x...>
>>> model = attack.model
```

The model accuracy can be measured using the pypuf accuracy metric `pypuf.metrics.accuracy()`.

```
>>> import pypuf.metrics
>>> pypuf.metrics.similarity(puf, model, seed=4)
array([0.97])
```

17.2 Example Usage [AZA18]

The implementation of the modeling attack by Aseeri et al. [AZA18] in pypuf is very similar to the version by Wisiol et al. [WMSZ21] given above, with notable differences in the parameter settings given to the attack, in the memory management, and framework used. While the attack by Aseeri et al. uses scikit learn, pypuf's implementation is Keras-based. To run the original attack using pypuf, use the network size as defined by Aseeri et al., i.e. $(2^k, 2^k, 2^k)$, and set the activation function of the hidden layers to ReLU. pypuf does not support the memory management introduced by Aseeri et al.

```
>>> puf = pypuf.simulation.XORArbiterPUF(n=64, k=5, seed=1)
>>> crps = pypuf.io.ChallengeResponseSet.from_simulation(puf, N=800000, seed=2)
>>> attack = pypuf.attack.MLPAttack2021(
...     crps, seed=3, net=[2 ** 5, 2 ** 5, 2 ** 5],
...     epochs=30, lr=.001, bs=1000, early_stop=.08,
...     activation_hl='relu',
... )
>>> model = attack.fit()
    Epoch 1/30
    ...
>>> pypuf.metrics.similarity(puf, model, seed=4)[0] > .9
True
```

Note that this is only an approximation of the original work of Aseeri et al., further differences may exist.

17.3 Applicability [WMSZ21]

The attack is noise-resilient and successfully models XOR Arbiter PUFs even if the available training data has response (label) noise [WMSZ21].

This implementation is also suitable to conduct the splitting attack [WMPN19] on the Interpose PUF [WMSZ21].

17.4 API

class pypuf.attack.**MLPAttack2021**(*crps: ChallengeResponseSet, seed: int, net: List[int], epochs: int, lr: float, bs: int, early_stop: float, patience: Optional[int] = None, activation_hl: str = 'tanh'*)

Multilayer-Perceptron modeling attack for XOR Arbiter PUFs.

Inspired by the works of Alkatheiri and Zhuang [AZ17] and Aseeri et al. [AZA18], introduced by Mursi et al. [MTZAA20] and Wisiol et al. [WMSZ21].

__init__(*crps: ChallengeResponseSet, seed: int, net: List[int], epochs: int, lr: float, bs: int, early_stop: float, patience: Optional[int] = None, activation_hl: str = 'tanh'*) → None

Initialize the Multilayer Perceptron modeling attack, using the parameters given.

Note that the complexity of the attack depends crucially on the parameters defined here. The attack by Aseeri et al. [AZA18] uses a network size of $(2^k, 2^k, 2^k)$ to model k-XOR Arbiter PUFs and the ReLU activation function. An advancement of this attack [WMSZ21] uses $(2^{k-1}, 2^k, 2^{k-1})$ and the tanh activation function to model the same with far less required challenge-response data.

Parameters

- **crps** (*pypuf.io.ChallengeResponseSet*) – Challenge-response data observed from the PUF under attack. 99% of CRP data will be used as training data, 1% will be used as validation set.
- **seed** (`int`) – Random seed for model initialization. Success of the attack may depend on the seed, in particular when little challenge-response data is used.
- **net** (`List[int]`) – Hidden-layer sizes for the multilayer perceptron. Note that the layers are all *dense*, i.e. fully connected.
- **epochs** (`int`) – Maximum number of epochs performed.
- **lr** (`float`) – Learning rate of the Adam optimizer used for optimization.
- **bs** (`int`) – Number of training examples that are processed together. Larger block size benefits from higher confidence of gradient direction and better computational performance, smaller block size benefits from earlier feedback of the weight adoption on following training steps.
- **early_stop** (`float`) – Training will stop when validation loss is below this threshold.
- **patience** (`Optional[int]`) – Training will stop when validation loss did not improve for the given number of epochs. Counter is not reset after validation improved in one epoch.
- **activation_hl** (`str`) – Activation function used on the hidden layers.

`fit()` → Model

Using tensorflow, runs the attack as configured and returns the obtained model.

Note: Tensorflow will write to stdout.

Todo: Currently, a copy of the challenges is created to compute the features for learning. This essentially doubles memory consumption. If the challenges can be overwritten, this can be performed in-situ to reduce memory footprint of the attack.

Returns

Model of the XOR Arbiter PUF under attack.

property history: Optional[dict]

After *fit()* was called, returns a dictionary that contains information about the training process. The dictionary contains lists of length corresponding to the number of executed epochs:

- `loss` the training loss,
- `val_loss` the validation loss,
- `accuracy` the training accuracy, and
- `val_accuracy` the validation accuracy.

property model: Optional[*Simulation*]

The model that was obtained running the *fit()* method, `None` if *fit()* was not run yet.

17.5 Performance [WMSZ21]

The pypuf implementation is tested using tensorflow 2.4 on Intel Xeon E5-2630 v4 attacking n-bit k-XOR Arbiter PUFs. Memory recorded here may be higher than actually needed by the most recent version of the attack.

reliability	n	k	CRPs	success rate	duration	cores	memory
1.00	64	4	150k	10/10	<1 min	40	1 GiB
1.00	64	5	200k	10/10	<1 min	20	3 GiB
1.00	64	6	2M	10/10	<1 min	40	2 GiB
1.00	64	7	4M	10/10	<1 min	40	3 GiB
1.00	64	8	6M	7/10	13 min	4	
1.00	64	9	45M	10/10	16 min	40	14 GiB
1.00	64	10	119M	7/10	291 min	40	41 GiB
1.00	64	11	325M	10/10	1898 min	40	104 GiB
1.00	128	4	1M	9/9	<1 min	40	1 GiB
1.00	128	5	1M	10/10	<1 min	40	2 GiB
1.00	128	6	10M	9/10	<1 min	20	5 GiB
1.00	128	7	30M	10/10	2 min	20	20 GiB
1.00	256	4	6M	10/10	1 min	40	6 GiB
1.00	256	5	10M	10/10	3 min	40	11 GiB
1.00	256	6	30M	0/8	–	40	33 GiB
1.00	256	7	100M	1/10	8 min	40	98 GiB
0.85	64	4	180k	9/10	<1 min	4	<1 GiB
0.85	64	5	150k	10/10	<1 min	4	<1 GiB
0.85	64	6	2M	10/10	<1 min	4	1 GiB
0.85	64	7	4M	9/9	3 min	4	2 GiB

CHAPTER

EIGHTEEN

LMN ALGORITHM

The LMN Algorithm can compute models for functions with Fourier spectra concentrated on the low degrees [LMN93], [ODon14] and part of the PUFMeter [GFS19] PUF testing toolbox.

The attack requires access to a number of uniformly random challenge-response pairs (CRPs). Depending on the amount of provided CRPs and the type of function provided, the results may have, with certain probability, a guaranteed accuracy. (For more details, we refer the reader to the PAC learning framework [ODon14].)

18.1 Example Usage

To run the attack, CRP data of the PUF token under attack is required. Such data can be obtained through experiments on real hardware, or using a simulation. In this example, we use the pypuf Arbiter PUF simulator and configure it to use feature vectors as inputs rather than challenge vectors by setting `transform='id'`:

```
>>> import pypuf.simulation, pypuf.io
>>> puf = pypuf.simulation.ArbiterPUF(n=32, transform="id", seed=1)
>>> challenges = pypuf.io.random_inputs(n=32, N=2000, seed=2)
>>> crps = pypuf.io.ChallengeResponseSet(challenges, puf.val(challenges))
```

To run the attack, we need to decide how many levels of Fourier coefficients we want to approximate. There are n levels, the i-th level has $\binom{n}{i}$ coefficients. The run time of the attack is linear in the number of coefficients. With the steeply increasing run time, in practice, degree 1 and degree 2 are reasonable choices. Increasing the degree further will not only lead to high requirements on computation time, but may actually *worsen* the predictive accuracy, as more coefficients need to be approximated.

```
>>> import pypuf.attack
>>> attack = pypuf.attack.LMNAttack(crps, deg=1)
>>> model = attack.fit()
```

The model accuracy can be measured using the pypuf accuracy metric `pypuf.metrics.accuracy()`.

```
>>> import pypuf.metrics
>>> pypuf.metrics.similarity(puf, model, seed=4)
array([0.95])
```

18.2 API

class pypuf.attack.**LMNAttack**(*crps: ChallengeResponseSet*, *deg: int = 1*)

__init__(*crps: ChallengeResponseSet*, *deg: int = 1*) → None

Given a list of function values of a function $f : \{-1, 1\}^n \to \mathbb{R}$, an approximation of the Fourier spectrum of the underlying function can be computed.

The approximation is guaranteed to be correct, if the list contains all function values and the degree equals n, shown here using the Boolean AND function:

```
>>> import numpy as np
>>> import pypuf.io, pypuf.attack
>>> challenges = 1 - 2 * np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
>>> responses_and = 1 - 2 * np.array([[0], [0], [0], [1]])
>>> crps_and = pypuf.io.ChallengeResponseSet(challenges, responses_and)
>>> (1 - pypuf.attack.LMNAttack(crps_and, 2).fit().eval(challenges)) // 2
array([[0.],
       [0.],
       [0.],
       [1.]], dtype=float32)
```

If additionally the responses are from $\{-1, 1\}$, then the sum of squares of the Fourier coefficients equals 1, as illustrated here using the majority vote function:

```
>>> challenges = 1 - 2 * np.array([[0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1],␣
↪[1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]])
>>> responses_maj = 1 - 2 * np.array([[0], [0], [0], [1], [0], [1], [1], [1]])
>>> crps_maj = pypuf.io.ChallengeResponseSet(challenges, responses_maj)
>>> exp = pypuf.attack.LMNAttack(crps_maj, 3).fit().expansion
>>> exp
array([[ 0. ,  0.5,  0.5,  0.5,  0. ,  0. ,  0. , -0.5]], dtype=float32)
>>> np.sum(exp**2)
1.0
```

fit() → FourierSimulation

Runs the attack configured in this attack object. The obtained model is stored as *model()* property and provided as return value.

property model: *Simulation*

The model that was obtained running the *fit()* method, None if *fit()* was not run yet.

CHAPTER

NINETEEN

LINEAR REGRESSION

Linear Regression fits a linear function on given data. The resulting linear function, also called map, is guaranteed to be optimal with respect to the total squared error, i.e. the sum of the squared differences of actual value and predicted value.

Linear Regression has many applications, in pypuf, it can be used to model *Integrated Optical PUFs* and *Arbiter PUFs and Compositions*.

19.1 Arbiter PUF Reliability Side-Channel Attack [DV13]

For Arbiter PUFs, the reliability for any given challenge c has a close relationship with the difference in delay for the top and bottom line. When modeling the Arbiter PUF response as

$$r = \operatorname{sgn}\left[D_{\text{noise}} + \langle w, x \rangle\right],$$

where x is the feature vector corresponding to the challenge c and $w \in \mathbb{R}^n$ are the weights describing the Arbiter PUF, and D_{noise} is chosen from a Gaussian distribution with zero mean and variance σ^2_{noise} to model the noise, then we can conclude that

$$\mathrm{E}[r(x)] = \operatorname{erf}\left(\frac{\langle w, x \rangle}{\sqrt{2}\sigma_{\text{noise}}}\right).$$

Hence, the delay difference $\langle w, x \rangle$ can be approximated based on an approximation of E[r(x)], which can be easily obtained by an attacker. It gives

$$\langle w, x \rangle = \sqrt{2}\sigma_{\text{noise}} \cdot \operatorname{erf}^{-1}\mathrm{E}[r(x)].$$

This approximation works well even when $\mathrm{E}[r(x)]$ is approximated based on only on few responses, e.g. 3 (see below).

To demonstrate the attack, we initialize an Arbiter PUF simulation with noisiness chosen such that the reliability will be about 91% *on average*:

```
>>> import pypuf.simulation, pypuf.io, pypuf.attack, pypuf.metrics
>>> puf = pypuf.simulation.ArbiterPUF(n=64, noisiness=.25, seed=3)
>>> pypuf.metrics.reliability(puf, seed=3).mean()
0.908...
```

We then create a CRP set using the *average* value of responses to 500 challenges, based on 5 measurements:

```
>>> challenges = pypuf.io.random_inputs(n=puf.challenge_length, N=500, seed=2)
>>> responses_mean = puf.r_eval(5, challenges).mean(axis=-1)
>>> crps = pypuf.io.ChallengeResponseSet(challenges, responses_mean)
```

Based on these approximated values `responses_mean` of the linear function $\langle w, x \rangle$, we use linear regression to find a linear mapping with small error to fit the data. Note that we use the `transform_atf` function to compute the feature vector x from the challenges c, as the mapping is linear in x (but not in c).

```
>>> attack = pypuf.attack.LeastSquaresRegression(crps, feature_map=lambda cs: pypuf.
↪simulation.ArbiterPUF.transform_atf(cs, k=1)[:, 0, :])
>>> model = attack.fit()
```

The linear map `model` will predict the delay difference of a given challenge. To obtain the predicted PUF response, this prediction needs to be thresholded to either -1 or 1:

```
>>> model.postprocessing = model.postprocessing_threshold
```

To measure the resulting model accuracy, we use `pypuf.metrics.similarity()`:

```
>>> pypuf.metrics.similarity(puf, model, seed=4)
array([0.902])
```

19.2 Modeling Attack on Integrated Optical PUFs [RHUWDFJ13]

The behavior of an integrated optical PUF token can be understood as a linear map $T \in \mathbb{C}^{n \times m}$ of the given challenge, where the value of T are determined by the given PUF token, and n is number of challenge pixels, and m the number of response pixels. The speckle pattern of the PUF is a measurement of the intensity of its electromagnetic field at the output, hence the intensity at a given response pixel r_i for a given challenge c can be written as

$$r_i = |c \cdot T|^2 .$$

pypuf ships a basic simulator for the responses of *Integrated Optical PUFs*, on whose data a modeling attack can be demonstrated. We first initialize a simulation and collect challenge-response pairs:

```
>>> puf = pypuf.simulation.IntegratedOpticalPUF(n=64, m=25, seed=1)
>>> crps = pypuf.io.ChallengeResponseSet.from_simulation(puf, N=1000, seed=2)
```

Then, we fit a linear map on the data contained in `crps`. Note that the simulation returns *intensity* values rather than *field* values. We thus need to account for quadratic terms using an appropriate `feature map`.

```
>>> attack = pypuf.attack.LeastSquaresRegression(crps, feature_map=pypuf.attack.
↪LeastSquaresRegression.feature_map_optical_pufs_reloaded_improved)
>>> model = attack.fit()
```

The success of the attack can be visually inspected or quantified by the *Correlation* of the response pixels:

```
>>> crps_test = pypuf.io.ChallengeResponseSet.from_simulation(puf, N=1000, seed=3)
>>> pypuf.metrics.correlation(model, crps_test).mean()
0.69...
```

Note that the correlation can differ when additionally, post-processing of the responses is performed, e.g. by thresholding the values such that half the values give -1 and the other half 1:

```
>>> import numpy as np
>>> threshold = lambda r: np.sign(r - np.quantile(r.flatten(), .5))
>>> pypuf.metrics.correlation(model, crps_test, postprocessing=threshold).mean()
0.41...
```

19.3 API

class pypuf.attack.**LeastSquaresRegression**(*crps: ChallengeResponseSet*, *feature_map: Optional[Callable[ndarray, ndarray]] = None*)

__init__(*crps: ChallengeResponseSet*, *feature_map: Optional[Callable[ndarray, ndarray]] = None*) → None

Initialize the modeling attack. After initialization, the attack can be run using the *fit()* function.

Parameters

crps – Information about observed behavior of the PUF on known challenges.

static feature_map_optical_pufs_reloaded(*challenges: ndarray*) → ndarray

Computes features of an optical PUF token using all ordered pairs of challenge bits [RHUWDFJ13]. An optical system may be linear in these features.

Note: This representation is redundant since it treats ordered paris of challenge bits are distinct. Actually, only unordered pairs of bits should be treated as distinct. For applications, use the function *feature_map_optical_pufs_reloaded_improved*, which achieves the same with half the number of features.

Parameters

challenges – array of shape (N, n) representing challenges to the optical PUF.

Returns

array of shape (N, n^2), which, for each challenge, contains the flattened dyadic product of the challenge with itself.

static feature_map_optical_pufs_reloaded_improved(*challenges: ndarray*) → ndarray

Computes features of an optical PUF token using all unordered pairs of challenge bits [RHUWDFJ13]. An optical system may be linear in these features.

Parameters

challenges – 2d array of shape (N, n) representing N challenges of length n.

Returns

array of shape $(N, \frac{n \cdot (n+1)}{2})$. The result *return[i]* consists of all products of unordered pairs taken from *challenges[i]*, which has shape *(N,)*.

```
>>> import numpy as np
>>> import pypuf.attack
>>> challenges = np.array([[2, 3, 5], [1, 0, 1]])  # non-binary numbers for␣
↪illustration only.
>>> pypuf.attack.LeastSquaresRegression.feature_map_optical_pufs_reloaded_
↪improved(challenges)
array([[ 4,  6, 10,  9, 15, 25],
       [ 1,  0,  1,  0,  0,  1]])
```

fit() → *Simulation*

Runs the attack configured in this attack object. The obtained model is stored as *model()* property and provided as return value.

property model: ***Simulation***

The model that was obtained running the *fit()* method, None if *fit()* was not run yet.

CHAPTER

TWENTY

RANDOMNESS AND REPRODUCIBILITY

pypuf strives to make all results fully reproducible.

One key ingredient to this is to only use seeded random number generators. To avoid re-using seeds where different random generators are expected, pypuf implements a convenience function to obtain random generators based on a string description.

`pypuf.random.prng`(*description: str*) → Generator

Returns an instance of `numpy.random.Generator` seeded based on a text description.

```
>>> from pypuf.random import prng
>>> seed = 5
>>> my_prng = prng(f'my favorite random numbers, seed {seed}')
>>> my_prng.integers(2, 6, 3)
array([4, 3, 2])
```

CHAPTER

TWENTYONE

LARGE-SCALE EXPERIMENTS

To assess the quality of an attack or Strong PUF design, it is often necessary to run several attacks for different parameter settings and many instances. Often, attacks will run long or require much memory.

In pypuf, *studies* can be defined that run a predefined *experiment* for a parameter matrix. A CLI is provided that can run single, subsets or all experiments.

An example study could look like this:

```python
import sys
from datetime import timedelta
from typing import List

from pypuf.batch import StudyBase


class ExampleStudy(StudyBase):

    @staticmethod
    def parameter_matrix() -> List[dict]:
        N = {
            (64, 1): 4000,
            (64, 2): 8000,
            (64, 4): 30000,
            (128, 1): 10000,
            (128, 2): 20000,
            (128, 4): 60000,
        }

        return [
            {'n': n, 'k': k, 'N': N[n, k]}
            for n in [64, 128]
            for k in [1, 2, 4]
        ]

    def primary_results(self, results: dict) -> dict:
        return {
            'accuracy': results['accuracy'],
            'duration': results['duration'],
        }

    def run(self, n: int, k: int, N: int) -> dict:
```

(continues on next page)

(continued from previous page)

```
        # Do some sort of experiment to determine so sort of result

        # If this runs long, maybe it's useful to store some data in self.log,
        # and store to disk with self.save_log()
        self.log = []
        self.log.append({
            'accuracy': 1.0  # just any information about the current state of affairs
        })  # can be any pickle-able data type!
        self._save_log()  # calls to save_log a throttled automatically, unless 'force'␣
↪is used.

        # The result is represented by a dict
        return {
            'accuracy': 1.0,  # VERY GOOD experiment done here ...
            'duration': timedelta(seconds=3),  # .. and fast
            'additonal_info': 'foobar!',
            'is_example': True,
        }


if __name__ == '__main__':
    ExampleStudy.cli(sys.argv)
```

Such a study can be run with `python3 -m example_study debug 0 1`. The parameters are defined as follows. In `python3 -m <module> <results> <idx> <total>`,

1. <module> sets the name of the module to be run, i.e. the module where `ExampleStudy.cli` is called;
2. `<results>` sets the name of the result file in the current directory, in which all return values of the *run* method are stored,
3. `<idx>` is the zero-based index of the subset of parameters that shall be run,
4. `<total>` is the total number of subsets (blocks).

This interface nicely interact with SLURM, where a job file like this can be used to distribute pypuf studies across a SLURM cluster:

```
#!/bin/bash

# 1. Adjust
#    - email address;
#    - job name;
#    - time, memory, CPUs;
#    - study length.
# 2. Make sure to load the correct module
# 3. Setup the virtual environment and cd and source correctly
#
# Then run with
#   sbatch --array=0..39 this_file.sh  # adjust for study length
#

#SBATCH --mail-type=END
#SBATCH --mail-user=<email>
#SBATCH --job-name <jobname>
```

(continues on next page)

(continued from previous page)

```
#SBATCH --time=2-00:00:00
#SBATCH --mem=8G
#SBATCH --cpus-per-task=2
#SBATCH --nodes=1

# Usually High Performance Clusters will need to load Python before it is available
module load python/3.7.6_tf_2.1.0

# Navigate to your study file, if necessary
cd ~/my_study/

# Load your virtual environment where pypuf is installed
source venv/bin/activate

# Limit the number of CPUs used by numpy and tensorflow
export TF_NUM_INTRAOP_THREADS=${SLURM_CPUS_PER_TASK}
export TF_NUM_INTEROP_THREADS=${SLURM_CPUS_PER_TASK}
export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}
export NUMEXPR_NUM_THREADS=${SLURM_CPUS_PER_TASK}
export MKL_NUM_THREADS=${SLURM_CPUS_PER_TASK}

# run the study
python3 -m example_study "${SLURM_JOB_NAME}" "${SLURM_ARRAY_TASK_ID}" 40
```

CHAPTER TWENTYTWO

ACKNOWLEDGEMENTS

This version of pypuf is based on a development version with contributions due to (in chronological order) Nils Wisiol, Christoph Graebnitz, Christopher Mühl, Benjamin Zengin, Tudor A. A. Soroceanu, and Niklas Pirnay.

Former versions of pypuf were used in a number of research articles [WP20] [WMPN19] [WBMS19] [WM19].

CHAPTER

TWENTYTHREE

BIBLIOGRAPHY

CHAPTER

TWENTYFOUR

GETTING STARTED

pypuf is available via pypi:

```
pip3 install pypuf
```

To simulate an XOR Arbiter PUF with 64 bit challenge length and 2 arbiter chains, follow these steps.

First, create a new XOR Arbiter PUF instance with the given dimensions and a fixed seed for reproducibility.

```
>>> from pypuf.simulation import XORArbiterPUF
>>> puf = XORArbiterPUF(n=64, k=2, seed=1)
```

Then generate a list of *N* random challenges of length 64 (again with a seed for reproducibility).

```
>>> from pypuf.io import random_inputs
>>> challenges = random_inputs(n=64, N=10, seed=2)
```

Finally, evaluate the XOR Arbiter PUF on these challenges, it will yield 10 responses.

```
>>> puf.eval(challenges)
array([-1, -1,  1,  1,  1,  1,  1,  1, -1, -1], dtype=int8)
```

For a more detailed information on simulation of PUFs, continue with *the section on simulations*.

CHAPTER

TWENTYFIVE

GETTING HELP

If you need help beyond this documentation, please contact me at pypuf(a-t)nils-wisiol.de.

TWENTYSIX

CITATION

To refer to pypuf, please use DOI *10.5281/zenodo.3901410*. pypuf is published via Zenodo. Please cite this work as

> Nils Wisiol, Christoph Gräbnitz, Christopher Mühl, Benjamin Zengin, Tudor Soroceanu, Niklas Pirnay, Khalid T. Mursi, & Adomas Baliuka. pypuf: Cryptanalysis of Physically Unclonable Functions (Version v2, August 2021). Zenodo. https://doi.org/10.5281/zenodo.3901410

or use the following BibTeX:

```
@software{pypuf,
  author       = {Nils Wisiol and
                  Christoph Gräbnitz and
                  Christopher Mühl and
                  Benjamin Zengin and
                  Tudor Soroceanu and
                  Niklas Pirnay and
                  Khalid T. Mursi and
                  Adomas Baliuka},
  title        = {{pypuf: Cryptanalysis of Physically Unclonable
                   Functions}},
  year         = 2021,
  publisher    = {Zenodo},
  version      = {v2},
  doi          = {10.5281/zenodo.3901410},
  url          = {https://doi.org/10.5281/zenodo.3901410}
}
```

CHAPTER

TWENTYSEVEN

ABOUT THIS DOCUMENT

To add to our documentation or fix a mistake, please submit a Pull Request at https://github.com/nils-wisiol/pypuf.

BIBLIOGRAPHY

[AM21] Aghaie, A. & Moradi, A. Inconsistency of Simulation and Practice in Delay-based Strong PUFs. IACR Transactions on Cryptographic Hardware and Embedded Systems 520–551 (2021) doi:10.46586/tches.v2021.i3.520-551.

[AZ17] Alkatheiri, M. S. & Zhuang, Y. Towards fast and accurate machine learning attacks of feed-forward arbiter PUFs. in 2017 IEEE Conference on Dependable and Secure Computing 181–187 (2017). doi:10.1109/DESEC.2017.8073845.

[AZA18] Aseeri, A. O., Zhuang, Y. & Alkatheiri, M. S. A Machine Learning-Based Security Vulnerability Study on XOR PUFs for Resource-Constraint Internet of Things. in 2018 IEEE International Congress on Internet of Things (ICIOT) 49–56 (2018). doi:10.1109/ICIOT.2018.00014.

[Bec15] Becker, G. T. The Gap Between Promise and Reality: On the Insecurity of XOR Arbiter PUFs. in Cryptographic Hardware and Embedded Systems – CHES 2015 (eds. Güneysu, T. & Handschuh, H.) 535–555 (Springer Berlin Heidelberg, 2015).

[CCLSR11] Chen, Q., Csaba, G., Lugli, P., Schlichtmann, U. & Ruhrmair, U. The Bistable Ring PUF: A new architecture for strong Physical Unclonable Functions. in 2011 IEEE International Symposium on Hardware-Oriented Security and Trust 134–141 (IEEE, 2011). doi:10.1109/HST.2011.5955011.

[CCPG21] Charlot, N., Canaday, D., Pomerance, A. & Gauthier, D. J. Hybrid Boolean Networks as Physically Unclonable Functions. IEEE Access 9, 44855–44867 (2021).

[DV13] Delvaux, J. & Verbauwhede, I. Side channel modeling attacks on 65nm arbiter PUFs exploiting CMOS device noise. in Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on 137–142 (IEEE, 2013).

[GCvDD02] Gassend, B., Clarke, D., van Dijk, M. & Devadas, S. Silicon Physical Random Functions. in Proceedings of the 9th ACM Conference on Computer and Communications Security 148–160 (ACM, 2002). doi:10.1145/586110.586132.

[GFS19] Ganji, F., Forte, D. & Seifert, J.-P. PUFmeter a Property Testing Tool for Assessing the Robustness of Physically Unclonable Functions to Machine Learning Attacks. IEEE Access 7, 122513–122521 (2019).

[GLCDD04] Gassend, B., Lim, D., Clarke, D., Dijk, M. van & Devadas, S. Identification and authentication of integrated circuits. Concurrency and Computation: Practice and Experience 16, 1077–1098 (2004).

[LMN93] Linial, N., Mansour, Y. & Nisan, N. Constant Depth Circuits, Fourier Transform, and Learnability. J. ACM 40, 607–620 (1993).

[MTZAA20] Mursi, K. T., Thapaliya, B., Zhuang, Y., Aseeri, A. O. & Alkatheiri, M. S. A Fast Deep Learning Method for Security Vulnerability Study of XOR PUFs. Electronics 9, 1715 (2020).

[MKP08] Majzoobi, M., Koushanfar, F. & Potkonjak, M. Lightweight Secure PUFs. in Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design 670–673 (IEEE Press, 2008).

[NSJM19] Nguyen, P. H. et al. The Interpose PUF: Secure PUF Design against State-of-the-art Machine Learning Attacks. IACR Transactions on Cryptographic Hardware and Embedded Systems 243–290 (2019) doi:10.13154/tches.v2019.i4.243-290.

[ODon14] O'Donnell, R. Analysis of Boolean Functions. (Cambridge University Press, 2014).

[RHUWDFJ13] Rührmair, U. et al. Optical PUFs Reloaded. https://eprint.iacr.org/2013/215 (2013).

[RSSD10] Rührmair, U. et al. Modeling Attacks on Physical Unclonable Functions. in Proceedings of the 17th ACM Conference on Computer and Communications Security 237–249 (ACM, 2010). doi:10.1145/1866307.1866335.

[SD07] Suh, G. E. & Devadas, S. Physical Unclonable Functions for Device Authentication and Secret Key Generation. in Proceedings of the 44th Annual Design Automation Conference 9–14 (ACM, 2007). doi:10.1145/1278480.1278484.

[SH14] Schuster, D. & Hesselbarth, R. Evaluation of Bistable Ring PUFs Using Single Layer Neural Networks. in Trust and Trustworthy Computing (eds. Holz, T. & Ioannidis, S.) 101–109 (Springer International Publishing, 2014).

[TB15] Tobisch, J. & Becker, G. T. On the scaling of machine learning attacks on PUFs with application to noise bifurcation. in International Workshop on Radio Frequency Identification: Security and Privacy Issues 17–31 (Springer, 2015).

[WBMS19] Wisiol, N. et al. Breaking the Lightweight Secure PUF: Understanding the Relation of Input Transformations and Machine Learning Resistance. in Smart Card Research and Advanced Applications (eds. Belaïd, S. & Güneysu, T.) 40–54 (Springer International Publishing, 2020). doi:10.1007/978-3-030-42068-0_3.

[WM19] Wisiol, N. & Margraf, M. Why attackers lose: design and security analysis of arbitrarily large XOR arbiter PUFs. J Cryptogr Eng 9, 221–230 (2019).

[WMPN19] Wisiol, N. et al. Splitting the Interpose PUF: A Novel Modeling Attack Strategy. IACR Transactions on Cryptographic Hardware and Embedded Systems 97–120 (2020) doi:10.13154/tches.v2020.i3.97-120.

[WMSZ21] Wisiol, N., Mursi K. T., Seifert, J.-P., Zhuang, Y. Neural-Network-Based Modeling Attacks on XOR Arbiter PUFs Revisited. In submission (2021).

[WP20] Wisiol, N. & Pirnay, N. Short Paper: XOR Arbiter PUFs Have Systematic Response Bias. in Financial Cryptography and Data Security (eds. Bonneau, J. & Heninger, N.) 50–57 (Springer International Publishing, 2020). doi:10.1007/978-3-030-51280-4_4.

[XRHB15] Xu, X., Rührmair, U., Holcomb, D. E. & Burleson, W. Security Evaluation and Enhancement of Bistable Ring PUFs. in Radio Frequency Identification (eds. Mangard, S. & Schaumont, P.) 3–16 (Springer International Publishing, 2015).

PYTHON MODULE INDEX

INDEX