Modelling Cryptographic Distinguishers Using Machine Learning

(article starts on next page)

REGULAR PAPER

Modelling cryptographic distinguishers using machine learning

Carlo Brunetta[1] · Pablo Picazo-Sanchez[1]

Abstract

Cryptanalysis is the development and study of attacks against cryptographic primitives and protocols. Many cryptographic properties rely on the difficulty of generating an adversary who, given an object sampled from one of two classes, correctly distinguishes the class used to generate that object. In the case of cipher suite distinguishing problem, the classes are two different cryptographic primitives. In this paper, we propose a methodology based on machine learning to automatically generate classifiers that can be used by an adversary to solve any distinguishing problem. We discuss the assumptions, a basic approach for improving the advantage of the adversary as well as a phenomenon that we call the *"blind spot paradox"*. We apply our methodology to generate distinguishers for the NIST (DRBG) cipher suite problem. Finally, we provide empirical evidence that the distinguishers might statistically have some advantage to distinguish between the DRBG used.

Keywords Cryptanalysis · Distinguisher · Machine learning · Cipher suite distinguishing problem · Pseudorandom generator

1 Introduction

Nowadays, we use cryptography for almost all online activities, *e.g.* payments, secure messaging and web navigation. Even though cryptography is usually seen as *"one"* piece, this is not the case. Cryptographic protocols use different primitives to provide security, and it is always required that each primitive needs to achieve a certain level of resistance against different attacks to be considered secure. Thus, it is crucial to define and classify *what an attack is*.

To define a cryptographic attack, we need to specify the *goal* and the *abilities* the adversary has with respect to a *security model* that describes how the primitive is used and attacked [16,19]. For example, the plaintext recovering of an encrypted message without having the key is, by no means, the classical example of an attack on an encryption scheme.

Let us consider the concrete scenario in which an adversary is given an object sampled from one of two possible classes and the goal is to correctly guess the class used to generate that object. This adversary is known as a **distinguisher** for a *distinguishing problem*.

A classical example of a distinguishing problem in cryptography is the one used for the pseudorandomness property of a primitive (G) [16]. Such a problem is defined as how to distinguish elements generated by G from those uniformly random generated, *i.e.* (G, rand). In other words, to prove the non-pseudorandomness, it would be sufficient to create a distinguisher D with a non-negligible advantage for solving the related distinguishing problem.

(ML) and cryptography have been widely combined in the literature, *e.g.* from random numbers generation [20,23], random number prediction [17,18,22] and supervised algorithm using encrypted data [12] to testing how good a (PRG) is [10].

Our contributions In this paper, we propose a constructive methodology based on ML that allows the generation of several distinguishers $\{D_i\}_i$ for a given distinguishing problem between two classes (G_0, G_1). We implement a tool named MLCrypto and freely release the code to facilitate future work on this line[1].

In a nutshell, we generate a dataset that contains tuples of elements y_i together with the classes from which they are sampled. This dataset is the input of an ML algorithm whose outputs are a distinguisher D_i. It also generates strategies and solutions to allow an adversary to improve her advantage. Concretely, we present a strategy that allows us to combine several distinguishers $(\{D_i\}_i)$ generated by MLCrypto to cre-

✉ Carlo Brunetta
  brunetta@chalmers.se

  Pablo Picazo-Sanchez
  pablop@chalmers.se

[1] Department of Computer Science, Chalmers University of Technology, Gothenburg, Sweden

---

[1] https://bitbucket.org/CharlieTrip/mlcryptocode/src/master/.

ate a more accurate distinguisher D. We further discuss the *blind spot paradox*, a paradoxical phenomenon that can annihilate any advantage when the attacker unconscionably uses tools like MLCrypto in realistic attack scenarios.

We present a case study on the cipher suite distinguishing problem on the PRG and based on the (NIST) DRBG. We remark the state-of-the-art generation of a distinguisher from statistical test suites and link it with the advantage in breaking the pseudorandomness property, that is, having an advantage in discriminating between the PRG and a random element, with the advantage of distinguishing between two PRG (G_0, G_1). We design an experiment that uses MLCrypto as a distinguisher generator between DRBG recommended by NIST [4]. In more detail, MLCrypto generates Naive Bayes classifiers because of their (i) computational efficiency, (ii) implementation simplification and (iii) the lack of learning parameter to be tuned.

From our experiments, we conclude that both our methodology and MLCrypto can be used for efficiently generating general purpose distinguishers.

Case study: distinguishing NIST DBRGs There are two main approaches to generate distinguishers: theoretical and empirical. The theoretical approach consists of searching for flaws by scrutinising the mathematical primitive definition. For instance, there are theoretical attacks [8,28] against PRG proposed by NIST [4] based on specific differential cryptanalysis [6]. The empirical approach relies on defining a statistically significant number of experiments to provide enough confidence of the results, used to create a distinguisher. For instance, the test suite provided by NIST [5] is composed of multiple statistical tests that check whether the outputs generated by the PRG have some kind of correlation with the presence of some pattern—defined by each one of the tests. After running these tests, the outputs are compared to the result that a uniform distribution generates. The more the passed tests, the more the confidence in stating that the PRG is pseudorandom.

However, all these tests can, and more specifically the failed ones, be used to distinguish between PRG and real randomness. By observing the failing tests, a distinguisher can infer that the input elements are generated by PRG. They can be used to define fingerprints of the PRG, *i.e.* each PRG is prone to fail the same tests, uniquely identifying them. Concretely, this distinguisher can be used to solve a related problem named *cipher suite distinguishing problem* [16]. Similar to pseudorandomness, an attacker has to discriminate between objects generated by two different primitives (G_0, G_1) and not from random elements.

Related work Other works propose to distinguish between random numbers generated with block ciphers [2,9,11,14,15, 25,29] of which a vast majority extract features coming from the statistical tests proposed by NIST (NIST STS) [5] and use them as inputs of ML algorithms. While the documentation provided by the NIST does not provide any formal security analysis [13], Woodgate *et al.* [28] carry out an indepth security review. Contrarily to prior proposals, we apply MLCrypto to DRBG recommended by NIST [4], being able to statistically distinguish between two pairs of generators.

To extract features from NIST STS to distinguish between random data generated from block ciphers, Zhao *et al.* [29] use (SVM). They use OpenSSL to generate ciphertexts from AES, Camellia, Blowfish, DES, IDEA and TDEA algorithms. Authors derive 54 features from the NIST STS, obtaining that accuracies of 42 features are higher than 50% while the accuracies of 12 features are higher than 60%. Hu *et al.* [15] use random forest to classify random data from 16 block chipers instead of the 6 that Zhao *et al.* use, obtaining an accuracy of 88% in the classification. Svenda *et al.* [26] use software circuits together with evolutionary algorithms to search for patterns, random bit predictability and random data indistinguishability.

Contrarily to the aforementioned works, instead of distinguishing between *random* data, we use MLCrypto as a machine learning approach to distinguish between *the functions* that generate these data, *i.e.* in our case study, we create distinguishers between NIST DRBG [4].

Paper organisation In Sect. 2, we give a brief introduction to pseudorandom generators, NIST DRBG and machine learning. Section 3 describes the methodology for generating distinguishers using machine learning and additionally discusses limitation, such as the blind spot paradox, and a possible strategy to amplify the adversarial advantage. In Sect. 4, we implement our methodology into the MLCrypto tool and consider a particular case study based on DRBG recommended by NIST. This paper ends with ideas for future work in Sect. 5.

2 Preliminaries

In this section, we present definitions and concepts used throughout the paper.

Notation Let $\Pr_{x \in X} [E]$ denote the probability computed over the $x \in X$ that the event E occurs. We will omit the probability space whenever it is clear by the context, *i.e.* $\Pr[E]$. The random sampling in the set X is denoted as $x \leftarrow_\$ X$ and, whenever it is not specified, the sampling is always considered to be uniform at random. Let the natural number be denoted with $\mathbb{N}$, the real number field with $\mathbb{R}$ and the positive ones with $\mathbb{R}_+$. Let $[a, b]$ denote the interval between a and b, comprised. The space of binary strings of length ℓ is $\{0, 1\}^\ell$ while $\parallel$ denotes binary concatenation.

Cryptography We report the definition of a pseudorandom generator (PRG) and the abstract NIST construction frame-
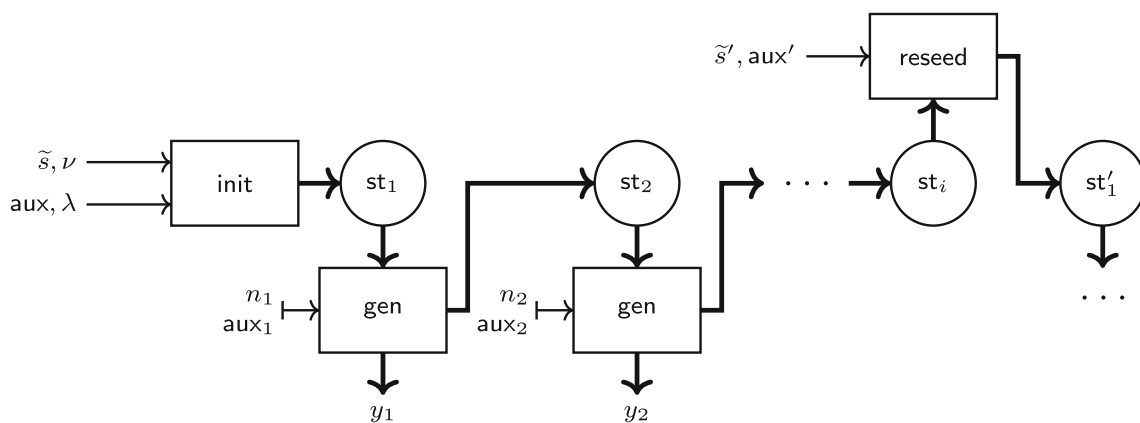
Fig. 1 A state machine representation of the NIST DRBG work flow

work for a DRBG. For readability, we omit the error handling of these constructions.

Definition 2.1 (*PRG* [19]) Given the positive integers $\ell_{\text{in}}, \ell_{\text{out}} \in \mathbb{N}$ with $\ell_{\text{out}} > \ell_{\text{in}}$, let $\mathsf{G} : \{0,1\}^{\ell_{\text{in}}} \rightarrow \{0,1\}^{\ell_{\text{out}}}$ be a deterministic function. We say that G is a **pseudorandom generator** if the following two distributions are computationally indistinguishable for a distinguisher D:

– Sample a random seed $\mathsf{s} \leftarrow_{\$} \{0,1\}^{\ell_{\text{in}}}$ and output $\mathsf{G}(\mathsf{s})$.
– Sample a random string $\mathsf{r} \leftarrow_{\$} \{0,1\}^{\ell_{\text{out}}}$ and output r.

Definition 2.2 (*Abstract NIST DRBG*) Let $\lambda \in \mathbb{N}$ be the security parameter, $\widetilde{s} \in \{0,1\}^{\lambda}$ a bit string obtained by a random source, $\ell_s \in \mathbb{N}$ the seed length and $\ell_r \in \mathbb{N}$ the number of iterations before requiring the seed's reseed. We define a seed $\widetilde{s} \in \{0,1\}^{\ell_s}$, a nonce $\nu \in \{0,1\}^{\lambda}$ and an auxiliary string $\mathsf{aux} \in \{0,1\}^*$. Let a NIST abstract DRBG be defined by the algorithms:

– $\mathsf{init}(\widetilde{s}, \nu, \mathsf{aux}, \lambda) \rightarrow \mathsf{st}_1$: given a random binary string $\widetilde{s}$, a nonce ν, an auxiliary string aux and the security parameter λ, the instantiation algorithm outputs the initial internal stage st_1.
– $\mathsf{reseed}(\mathsf{st}, \widetilde{s}', \mathsf{aux}) \rightarrow \mathsf{st}_1'$: given an internal state st, a random binary string $\widetilde{s}'$, an auxiliary binary string aux, the reseeding algorithm outputs a fresh initial internal stage st_1'.
– $\mathsf{gen}(\mathsf{st}_i, n, \mathsf{aux}) \rightarrow (y, \mathsf{st}_{i+1})$: given the internal state st_i, a non-zero number of output bit $n \in \mathbb{N}$ and an auxiliary string aux, the generation algorithm outputs the pseudorandom bit-string $y \in \{0,1\}^n$ and the successive internal stage st_{i+1}.

The DRBG is defined as a state machine, and it is depicted in Fig. 1. It takes a random binary string $\widetilde{s}$, a nonce ν, an auxiliary string aux and the security parameter λ to initialise the internal state and generates the internal state st_1. The internal state st_i is used as input of subsequent updates together with a nonzero number $n \in \mathbb{N}$ indicating the number of random bits requested and an auxiliary string aux. It outputs a n random bit-string y and updates the internal state to the next state st_{i+1}. Whenever it is requested, the DRBG can be reseeded, *i.e.* it starts again from a new internal state producing a new st_1' given a previous state st_i, a new random binary string $\widetilde{s}'$ and some auxiliary information aux'.

To correctly instantiate the DRBG, the NIST suggests three different constructions: (1) a *hash* function; (2) the HMAC of a hash function and (3) a *block cipher in counter-mode*. NIST requires the use of recommended cryptographic primitives [4], *e.g.* HMAC with a secure hash function, AES-128 or SHA-2 family, **and** a bit string obtained by a secure random source [3,24]. Whenever it is not specified, we always consider NIST approved primitives and security parameters.

Machine learning Roughly speaking, ML is a set of algorithms whose goal is finding and describing patterns over a dataset. The dataset is usually composed of independent *instances* each one defined by a set of *features* or *attributes*. Once the dataset is generated, it is used as input of the ML that produces the knowledge that has been learnt [1].

There are for main types of learning: (i) supervised learning or classification; (ii) unsupervised learning or clustering; (iii) association; and (iv) numeric prediction [27]. In supervised learning, the ML learns from an already labelled dataset and it tries to predict the class of a new instance. On the contrary, in unsupervised learning, the dataset is not labelled and the ML algorithm looks for common patterns based on heuristics. Association seeks for relationships between the features of the dataset, whereas the goal of the numeric prediction learning algorithms is to predict numbers instead of (labelled) data.

Naïve Bayes. The intuition behind Naïve Bayes is that features are independent and equally important. This is the consequence of applying the Bayes theorem into a classifi-

cation algorithm. There is a particular case of Naïve Bayes algorithm when the likelihood of the features follows a Gaussian distribution, *i.e.* when the (continuous) values associated with each feature are distributed according to a Gaussian distribution.

3 Machine learning distinguishers

In this section, we formally define the distinguishing problem and present our methodology which explains how ML can be used to solve a distinguishing problem. We discuss how to use the accuracy we obtain from ML as a cryptographic advantage, a curious phenomenon we call *"blind spot paradox"*, and propose a generic methodology to increase the advantage of a distinguisher at the cost of generating multiple ones.

In cryptography, it is common to find security properties defined by the probability of an adversary $\mathcal{A}$ being able to distinguish between **two** different instances. For example, in a simulation-based proof, $\mathcal{A}$ must discriminate between a real execution of a protocol and an ideal functionality assumed to be secure. Whenever proving the pseudorandomness of a function, $\mathcal{A}$ must choose if a value is computed by the function or if it is randomly sampled.

Definition 3.1 (*Distinguish problem*) Let G_0 and G_1 be two classes, $b \leftarrow_\$ \{0, 1\}$ a random coin flip and y an element of G_b. Consider a distinguisher D that takes as input y and outputs a guess b'. We define the **distinguish problem** as D's task in discriminating the membership of the value $y \in G_b$ between the two classes (G_0, G_1) and with advantage[2]:

$$\mathsf{Adv}^D_{G_0, G_1} = \left| 2 \cdot \Pr[D(y) = b_i] - 1 \right| \tag{3.1}$$

Even though the abstract definition form, the distinguishing problem appears as the core concept behind many important cryptographic security problems: *pseudorandomness* is defined as a distinguishing problem between a primitive G and a real random process; in an *indistinguishable cipher plaintext attacks*, it is required to distinguish a ciphertext between two possible messages, and the *cipher suite problem* requires to discriminate between different primitives (G_0, G_1).

3.1 Our methodology: from classifiers to distinguishers

Our methodology, depicted in Fig. 2, is based on the idea that a supervised learning algorithm can be used by an adversary $\mathcal{A}$ to create a distinguisher D between two classes

(G_0, G_1). We must observe that a supervised learning algorithm requires an input of a *labelled dataset* of correctly classified values (y_i, G_{b_i}), such that $y_i \in G_{b_i}$, that are used to define the classifier. Our methodology assumes that an adversary $\mathcal{A}$ can pre-compute any labelled *simulated* training dataset, *i.e.* $\mathcal{A}$ can easily compute *different but related* instances of (G_0, G_1), *e.g.* by sampling a different secret key. In this way, $\mathcal{A}$ can simulate arbitrarily labelled datasets which might not refer to the original problem instance between (G_0, G_1) but are somehow related, and thus, we consider them as *correct*.

The output of the algorithm is a classifier D that works exactly as a distinguisher, *i.e.* provided an element y, it guesses whether y belongs to G_0 or G_1. The next step is to consistently evaluate the *accuracy* that this distinguisher obtains. For the sake of simplicity, in this paper, we consider the **classifier accuracy** as the probability of correctly guessing the class for every element of a **target dataset** Y. However, other mechanisms can also be used to evaluate the accuracy like computing the confusion matrix and cross-validate the obtained results. We formally[3] define the accuracy as:

$$\mathsf{Acc}^D_{G_0, G_1}(Y) = \Pr_{y_i \in Y}\left[D(y_i) = G_{b_i}\right]$$

Observe that the distinguisher's accuracy highly depends on the target dataset Y. This implies that the accuracy computed by a distinguisher generated by our methodology **is not** directly related to the distinguisher's advantage previously described in the distinguishing problem of Definition 3.1. The reason is that the accuracy is computed over a target subset Y, which is generally much smaller than the set $\mathbb{Y}$ of all the possible elements. In other words, it is not possible to compare the accuracy $\Pr_{y_i \in Y}[D(y_i) = b_i]$ and the probability $\Pr_{y_i \in \mathbb{Y}}[D(y_i) = b_i]$ because the target Y might not be *representative* of the whole space $\mathbb{Y}$, *i.e.* Y might, for example, only contain *"easy to classify"* elements providing therefore a high accuracy for D though it might have no cryptographic advantage.

Roughly speaking, the accuracy can be seen as a statistical estimator of the advantage Adv^D meaning that there is a strong conceptual gap between theoretical and empirical results. However, it is possible to estimate both the dimensions and the number of samples needed to achieve a **statistically relevant** distinguisher, *e.g.* by verifying some accuracy properties with an appropriate statistical test and later evaluate the *power analysis* to confirm/evaluate the amount of sample needed to reach statistic relevance.

For the rest of the paper, we assume that there is always a way to correctly generate statistically relevant distinguishers

---

[2] We omit to specify the classes, *i.e.* Adv^D, when they are clear by the context.

[3] We will omit to specify the classes whenever they are clear by the context.

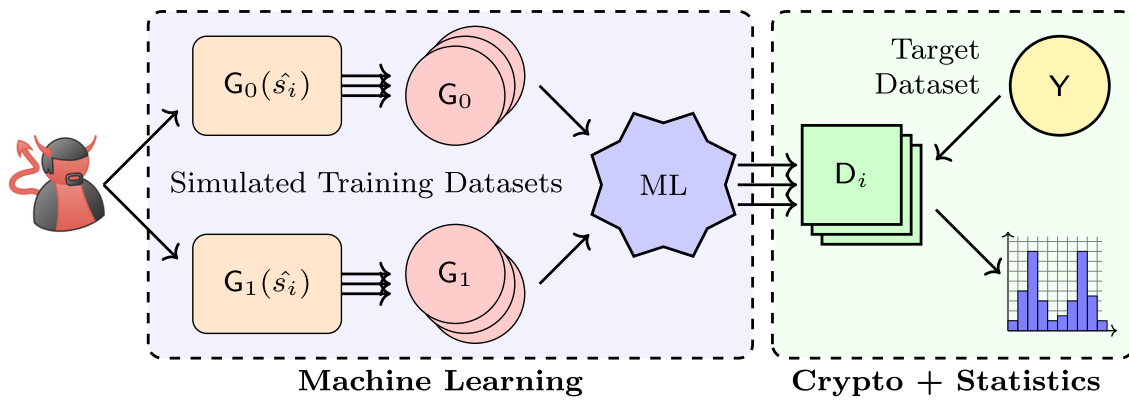**Fig. 2** Abstract representation of our methodology

D_i for any pair of classes (G_0, G_1). Furthermore, we refer to D's advantage as:

$$\mathsf{Adv}^{\mathsf{D}}_{\mathsf{G}_0,\mathsf{G}_1}(\mathsf{Y}) = \left| 2 \cdot \mathsf{Acc}^{\mathsf{D}}_{\mathsf{G}_0,\mathsf{G}_1}(\mathsf{Y}) - 1 \right|$$

Note that, whenever it is possible, the adversary $\mathcal{A}$ can generate many different training datasets, thus obtaining a set of n distinguishers $\{D_i\}_{i=1}^{n}$ each having its own accuracy $\mathsf{Acc}^{\mathsf{D}_i}_{\mathsf{G}_0,\mathsf{G}_1}(\mathsf{Y})$. By correctly analysing the accuracy's distribution, $\mathcal{A}$ can consider different attack *strategies*. Let us explain this concept with an example. Suppose that all the distinguishers generated by $\mathcal{A}$ have the same accuracy of 0.5. This means that $\mathcal{A}$ has no advantage and therefore must abandon the idea of solving the distinguishing problem. Differently, if $\mathcal{A}$ observes that a distinguisher D_i has an accuracy $0.5 - \delta$ for some positive $\delta \in \mathbb{R}_+$, $\mathcal{A}$ can invert D_i's output to define a new distinguisher D_i' with accuracy $0.5 + \delta$. In this case, $\mathcal{A}$ can transform distinguishers with an advantage in making wrong guesses into distinguishers that make correct guesses with the same advantage.

In summary, our methodology allows an adversary $\mathcal{A}$ to produce ML generated distinguishers if $\mathcal{A}$ can:

(i) pre-compute labelled simulated training datasets;
(ii) obtain statistically relevant target datasets, and;
(iii) run appropriate tests to evaluate the accuracy.

Consider an adversary $\mathcal{A}$ that, after executing our methodology, obtains several distinguishers of which she **does not** know the accuracy distribution. Despite the odd requirement, observe that this is the standard in practice since, to compute the accuracy distribution, it is required to obtain a *correct* target dataset which might not be obtainable, *e.g.* a primitive's security might be defined as a distinguisher problem where the adversary cannot query the correct primitive instantiation, thus not allowing $\mathcal{A}$ to get any target dataset.
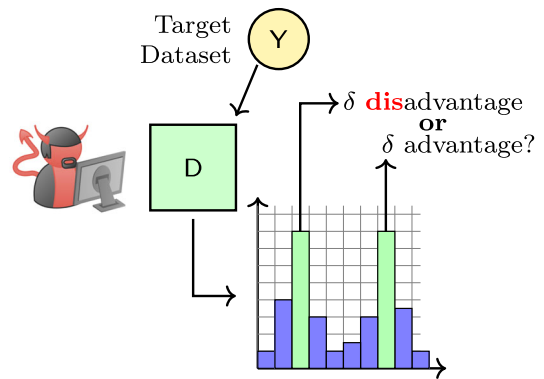

Fig. 3 Representation of the blind spot paradox

The **blind spot paradox**, depicted in Fig. 3, is the paradoxical phenomenon where a *blind* adversary $\mathcal{A}$ that does not know whether a specific distinguisher has an advantage or not is unable to *spot* how to correctly utilise the results, thus annihilating any advantage possessed. This paradox arises naturally whenever the accuracy is distributed symmetrically with respect to the probability of 0.5. Consider a distinguisher D and observe that, without any precise knowledge, it is impossible to know if D has a potential advantage δ or $-\delta$. The symmetric accuracy's distribution property implies that the probability of D being a *"good"* or a *"bad"* distinguisher is the same. For this reason, $\mathcal{A}$ is unable to properly utilise the potential advantage obtained, thus giving rise to the paradox. To avoid the paradox, it is necessary to allow the adversary to receive *"hints"* in the form of a statistically relevant list of target's outputs correctly classified. In this way, the adversary can get an estimation of the accuracy distribution and use this information to *"filter out the bad"* distinguishers. This completely breaks the symmetry of the distinguishers and allows them to use the *"good"* distinguishers. Of course, these hints might not be allowed by some theoretical security's properties **but** might better represent a realistic usage of such property.

3.2 Distinguisher accuracy amplification

In this section, we propose a generalisation method to combine and amplify the advantage of several independent distinguishers into a more accurate one by assuming that all the distinguishers have the same accuracy. The underlying reasoning still holds even when considering different accuracy's distribution assumptions.

Let us assume we have n distinguishers $\{D_i\}_{i=1}^n$, between classes (G_0, G_1), all with the same accuracy $p > 0.5$. We require the distinguishers to be *independent* in the sense that they are generated from *different and independent training sets*. Our goal is to consider the *majority* of all the n distinguisher's guesses. In order to always have a majority, we must assume that n is odd, *i.e.* there exists $k \in \mathbb{N}$ such that $2k + 1 = n$.

Proposition 3.1 *Let* $k \in \mathbb{N}$, $0.5 < p < 1$, $n = 2k + 1$ *and* $\{D_i\}_{i=1}^n$ *be independent distinguishers with accuracy* p. *We define the distinguisher* D' *as the majority function of the* n *independent* D_i *guesses. Formally,* $D'(y) = \mathrm{maj}\,(D_1(y), \ldots, D_n(y))$. *Then, it holds that* D' *has an accuracy* p_k *greater than* p.

Proof Note that the distinguisher' outputs define a binomial distribution of parameters p and n where the probability of "t distinguishers are correct" is:

$$\Pr\,[t \text{ are correct, }] = \binom{2k+1}{t} \cdot (1-p)^{2k+1-t} \cdot p^t$$

The final guess of D' is defined by at least $k+1$ distinguishers that have the same guess. This implies that the accuracy of D' directly depends on p and n. Formally, the probability of correctly guessing the distinguishing game for D', with $q = (1-p)$, is:

$$p_k = \Pr\,[D' \text{correct}] = \Pr\left[\begin{array}{c} \geq k+1 \\ D_i \text{correct} \end{array}\right]$$
$$= \sum_{t=k+1}^{2k+1} \binom{2k+1}{t} \cdot q^{2k+1-t} \cdot p^t$$

Let us recall the binomial identities $\binom{j}{k} = \binom{j-1}{k} + \binom{j-1}{k-1}$ and $\binom{2k-1}{t} = 0$ whenever $t > 2k - 1$. Let us define p_0 to be exactly p. Our goal is to consider the probability p_k and obtain a relation with respect to p_{k-1}. Then, it holds that:

$$p_k = \sum_{t=k+1}^{2k+1} \binom{2k+1}{t} \cdot q^{2k+1-t} \cdot p^t$$
$$= \sum_{t=k+1}^{2k+1} \left(\binom{2k-1}{t} + 2 \cdot \binom{2k-1}{t-1} + \binom{2k-1}{t-2}\right) \cdot$$

$$\cdot q^{2k+1-t} \cdot p^t$$
$$= \sum_{t=k+1}^{2k+1} \binom{2k-1}{t} \cdot q^{2k+1-t} \cdot p^t$$
$$+ 2 \sum_{t=k+1}^{2k+1} \binom{2k-1}{t-1} \cdot q^{2k+1-t} \cdot p^t$$
$$+ \sum_{t=k+1}^{2k+1} \binom{2k-1}{t-2} \cdot q^{2k+1-t} \cdot p^t \quad (3.2)$$

Let us take a look at the addend and observe that it can be rewritten as:

$$\sum_{t=k+1}^{2k+1} \binom{2k-1}{t} \cdot q^{2k+1-p} \cdot p^t$$
$$= q^2 \cdot \sum_{t=k+1}^{2k-1} \binom{2k-1}{t} \cdot q^{2k-1-t} \cdot p^t$$
$$= q^2 \cdot \left(\sum_{t=k}^{2k-1} \binom{2k-1}{t} \cdot q^{2k-1-t} \cdot p^t\right)$$
$$- q^2 \binom{2k-1}{k} \cdot q^{k-1} \cdot p^k$$
$$= q^2 \cdot p_{k-1} - \binom{2k-1}{k} \cdot q^{k+1} \cdot p^k \quad (3.3)$$

where we note the presence of a relation to the winning probability p_{k-1}. Similarly, we manipulate the second and third addends and obtain:

$$2 \sum_{t=k+1}^{2k+1} \binom{2k-1}{t-1} \cdot q^{2k+1-t} \cdot p^t$$
$$= 2 \cdot p \cdot q \cdot \sum_{t=k+1}^{2k} \binom{2k-1}{t-1} \cdot q^{2k-t} \cdot p^{t-1}$$
$$= 2 \cdot p \cdot q \cdot \sum_{t=k}^{2k-1} \binom{2k-1}{t} \cdot q^{2k-1-t} \cdot p^t$$
$$= 2 \cdot p \cdot q \cdot p_{k-1} \quad (3.4)$$

$$\sum_{t=k+1}^{2k+1} \binom{2k-1}{t-2} \cdot q^{2k+1-t} \cdot p^t$$
$$= p^2 \cdot \sum_{t=k+1}^{2k+1} \binom{2k-1}{t-2} \cdot q^{2k+1-t} \cdot p^{t-2}$$
$$= p^2 \cdot \sum_{t=k}^{2k-1} \binom{2k-1}{t} \cdot q^{2k-1-t} \cdot p^t +$$
$$+ p^2 \cdot \binom{2k-1}{k-1} \cdot q^k \cdot p^{k-1}$$

$$= p^2 \cdot p_{k-1} + \binom{2k-1}{k-1} \cdot q^k \cdot p^{k+1}$$

$$= p^2 \cdot p_{k-1} + \binom{2k-1}{k} \cdot q^k \cdot p^{k+1} \qquad (3.5)$$

where we used the fact that:

$$\binom{2k-1}{k-1} = \frac{(2k-1)!}{(k-1)! \cdot k!} = \binom{2k-1}{k}$$

By putting together Equations 3.3, 3.4, 3.5 into Equation 3.2, it holds that:

$$p_k = p_{k-1}\left(q^2 + 2qp + p^2\right) + \binom{2k-1}{k} \cdot q^k \cdot p^{k+1}$$
$$\quad - \binom{2k-1}{k} \cdot q^{k+1} \cdot p^k$$
$$= p_{k-1}(q+p)^2 + \binom{2k-1}{k} \cdot (q \cdot p)^k \cdot (p - q)$$
$$= p_{k-1} + \binom{2k-1}{k} \cdot (q \cdot p)^k \cdot (2p - 1)$$

from which we observe that $p_k > p_{k-1}$ whenever:

$$p_k > p_{k-1} \Leftrightarrow \binom{2k-1}{k} \cdot (q \cdot p)^k \cdot (2p - 1) > 0$$
$$\Leftrightarrow (2p - 1) > 0 \iff p > \frac{1}{2}$$

which is true by our hypothesis. The distinguisher D' built with $2k + 1$ distinguisher has an accuracy $p_k > p_{k-1} > \cdots > p_0 = p$, concluding our proof. $\qquad \square$

4 Case study: cipher suite distinguisher for pseudorandom generators

In this section, we implement our methodology into the MLCrypto tool which we use to create distinguishers for NIST DRBG. We also discuss the connection between our empirical results and the constraints posed by a possible real attack against the primitives.

Let us consider a PRG $G : \{0, 1\}^{\ell_{in}} \to \{0, 1\}^{\ell_{out}}$, as in Definition 2.1, and focus on the *pseudorandomness* property. Such a property states the indistinguishability between the distributions of the G's outputs and the uniformly random elements. By using the game-proving framework, it is required that any distinguisher D is unable to distinguish between a random value and G's output when provided by the challenger. Formally, we define the advantage as:

$$\mathsf{Adv}^D_{G,\text{rand}}(\lambda) = |\Pr[D(G(s)) = G] - \Pr[D(r) = G]|$$

for some random seed $s \leftarrow_\$ \{0, 1\}^{\ell_{in}}$, uniformly sampled $r \leftarrow_\$ \{0, 1\}^{\ell_{out}}$. The theoretical approach is conceptually simple and tight but infeasible because it requires a function that outputs random elements, which is, by other terms, precisely what the PRG tries to emulate, thus creating a brain-twisting loophole in which the goal is the solution at the same time.

To avoid this loophole, we can use a statistical approach, which consists of running several statistical tests using the outputs of G. After running G, the tests compare the real and the theoretical distributions to accept/reject the hypothesis that G is random or not. There are several statistical test suites to analyse the PRG such as NIST STS [5], Dieharder [7] and TestU01 [21].

Let us explain the approach with an example. Consider a list of N outputs $\{y_i\}_{i=1}^N$ from a pseudorandom G of which we want to determine if they appear random. To do so, consider the statistical test that shows the frequency of 1s in the output, *i.e.* it returns the number of 1s in a given output binary string.

Theoretically, we know that the output should describe the binomial distribution of which we know the *characteristic function*, *i.e.* the function that describes the probability distribution. For this reason, we apply the test on the set of outputs $\{y_i\}_{i=1}^N$ and compare it with the theoretical binomial ones thus testing if the outputs are *"binomial enough"*. In Fig. 4, we illustrate the possible outcomes of the test where we compare the ideal distribution *(b)* with respect to a fitting *(c)* and a completely random one *(a)*.

The tests take an analytical approach by computing precise values, *e.g.* the p-value for some specific statistical test. By repeating the test multiple times, it is possible to improve the *confidence* of the result. Sadly, regardless of the number of different tests we can perform and analyse, this approach can only state if a generator is plausibly pseudorandom or not.

On the other hand, the statistical approach allows the direct construction of a distinguisher D for the general pseudorandomness property, *i.e.* D executes the statistical test on the given output and uses the test results to discriminate between pseudorandom and non-pseudorandom. A failing test result allows D to have an advantage in discriminate non-pseudorandom PRG.

Let us take a step back and observe that the pseudorandom property can be modified into a *cipher suite distinguishing problem* in which a distinguisher D must distinguish between **two** different generators G_0 and G_1, regardless of their pseudorandom properties. By arithmetic manipulation of Equation 3.1, we obtain:

$$\mathsf{Adv}^D_{G_0,G_1}(\lambda)$$
$$= \left| \Pr[D(G_1(s)) = G_1] - \Pr[D(G_0(s)) = G_1] \right|$$
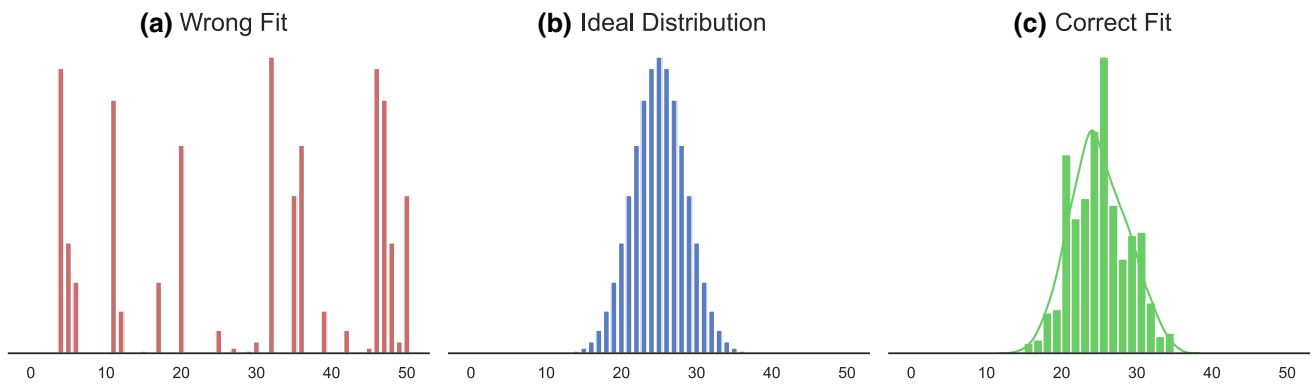$$= \left| \Pr[D(G_1(s)) = G_1] - \Pr[D(r) = G_1] \right|$$

(a) Wrong Fit   **(b)** Ideal Distribution   **(c)** Correct Fit

Fig. 4 Example of distribution fitting with respect to an ideal binomial distribution

$$+ \Pr\left[D(r) = G_1\right] - \Pr\left[D(G_0(s)) = G_1\right] \Big|$$

$$\leq \Big| \Pr\left[D(G_1(s)) = G_1\right] - \Pr\left[D(r) = G_1\right] \Big|$$

$$+ \Big| \Pr\left[D(r) = G_1\right] - \Pr\left[D(G_0(s)) = G_1\right] \Big|$$

$$\leq \mathsf{Adv}^D_{G_0},\mathsf{rand}(\lambda) + \mathsf{Adv}^D_{\mathsf{rand},G_1}(\lambda)$$

where the second addend:

$$\Big| \Pr\left[D(r) = G_1\right] - \Pr\left[D(G_0(s)) = G_1\right] \Big| \leq \mathsf{Adv}^D_{\mathsf{rand},G_1}(\lambda)$$

measures the probability of D to wrongly distinguishing G_0. By the nature of the absolute value, we can modify this faulty distinguisher into a correct one by just flipping D's output. The idea behind our observation is that, by triangular disequality, distinguishing between two generators imposes a lower bound on the generator's pseudorandomness advantage. Formally:

$$\mathsf{Adv}^D_{G_0,G_1}(\lambda) \leq \mathsf{Adv}^D_{G_0,\mathsf{rand}}(\lambda) + \mathsf{Adv}^D_{\mathsf{rand},G_1}(\lambda) \quad (4.1)$$

Since executing the cryptanalysis necessary to create D is tedious, time-consuming and a human-intensive task, we use MLCrypto to automatically generate D from different NIST DRBG outputs.

4.1 Experiments and results

In this section, we analyse the distinguishers generated by MLCrypto for the cipher suite distinguishing problem. Concretely, we focus on the DRBG that NIST recommends [4]. Also, all the experiments we present in this section were run on an Intel(R) Core(TM) i7-4790 CPU @3.60GHz and 16GB of RAM with Linux. We implement MLCrypto in Python, and

all the source code of our tool is freely released for future research[4].

For this experiment, we consider the NIST DRBG based on the primitives TDEA, AES-256, SHA-256 and HMAC-SHA-256. The choice of these DRBG is arbitrary, and if other primitives were chosen, the conclusions remain the same.

For all the experiments, there is a common initial phase where we calculate all possible pairs of combinations $(\mathsf{alg}_0, \mathsf{alg}_1)$ of the primitives and we accordingly generate the training and target datasets. For the training dataset, we want to simulate an adversary who cannot create such a dataset with the same seed as the target. Thus, all the training datasets have different seeds than the targets ones. In our case study, we analyse if the distribution of the accuracy of the distinguishers generated by MLCrypto (see Sect. 3) is affected by (i) the size of the datasets (training and target) and (ii) different target dataset.

The reason why we chose Naive Bayes classifiers for MLCrypto is that they are (i) computationally efficient, (ii) simple to implement and (iii)lack of learning parameter to be tuned.

Dataset size To cross-validate our ML classifiers, we check if the size of the datasets affects the output of the distinguisher. To do so, we generate for each primitive alg a training dataset X_{alg} containing n_X outputs of alg and a target dataset Y_{alg} containing n_Y outputs of alg. In more detail, the size of the training (n_X) and the target (n_Y) datasets are $n_X \in \{2^i : i \in [12, 14]\}$ and $n_Y \in \{2^i : i \in [14, 16]\}$, respectively. The datasets generation is computationally efficient, and the size average with 2^{16} values is ~ 1.1 MB. We independently execute MLCrypto t_X times with a freshly generated training dataset, say X', **but** with the same target dataset Y. Concretely, we consider $t_X = 2^{10}$ which would provide to compute a Cohen's coefficient of $d = 0.0876$ for
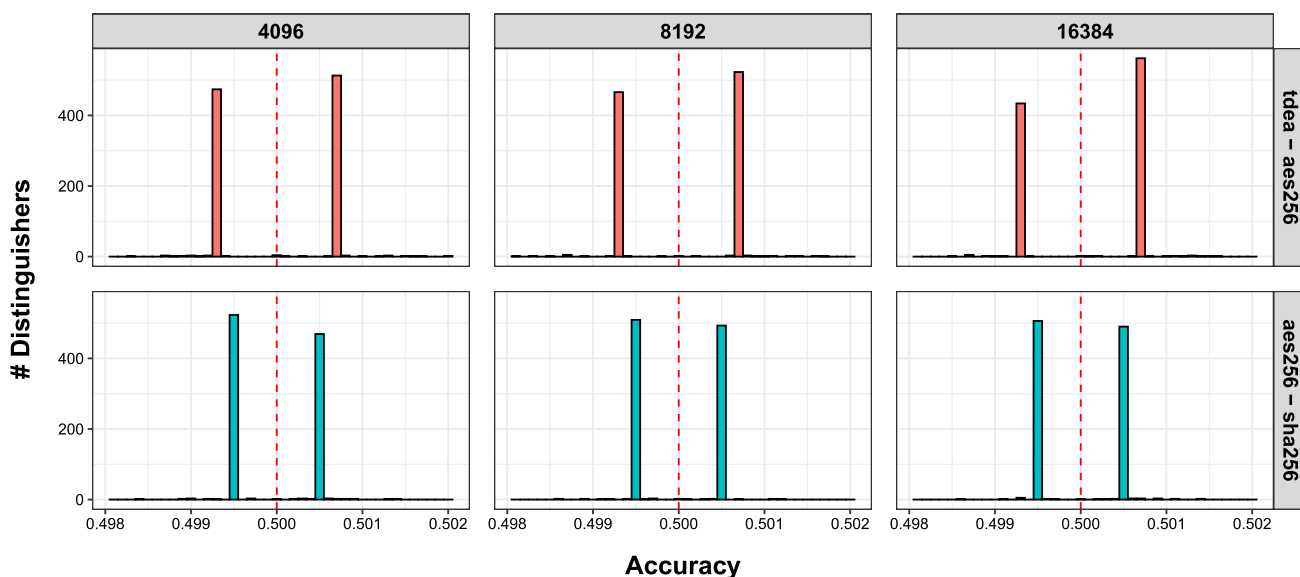
4 https://bitbucket.org/CharlieTrip/mlcryptocode/src/master/.

Fig. 5 Distinguishers' accuracy distributions of two arbitrary primitives computed for 3 different training dataset sizes
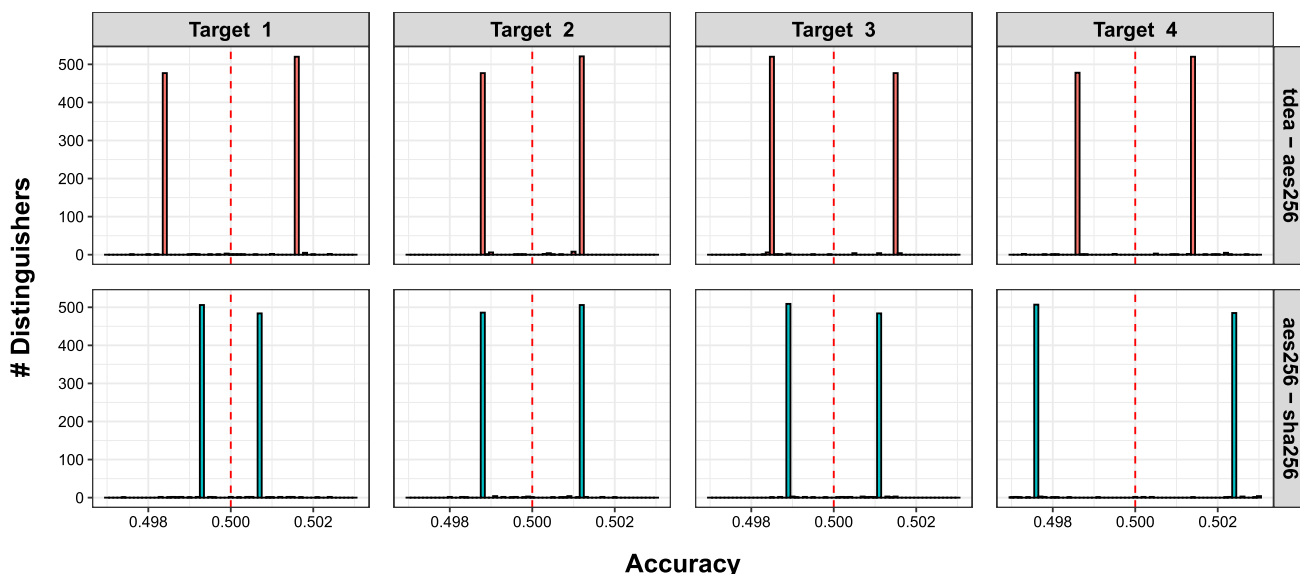

Fig. 6 Distinguishers' accuracy distributions of two arbitrary primitives computed for 4 different target datasets

a statistical power of $p = 0.8$, whenever analysing the distinguishers' accuracy distribution with a one-sample t-Student test with significance level $\alpha = 0.5$. In other words, the size of our datasets, as well as the number of tests, provides a (simplistic) statistical analysis that the obtained classifiers accuracy's distribution has some statistical confidence. In Fig. 5, we observe that changing the training dataset size n_X does not have any major impact on the accuracy distribution. This suggests that it is possible to provide smaller training datasets and still achieving the same accuracy distribution. Finally, we also checked our model's ability to predict new data (*i.e.* avoid overfitting or selection bias); we obtained the cross-validation value of each one of the experi-

ments we performed. In more detail, we computed the 10-fold cross-validation using the function provided by *scikit-learn* and got a consistent accuracy in all our independent experiments.

Different targets We generate the training datasets of such primitives and obtain a distinguisher D for the algorithms (alg_0, alg_1). Once we have D, we randomly generate a target dataset and compute the accuracy of the distinguisher as $\mathsf{Acc}^{\mathsf{D}}_{\mathsf{alg}_0,\mathsf{alg}_1}$. Figure 6 depicts that the same distinguishers define different accuracy distributions when computed on different target datasets. This phenomenon is explained by the fact that each target dataset is generated using a *different seed* thus making the generator *de facto* different. This
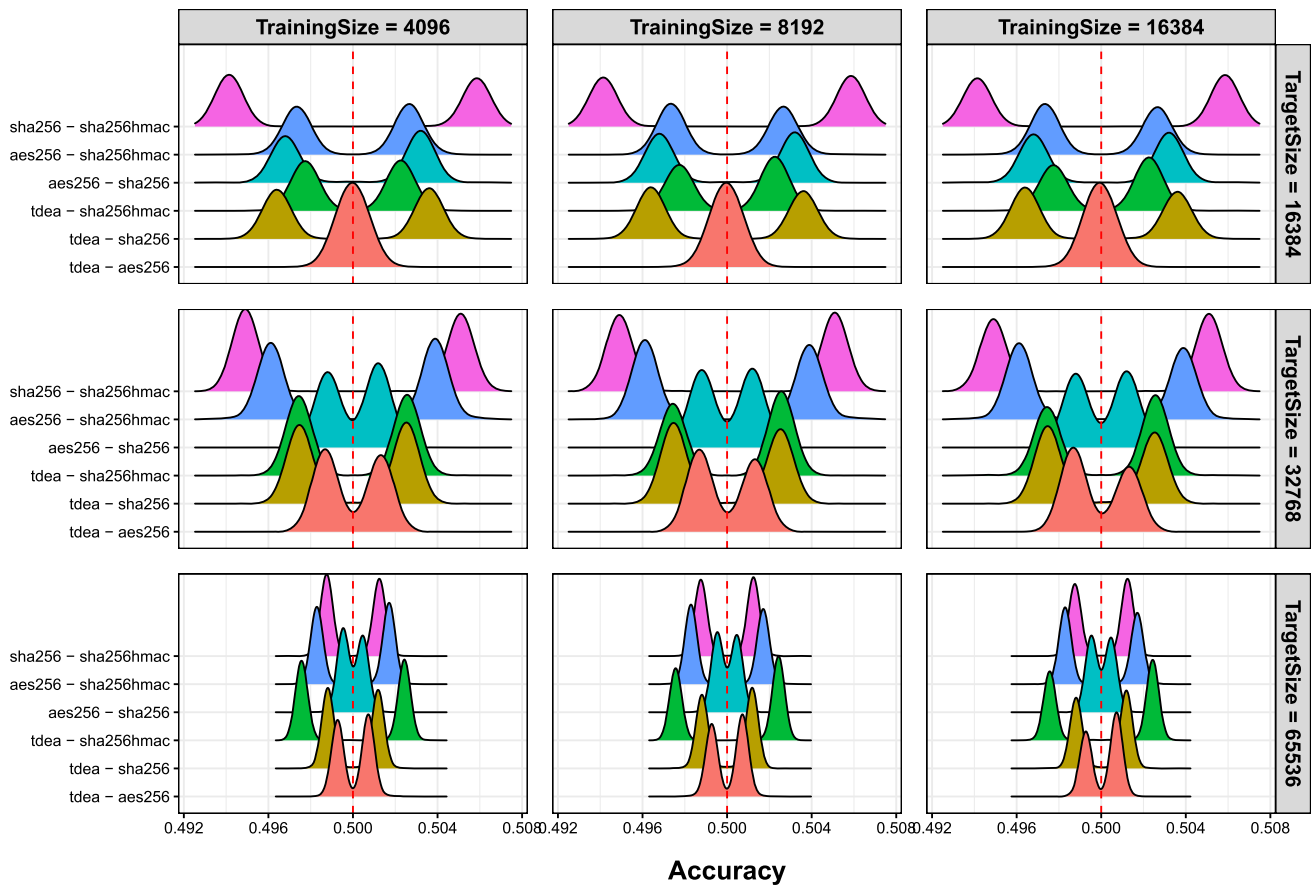
Fig. 7 Distinguishers' accuracy distributions of the combination between the primitives in alg. We compute the distributions for 3 target dataset sizes and 3 training ones

implies that an increased accuracy advantage δ for a distinguisher D holds exclusively for a specific target. By changing the target, D changes the advantage to a different value δ'. We also consider a variation of n_Y and observe that the peaks are differently spread. This is coherent when considering that a smaller dataset X' is a sample of a bigger one X, meaning that X' might not be a statistically significant representation of X. This implies the necessity of always using statistically significant target datasets when computing the accuracy distribution.

Timing and space efficiency In total, we generate $4 \cdot (1 + t_X) = 4100$ independent datasets, being 4 the number of different primitives considered, and $\binom{4}{2} \cdot t_X \cdot 3 = 18432$ distinguishers, being 3 the distinct n_X possible values. Each distinguisher outputs 3 values, being 3 the number of distinct n_Y possible values of a total of 55296 measurements. In Fig. 7, we show how the accuracy of the distinguishers is always distributed with either a single peak centred in 0.5 or as two symmetric peaks at value $0.5 \pm \delta$ for some non-negligible $\delta \in \mathbb{R}_+$ of the order of $\delta \sim 10^{-3}$. This demonstrates that MLCrypto can create a distinguisher D with advantage $\mathsf{Adv}^D = 2\delta$. Even

though that δ might initially be small when we consider only the distinguishers with accuracy $0.5 + \delta$, we apply the distinguisher' amplification method presented in Proposition 3.1 to increase up that advantage. For instance, in this case we have $\frac{t_X}{2} - 1 = 511$ distinguishers with an accuracy of $p \sim 50.1\%$ which implies that the amplification method creates a distinguisher D' with accuracy $p' \sim 51.8\%$.

For completeness, we execute MLCrypto over all the NIST DRBG, with training datasets size $n_X = 2^{13}$ and target dataset size $n_Y = 2^{16}$ of which accuracy distributions are depicted in Fig. 8. We observe that the accuracy distribution is always symmetric. This means that a blind adversary $\mathcal{A}$ must face the blind spot paradox, allowing us to empirically confirm that NIST DRBG are, most probably, hard to distinguish between themselves. On the other hand, if $\mathcal{A}$ can reconstruct the distribution, then there is a concrete possibility to achieve a non-negligible advantage in distinguishing between the primitives.
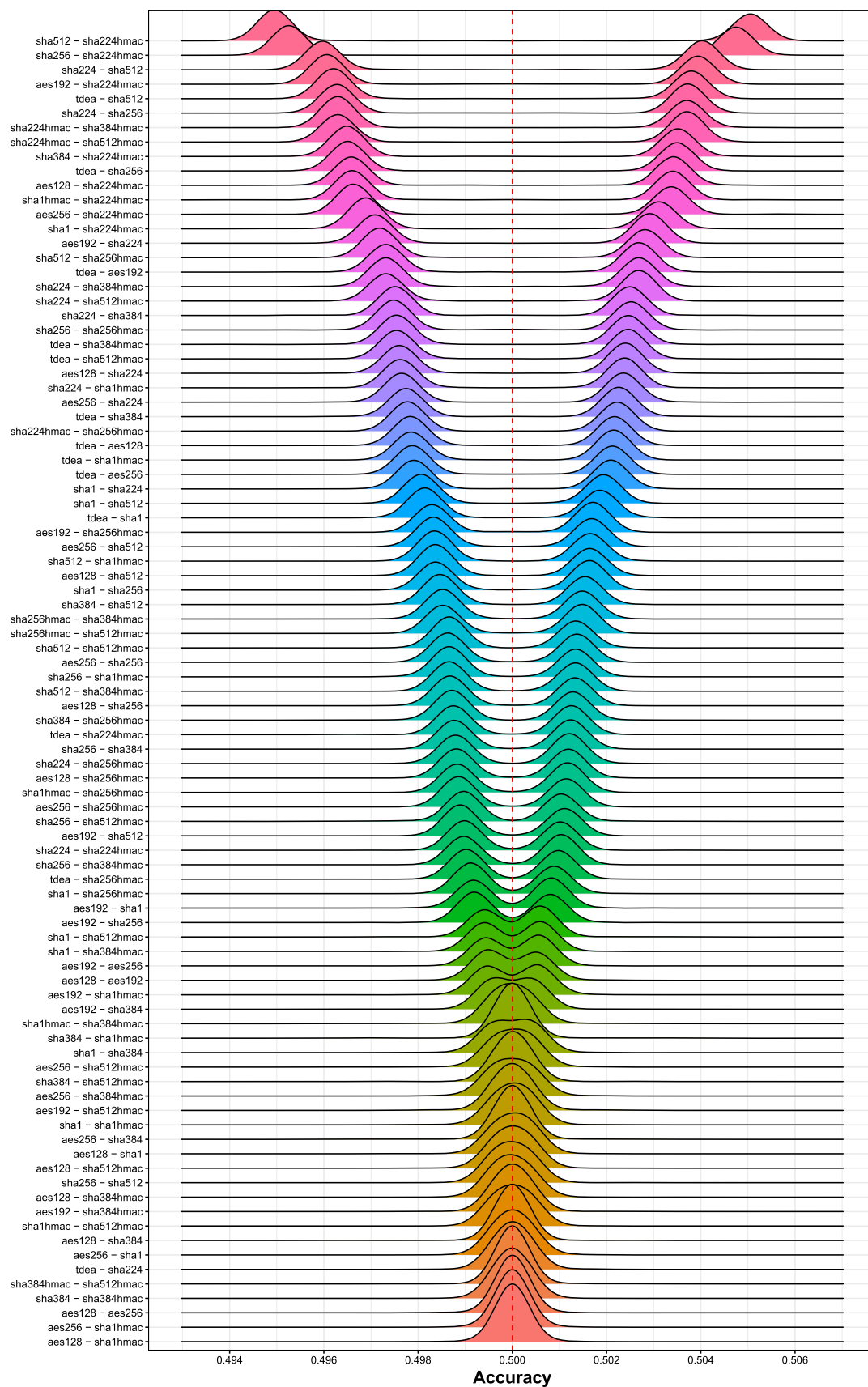
Fig. 8 Distinguishers' accuracy distribution of all the NIST recommended DRBG combinations with training dataset size $n_X = 2^{13}$ and target dataset size $n_Y = 2^{16}$

5 Conclusions and future work

In this paper, we presented a methodology to use ML in developing practical distinguisher for cryptographic purposes. In particular, we show how it can be used for solving and analysing instances of distinguishing problems, *e.g.* we analyse the distinguishers obtained by MLCrypto for the cipher suite distinguishing problem between NIST DRBG. We foresaw the possibility of applying our tool to cipher suite distinguishing problems for block ciphers, hash functions, message authentication codes and similar primitives. The generality of our method allows it to be used for more practical problems related to *side-channel attacks* where the attacker is interested in distinguishing between two primitives based on non-cryptographic measurements, *e.g.* the power consumption and the computational timing, and provides a consistent framework for future comparison between distinguishers generated by different ML approaches, *e.g.* random forest, neural network or the multi-layers perceptron model [2].

Declarations

Conflict of interest The authors declare that they do not have conflict of interests.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

References

1. Alpaydin, E.: Introduction to Machine Learning, 3rd edn. (2014)
2. Baksi, A., Breier, J., Chen, Y., Dong, X.: Machine learning assisted differential distinguishers for lightweight ciphers (extended version) (2020)
3. Barker, E., Kelsey, J.: Recommendation for Random Bit Generator (RBG) Constructions. Technical report. National Institute of Standards and Technology (2016)
4. Barker, E.B., Kelsey, J.M.: Recommendation for Random Number Generation Using Deterministic Random Bit Generators. Technical report. NIST SP 800-90Ar1. National Institute of Standards and Technology (2015). https://doi.org/10.6028/NIST.SP.800-90Ar1
5. Bassham, L., Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, N., Dray, J.: A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. Technical report. National Institute of Standards and Technology (2010). https://doi.org/10.6028/NIST.SP.800-22r1a
6. Biham, E., Shamir, A.: Differential cryptanalysis of DES-like cryptosystems. J. Cryptol. (1991). https://doi.org/10.1007/BF00630563
7. Brown, R.G., Eddelbuettel, D., Bauer, D.: Dieharder: A Random Number Test Suite. Open Source Software Library (2013)
8. Cohney, S., Kwong, A., Paz, S., Genkin, D., Heninger, N., Ronen, E., Yarom, Y.: Pseudorandom black swans: cache attacks on CTR_DRBG. In: S&P (2020)
9. Dileep, A., Sekhar, C.: Identification of block ciphers using support vector machines. In: IEEE International Joint Conference on Neural Network Proceedings (2006). https://doi.org/10.1109/IJCNN.2006.247172
10. Fischer, T.: Testing cryptographically secure pseudo random number generators with artificial neural networks. TrustCom/BigDataSE (2018). https://doi.org/10.1109/TrustCom/BigDataSE.2018.00168
11. Gohr, A.: Improving attacks on round-reduced speck32/64 using deep learning. In: Advances in Cryptology—CRYPTO 2019 (2019). https://doi.org/10.1007/978-3-030-26951-7_6
12. González-Serrano, F.J., Amor-Martín, A., Casamayón-Antón, J.: Supervised machine learning using encrypted training data. Int. J. Inf. Secur. **17**(4), 365–377 (2018)
13. Hirose, S.: Security analysis of DRBG using HMAC in NIST SP 800–90. Inf. Secur. Appl. (2009). https://doi.org/10.1007/978-3-642-00306-6_21
14. Hospodar, G., Gierlichs, B., De Mulder, E., Verbauwhede, I., Vandewalle, J.: Machine learning in side-channel analysis: a first study. J. Cryptogr. Eng. (2011). https://doi.org/10.1007/s13389-011-0023-x
15. Hu, X., Zhao, Y.: Block ciphers classification based on random forest. J. Phys. Conf. Ser. (2019). https://doi.org/10.1088/1742-6596/1168/3/032015
16. Joux, A.: Algorithmic Cryptanalysis (2009)
17. Kant, S., Khan, S.S.: Analyzing a class of pseudo-random bit generator through inductive machine learning paradigm. Intell. Data Anal. **10**(6), 539–554 (2006)
18. Kant, S., Kumar, N., Gupta, S., Singhal, A., Dhasmana, R.: Impact of machine learning algorithms on analysis of stream ciphers. In: ICM2CS (2009). https://doi.org/10.1109/ICM2CS.2009.5397953
19. Katz, J., Lindell, Y.: Introduction to Modern Cryptography (2014)
20. Koza, J.: Evolving a computer program to generate random numbers using the genetic programming paradigm. In: International Conference on Genetic Algorithms (1991)
21. L'Ecuyer, P., Simard, R.: TestU01: a C library for empirical testing of random number generators. ACM Trans. Math. Softw. (2007). https://doi.org/10.1145/1268776.1268777
22. Peinado, A., Ortiz, A.: Prediction of sequences generated by LFSR using back propagation MLP. Sococisisiceute (2014). https://doi.org/10.1007/978-3-319-07995-0_40
23. Sipper, M., Tomassini, M.: Generating parallel random number generators by cellular programming. Int. J. Mod. Phys. C (1996). https://doi.org/10.1142/S012918319600017X
24. Sönmez Turan, M., Barker, E., Kelsey, J., McKay, K., Baish, M., Boyle, M.: Recommendation for the Entropy Sources Used for Random Bit Generation. Technical report. NIST Special Publica-

tion (SP) 800-90B, National Institute of Standards and Technology (2018). https://doi.org/10.6028/NIST.SP.800-90B

25. Souza, W.A.D., Tomlinson, A.: A distinguishing attack with a neural network. Data Min. Workshops (2013). https://doi.org/10.1109/ICDMW.2013.116

26. Svenda, P., Ukrop, M., Matyáš, V.: Towards cryptographic function distinguishers with evolutionary circuits. In: SECRYPT (2013)

27. Witten, I.H., Frank, E.: Data mining: practical machine learning tools and techniques with Java implementations. SIGMOD Rec. (2002). https://doi.org/10.1145/507338.507355

28. Woodage, J., Shumow, D.: An analysis of NIST SP 800–90A. EUROCRYPT (2019). https://doi.org/10.1007/978-3-030-17656-3_6

29. Zhao, Z., Zhao, Y., Liu, F.: The research of cryptosystem recognition based on randomness test's return value. Cloud Comput. Secur. (2018). https://doi.org/10.1007/978-3-030-00015-8_1