Is ML-Based Cryptanalysis Inherently Limited? Simulating Cryptographic Adversaries via Gradient-Based Methods

Avital Shafran* Eran Malach† Thomas Ristenpart‡ Gil Segev* Stefano Tessaro§

Abstract

Given the recent progress in machine learning (ML), the cryptography community has started exploring the applicability of ML methods to the design of new cryptanalytic approaches. While current empirical results show promise, the extent to which such methods may outperform classical cryptanalytic approaches is still somewhat unclear.

In this work, we initiate exploration of the theory of ML-based cryptanalytic techniques, in particular providing new results towards understanding whether they are fundamentally limited compared to traditional approaches. Whereas most classic cryptanalysis crucially relies on directly processing individual samples (e.g., plaintext-ciphertext pairs), modern ML methods thus far only interact with samples via gradient-based computations that average a loss function over all samples. It is, therefore, conceivable that such gradient-based methods are inherently weaker than classical approaches.

We introduce a unifying framework for capturing both "sample-based" adversaries that are provided with direct access to individual samples and "gradient-based" ones that are restricted to issuing gradient-based queries that are averaged over all given samples via a loss function. Within our framework, we establish a general feasibility result showing that any sample-based adversary can be simulated by a seemingly-weaker gradient-based one. Moreover, the simulation exhibits a nearly optimal overhead in terms of the gradient-based simulator's running time. Finally, we extend and refine our simulation technique to construct a gradient-based simulator that is fully parallelizable (crucial for avoiding an undesirable overhead for parallelizable cryptanalytic tasks), which is then used to construct a gradient-based simulator that executes the particular and highly useful gradient-descent method.

Taken together, although the extent to which ML methods may outperform classical cryptanalytic approaches is still somewhat unclear, our results indicate that such gradient-based methods are not inherently limited by their seemingly restricted access to the provided samples.

* School of Computer Science and Engineering, Hebrew University of Jerusalem, Jerusalem 91904, Israel. Email: `{avital.shafran,segev}@cs.huji.ac.il`. Supported by the Israel Science Foundation (Grant No. 1336/22), and by the European Union (ERC, FTRC, 101043243). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them. This work was performed while A. Shafran was visiting Cornell Tech, and was supported in part by NSF grant CNS-2120651.

† Kempner Institute for the Study of Natural and Artificial Intelligence, Harvard University, MA, USA. Email: `emalach@fas.harvard.edu`.

‡ Department of Computer Science, Cornell Tech, New York, USA. Email: `ristenpart@cornell.edu`. Supported in part by NSF grants CNS-2055169, CNS-2120651.

§ Paul G. Allen School of Computer Science & Engineering, University of Washington, Seattle, WA, USA. Email: `tessaro@cs.washington.edu`. Supported in part by NSF grants CNS-2026774, CNS-2154174, a JP Morgan Faculty Award, a CISCO Faculty Award, and a gift from Microsoft.

Contents

1 Introduction

The interplay between cryptography and learning theory has been extensively studied over the years, consistently leading to fundamental insights, both theoretical and practical. Traditionally, the study of this interplay has mostly focused on establishing a fruitful relationship between the hardness of cryptographic problems and that of learning problems (see, for example, [Val84, KV89, Riv91, Kha93, Reg05, KS09, ABW10, SZB21] and the references therein). Quite recently, however, following the tremendous progress in the area of machine learning, the cryptographic community has gradually started revealing an additional exciting avenue: Exploring the applicability of modern machine learning methods (in particular, those based on training deep neural networks, or DNNs) to the design of new cryptanalytic approaches that may hopefully improve upon the classic, somewhat "manual", cryptanalytic approaches.

ML-based cryptanalysis. ML-based cryptanalytic methods have so far been developed mostly for attacking block ciphers, starting with the work of Gohr [Goh19] focusing on differential cryptanalysis of SPECK [BSS+13], and for attacking the LWE problem, starting with the work of Wenger, Chen, Charton, and Lauter [WCC+22]. In both cases, essentially, the same high-level ML-based methodology was followed: First, a large sample set was used to train a DNN using a certain network architecture and learning algorithm. Then, the resulting DNN was challenged with respect to an appropriate key recovery task or an indistinguishability task. For example, the sample set used by Gohr consisted of samples $((C_1, C_2), y)$ corresponding to encryptions under a secret key of either two uniformly random plaintexts (when $y = 0$) or uniformly chosen plaintexts that have a fixed difference $\Delta = P \oplus P'$ (when $y = 1$).

Similarly, the sample set used by Wenger et al. consisted of a large number of independently generated LWE samples $(x, y = x \cdot s + e \bmod q)$ for a secret s and added noise e.

The work of Gohr and that of Wenger et al. motivated a large and growing amount of follow-up work, as we further discuss in Section 1.2. This includes additional applications of ML-based methods both in the context of block ciphers [JKM20, BGP+21, GLN22, BB22] and in the context of the LWE problem [LSW+23a, LSW+23b, SWL+24]. The extent to which these ML-based techniques outperform classic ones has been the subject of debate [DPS23], and further empiricism is warranted. We suggest complementing empiricism with theoretical treatments. To date, however, no formal frameworks have been suggested.

Sample-based vs. gradient-based methods. In this work, we initiate the exploration of the theoretical foundations of ML-based cryptanalysis, focusing on the question of whether learning-based approaches are *fundamentally* limited compared to others.

Motivating our exploration is the observation that modern ML-based methods fall into a specific template (as discussed above) that only access samples via gradient-based computations, which are averaged over all samples via a "loss function". In comparison, traditional approaches use full access to process samples directly. This crucial difference motivates the following question:

Are ML-based cryptanalytic methods inherently limited due to their seemingly restricted access to cryptographic samples?

1.1 Our Contribution

We initiate a foundational exploration of the extent to which machine learning methods may lead to significant cryptanalytic advances. First, we introduce a unifying framework that models both "sample-based" adversaries and "gradient-based" ones, and put forward strong simulation notions capturing a concrete and quantifiable extent to which a sample-based adversary may be simulated by a gradient-based one. Then, within our framework, we provide a somewhat surprising negative answer to the above fundamental question via a sequence of increasingly refined general feasibility results, showing that any sample-based adversary can be simulated by a seemingly weaker gradient-based one with no significant loss in efficiency.

Taken together, our results indicate that the extent to which machine learning methods may lead to significant cryptanalytic advances is not inherently limited by their seemingly restricted access to the provided samples. In what follows, we provide a high-level overview of our main contributions.

Modeling gradient-based methods. Our notions of sample-based adversaries and gradient-based ones model algorithms that are provided with access to a set S of samples $(x, y) \in \mathcal{X} \times \mathcal{Y}$, for some finite sets $\mathcal{X}$ and $\mathcal{Y}$. However, while sample-based adversaries receive the sample set S as an explicit input, gradient-based ones receive only the size $|S|$ of the sample set as an explicit input and are then restricted to accessing the sample set itself by querying a corresponding "gradient oracle" $\mathcal{O}_{\mathsf{G}}$.

Our gradient oracle, which models a gradient-based training process, is provided with the sample set S, and receives queries of the form $(\ell, h, \vec{\theta})$, where:

- $\ell : \mathbb{R} \times \mathcal{Y} \to \mathbb{R}^+$ is a differentiable *loss function*, and
- $h : \mathbb{R}^p \times \mathcal{X} \to \mathbb{R}$ is a differentiable *model function* equipped with its *model parameters* $\vec{\theta} \in \mathbb{R}^p$.

At a high level, the model function h represents the gradient-based adversary's current estimate of a mapping between $\mathcal{X}$ and $\mathcal{Y}$, and the loss function ℓ is used by the oracle to determine the extent to which the estimated mapping is accurate. Specifically, given a sample set S and a query $(\ell, h, \vec{\theta})$, the gradient oracle $\mathcal{O}_{\mathsf{G}}$ averages, over all samples $(x, y) \in S$, the gradient $\nabla_\theta \ell(h(\vec{\theta}, x), y)$ of the loss function when composed with the model function $h(\vec{\theta}, \cdot)$. Intuitively, this provides a measure of the way the model parameters $\vec{\theta}$ may be modified in order to minimize the value of the loss function. Note that the gradient oracle still enables general access in terms of adversarial choice of ℓ and h that can change adaptively with each query. Common ML algorithms, including cryptanalytic ones, interact with gradient oracles in a much more limited fashion, most often via gradient descent (GD). Here, algorithms start with some initial set of parameters $\vec{\theta}_0$ and then, over a fixed number of iterations, refine them by (1) querying $(\ell, h, \vec{\theta}_{t-1})$ to obtain a gradient vector $\vec{g}$, and (2) applying a fixed update rule $\vec{\theta}_t = \vec{\theta}_{t-1} - \eta \cdot \vec{g}$. Here, η is a parameter called the learning rate.

As we concretely demonstrate in Appendix A, this modeling of a gradient-based training process is indeed sufficiently general and expressive for capturing the ML-based cryptanalytic methods that have so far been developed. At the same time, such gradient-based and, even more so, GD-based algorithms clearly utilize very limited access to samples. Intuitively, it would seem that this could limit the power of such adversaries, at least compared to ones that have unfettered sample access.

Notions of simulation. To explore this intuition, we put forward strong simulation notions capturing a concrete and quantifiable extent to which a sample-based adversary may be simulated by a gradient-based one. Our simulation notions are strong in the sense that they enable replacing any sample-based adversary in a cryptographic experiment with a gradient-based adversary that simulates it. At a high level, we quantify the "ϵ-closeness" of such a simulation in an information-theoretic manner, and say that a gradient-based adversary $\mathcal{B}$ simulates a sample-based one $\mathcal{A}$ if, for any distribution $\mathcal{D}$ that produces samples sets, it holds that

$$\mathrm{SD}_{S \leftarrow \mathcal{D}}\left(\left(S, \mathcal{A}\left(S\right)\right), \left(S, \mathcal{B}^{\mathcal{O}_{\mathsf{G}}(S,\cdot,\cdot,\cdot)}\left(1^{|S|}\right)\right)\right) \leq \epsilon,$$

where SD denotes the statistical distance between the above two distributions. This guarantees that whenever we replace a sample-based adversary in a cryptographic experiment with a gradient-based one that simulates it within distance ϵ, the results of the two experiments differ by statistical distance at most ϵ.

Our results in this work, in fact, provide an even stronger guarantee that we formalize by asking for a single "universal" gradient-based adversary that can be used to simulate *any* sample-based algorithm. For this purpose, such a universal black-box gradient-based simulator is provided with oracle access both to the gradient-based oracle and to the sample-based adversary it simulates.

Goal: gradient-based simulation—with low overhead. Equipped with our framework, we then start exploring the feasibility of constructing gradient-based simulators for sample-based adversaries. Concretely, our first goal is to understand whether any sample-based adversary may be simulated by a gradient-based one. At this point, we crucially observe that such a feasibility result on its own may be insufficient for cryptographic applications: Although simulating within statistical distance ϵ guarantees that the *success probability* of the resulting gradient-based adversary would be essentially identical to that of the sample-based adversary that it simulates, it does not capture the *efficiency overhead* of the simulation.

Thus, given that the security of cryptographic primitives and schemes is measured via the tradeoff between the success probability of attacking them and the efficiency of the attack, we must therefore additionally consider the efficiency overhead of the simulator. We capture this overhead by considering the following two key measures of efficiency for any gradient-based simulator: The internal running time of the simulator and the number of queries that it issues to the gradient oracle.

Black-box perfect simulation via DFS-based extraction. As our first construction, we present a gradient-based adversary that black-box simulates any sample-based one. Moreover, the simulation is *perfect* in terms of producing a completely identical output distribution, and is highly efficient in terms of the simulator's internal running time and query complexity (both are essentially linear in the number of samples).

The main observation underlying this construction is that gradient queries are, somewhat counterintuitively, sufficiently expressive for efficiently extracting the entire set of samples. Specifically, we observe that, for any sample set $S = \{(x_i, y_i)\}_{i \in [s]}$ and for any given prefix z, a single gradient query enables to distinguish between the case in which there are no samples x_i that are prefixed with z, the case in which there is exactly one sample x_i that is prefixed with z, and the case in which there is more than one sample x_i that is prefixed with z. We then rely on this observation to realize a recursive depth-first search (DFS) based exploration of the given sample set by interacting with the gradient oracle. Ultimately, our simulator is rather efficient, using $O(|S| \cdot \log |\mathcal{X}|)$ gradient queries and has time overhead $O(|S| \cdot \log^2 |\mathcal{X}|)$ (see Theorem 4.1).

Our first construction provides a surprising negative answer to our main research question. But it is somewhat unsatisfying because, while on the one hand, it constructively shows that gradient-based adversaries are as strong as any sample-based one, it does so using gradient queries that are fundamentally adaptive and, in particular, are not naturally produced by any GD-based adversary, i.e., an adversary that only runs the GD algorithm on the samples. Therefore, it may still be the case that although gradient-based methods are *generally* not inherently limited, the commonly-used methods (most notably, GD and its various variants) are inherently limited.

Gradient descent via parallelizable gradient queries. We therefore go on to extend our approach to build a simulator that is GD-based. Such a simulator would have to somehow succeed at simulating a sample-based adversary despite being severely restricted in its interaction with the gradient oracle: it is not allowed to process the intermediate responses of the gradient oracle in any way other than determining the next query based on the fixed GD update rule. Crucially, the simulation can only choose the model function h and the loss function ℓ, along with the initial parameters $\vec{\theta}_0$, the learning rate η, and the iteration number T. The output of the simulation then can only rely on the final parameters $\vec{\theta}_T$ produced by the GD algorithm. The techniques underlying our DFS-based simulator above are now inapplicable as they rely on adaptively discarding "non-useful" paths in order to be efficient. We need a different approach.

As an intermediate step on our way to construct a GD-based simulator, we first devise a *non-adaptive* gradient-based simulation technique. We present a black-box gradient-based simulator whose gradient queries are issued in a non-adaptive manner and are thus fully parallelizable (i.e., all of its gradient queries can be issued within a single round of parallel queries). This simulator crucially relies on randomization and thus does not provide a perfect simulation as our first simulation. Instead, it provides simulation within any statistical distance ϵ, while its efficiency scales with just $\log(1/\epsilon)$ (therefore, from a cryptographic perspective, this allows choosing ϵ to be a negligible function of the security parameter). In fact, as we discuss in Section 5, this construction does not only present an intermediate simulation technique, but enables us to avoid an undesirable sequential overhead in the simulation of parallelizable cryptanalytic tasks.

Finally, we build off our parallelizable simulator to obtain a black-box GD-based simulator. The efficiency of this simulator crucially relies on our non-adaptive simulation, as each of its gradient oracle queries essentially encodes a number of non-adaptive queries. While encoding multiple queries, we introduce a mechanism to separate between them, such that each query issued by the GD-based simulator only replicates a single query issued by the non-adaptive simulator. When concluding all iterations, the simulator can extract the samples from the final set of parameters.

Summary and open problems. The framework we have introduced enables to formally reason on the applicability of gradient-based methods to the design of new cryptanalytic approaches. Focusing on the restricted manner in which such methods process their provided samples, our results indicate that gradient-based methods are not inherently limited when compared to classical cryptanalytic approaches. Moreover, this holds even when further restricting gradient-based methods to the most common gradient-descent learning methodology.

Our exploration gives rise to a variety of interesting problems, with the aim of both improving our concrete techniques and constructions, and extending the reach of this line of research to consider various restrictions on ML-based methods. In what follows, we briefly exemplify some of these potential future directions:

- Although our techniques already lead to highly efficient constructions in terms of their internal time and query complexities, a natural goal is to establish *lower bounds* on the required overhead for black-box simulating any sample-based adversary. Moreover, given that the running time of our GD-based simulator is somewhat higher than that of our other simulators, it would be interesting to identify any inherent additional overhead imposed by restricting the simulation to use gradient descent.

- Our parallelizable simulator crucially relies on randomization, as discussed above, and does not provide a perfect simulation as our DFS-based deterministic one. A foundational research direction is to explore the role of randomness in gradient-based simulation of sample-based adversaries, and whether it is indeed essential for parallelism in this setting.

- Each of our gradient-based simulators utilizes carefully crafted queries, which are based on non-trivial choices of the loss function, model function, and parameters. This setting is somewhat theoretical and might not fully reflect some common practices in ML. An interesting direction for further research would be to explore whether similar results may be obtained even when using a more restricted class of queries (e.g., fixing the loss function to be a commonly used loss such as the square loss).
- A weakened variant of our gradient oracle may be obtained by adding a certain amount of independent noise to each of its responses. This realistically models, for example, training with limited-precision samples, and training in the presence of random labeling errors. For this and other practically motivated variants of our gradient oracle, it would be fascinating to identify the extent to which they enable simulating sample-based adversaries.

1.2 Related Work

ML-based cryptanalysis. As discussed above, ML-based cryptanalytic methods have so far been developed in the contexts of block ciphers and the LWE problem. In the context of block ciphers, Gohr [Goh19] focused on round-reduced versions of Speck32/64 [BSS+13], and trained a ResNet-based [HZR+16] classification neural network to distinguish between ciphertext pairs that originate from plaintexts with a predefined difference, and ciphertext pairs that originate from independent plaintexts. Gohr's neural distinguisher outperforms classical methods for 5 to 8 rounds of Speck32/64. Additionally, a key-recovery attack for 11 rounds of Speck32/64 was proposed, based on the trained model, with an estimated time complexity of 2^{38}, improving upon the best-known attack of Dinur [Din14] with time complexity of 2^{46}, at the cost of higher data complexity of $2^{14.5}$ ciphertext pairs.

Benamira, Gerault, Peyrin, and Tan [BGP+21] revisited Gohr's work with the goal of obtaining a better understanding of its success, suggesting that his distinguisher was able to effectively approximate the cipher's differential distribution tables (DDT). Baksi [BB22] and Jain, Kohli, and Mishra [JKM20] proposed a somewhat different approach, and trained neural networks to classify input differences from a set of predefined differences, instead of distinguishing between pairs with a predefined difference and independent pairs. A significant amount of follow-up work subsequently applied similar ML-based methods to other block ciphers (see, for example, [HRC21a, SZM21, YK21, BB22, ZW22, BB22, CSY+23, ZW22, CSY+23, BB22, CSY+23, ZW22, JKM20, LLS+22, ZWw22, HRC21b, BGL+21, LLS+22, GLN22, BLY+23]).

In the context of the LWE problem, Wenger, Chen, Charton, and Lauter [WCC+22] trained a neural network on LWE samples with a shared secret s, which they then used for a secret-recovery attack. This enables them to extract secrets for very sparse (only 3 or 4 nonzero bits) and low-dimension (up to 128) instances of LWE. Li et al. [LSW+23b] proposed an improvement based on a preprocessing step, in which the LWE samples are processed by BKZ [CN11], a lattice reduction algorithm, for obtaining LWE samples with smaller coordinate variance. For larger moduli, this enabled them to extract secrets for higher dimensions (up to 350) and larger Hamming weights (up to 60). A follow-up work by Li et al. [LSW+23a] was then able to reduce the modulus size as well as to additionally recover ternary and small Gaussian secrets. Stevens et al. [SWL+24] proposed additional improvements that enabled to improve efficiency and reduce sample complexity. The extent to which these methods can outperform classical methods is still unclear, as also noted by Ducas et al. [DPS23]. In Appendix A we show how the previous ML-based cryptanalytic methods are indeed captured within our framework.

PAC vs. differential-based learning. In the context of machine learning theory, some works focused on studying the power and limitations of learning with GD and stochastic gradient-descent (SGD), a popular variant of GD. Abbe and Sandon [AS20] showed that a neural network trained with SGD with a batch size of 1 (i.e., each gradient update is based on the gradient of a single example) can implement any PAC learning algorithm [Val84]. Namely, any algorithm for learning with examples can be used to generate a neural network, such that running batch-1 SGD over this network simulates the learning algorithm. This result was extended by Abbe et al. [AKM+21], showing that SGD with a larger batch size can also simulate algorithms for learning from examples, as long as the gradients are not noisy, where the noise is modeled by the level of arithmetic precision. These results show that with low noise, i.e., high arithmetic precision, SGD is equivalent to PAC learning from examples, while SGD with high noise,i.e., low precision, is equivalent to learning from Statistical Queries [Kea98], a learning framework known to be weaker than PAC.

These works focused on comparing different learning frameworks (e.g., learning with SGD versus PAC and Statistical Query learning), but did not study the power of gradient-based algorithms in the context of cryptography. From the technical perspective, the work of Abbe et al. focuses on (a variant of) the SQ learning model as an intermediate learning model in between PAC learning and differential-based learning. In the cryptographic context, however, the SQ learning model does not seem to directly capture realistic applications (either classic ones or recent ML-based ones as those discussed above). Therefore, we focus on directly exploring the interplay between samples-based methods and gradient-based ones.

Specifically, in the SQ setting a learner interacts with the SQ oracle by issuing statistical queries about the sample set, while in the gradient-based setting, the learner interacts with the gradient oracle by issuing queries that consist of an estimated mapping (as a parameterized function) and a loss function (used to estimate the quality of the estimate). This difference in interaction with the oracle requires to structure the oracle queries differently in order to extract the same information, namely the sample set. For this, Abbe et al. constructed "counting" queries, that ask for the number of samples that match some rule, e.g., prefix. In our gradient-based setting, we construct the queries by designing a dedicated loss function and parameterized model function such that the gradient with respect to the samples will directly encode samples that match some rule, e.g., prefix. Most crucially, our direct approach enables us to focus on quantitative measures that are more fundamental in the cryptographic setting, such as the running time and query complexity of attackers, while Abbe et al. focus most of their technical attention on identifying the level of numeric precision that separates differential-based learning from either PAC learning or SQ learning.

Memorization in machine learning. Neural networks are known to be able to "memorize" some of their training samples. Empirically it has been shown that large enough deep neural networks can achieve good accuracy on large, randomized training sets [ZBH+17]. A line of previous works explored this phenomenon using neural tangent kernels [JGH18] to show that large enough networks, trained with GD, can, in theory, memorize the label of each sample in the training set [ADH+19, DLL+19, Dan20]. Some works showed how individual training samples can be reconstructed from trained models, both in the vision and in the text domains [CLE+19, HVY+22, BHY+23]. Song et al. [SRS17] demonstrated that by maliciously modifying the training algorithm, an adversary can cause intended memorization and increase its ability to extract samples or information about the training set. Seen in this light, our simulators can be viewed as malicious training algorithms, and our results go beyond prior ones by showing how to extract the *entire* training set (rather than just a few points), and moreover to do so efficiently, as we show in this work. That said, our simulators do not attempt to simultaneously memorize training data while also achieving good performance for some baseline tasks.

1.3 Paper Organization

The remainder of this paper is organized as follows. First, in Section 2 we present several standard notions that are used throughout this work. In Section 3 we present our framework for modeling sample-based adversaries and gradient-based ones. In Section 4 we establish our general feasibility result by presenting a gradient-based adversary that black-box simulates any sample-based one. In Sections 5 and 6 we then show that our approach extends to providing a gradient-based simulator whose gradient queries are fully parallelizable, and a gradient-based simulator that follows the particular gradient-descent algorithm, respectively. In Section 7 we provide extensions of our results, from the single-bit output case to the multi-bit one, and from the full-batch gradient oracle to the mini-batch case, as used in the popular SGD algorithm. In Appendix A we demonstrate the generality of our notion of gradient-based adversaries, and exemplify how previous ML-based cryptanalytic methods are indeed captured by it.

2 Preliminaries

For an integer $n \in \mathbb{N}$ we denote by $[n]$ the set $\{1, \ldots, n\}$. For a distribution X we denote by $x \leftarrow X$ the process of sampling a value x from the distribution X. Similarly, for a set $\mathcal{X}$ we denote by $x \leftarrow \mathcal{X}$ the process of sampling a value x from the uniform distribution over $\mathcal{X}$. For a binary string $x \in \{0,1\}^n$, we interchangeably refer to x both as a binary vector $x = x_1 \cdots x_n \in \{0,1\}^n$ and as a real-valued one $\vec{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$. For two random variables X and Y over a finite domain Ω we denote by $\mathrm{SD}(X,Y)$ their statistical distance, which is defined as

$$\mathrm{SD}(X,Y) = \frac{1}{2} \sum_{\omega \in \Omega} \left| \Pr\left[\, X = \omega \,\right] - \Pr\left[\, Y = \omega \,\right] \right| \ .$$

Let $\mathcal{M}_{n,k}$ be a family of functions mapping from n-bit inputs to k-bit outputs. The family $\mathcal{M}_{n,k}$ is said to be *pairwise-independent* if for every $x_1, x_2 \in \{0,1\}^n$ such that $x_1 \neq x_2$, and for every $y_1, y_2 \in \{0,1\}^k$, it holds that

$$\Pr_{M \leftarrow \mathcal{M}_{n,k}}\left[\, M(x_1) = y_1 \wedge M(x_2) = y_2 \,\right] = \frac{1}{2^{2k}} \ .$$

For a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$, the gradient $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ with respect to $\vec{x} = (x_1, \ldots, x_n)$ is defined as the vector of its partial derivatives with respect to each component of $\vec{x}$, i.e.,

$$\nabla f(\vec{x}) = \left(\frac{\partial f(\vec{x})}{\partial x_1}, \ldots, \frac{\partial f(\vec{x})}{\partial x_n} \right) \in \mathbb{R}^n \ .$$

For functions defined over multiple inputs, we use subscript to denote the input with respect to which the gradient is computed. For example, for a differentiable function $f : \mathbb{R}^{n_1} \times \mathbb{R}^{n_2} \to \mathbb{R}$ with two input variables $x \in \mathbb{R}^{n_1}$ and $z \in \mathbb{R}^{n_2}$, its gradient with respect to x is defined as

$$\nabla_x f(\vec{x}, \vec{z}) = \left(\frac{\partial f(\vec{x}, \vec{z})}{\partial x_1}, \ldots, \frac{\partial f(\vec{x}, \vec{z})}{\partial x_n} \right) \in \mathbb{R}^{n_1} \ .$$

3 Our Framework

In this section, we present our framework for modeling both "sample-based" adversaries that are provided with direct access to individual samples and "gradient-based" ones that are restricted to issuing gradient-based queries that are averaged over all given samples via a loss function. We begin by formally defining our notion of sample-based adversaries and exemplifying the range of fundamental cryptographic applications that they enable us to capture in this context. Then, we formally define our notion of gradient-based adversaries, and provide strong simulation notions capturing a concrete and quantifiable extent to which a sample-based adversary may be simulated by a gradient-based one.

Sample-based adversaries. Our notion of sample-based adversaries captures the basic form of an algorithm that explicitly receives as input a sample set, denoted S, and produces an output. Throughout this work, when we refer to a "sample set", we in fact refer to a *list* that contains pairs of the form $(x, y) \in \mathcal{X} \times \mathcal{Y}$, for some finite sets $\mathcal{X}$ and $\mathcal{Y}$ (the term *set* was chosen for consistency with the common ML terminology of learning from datasets).

Definition 3.1 (Sample-based adversary)**.** Let $\mathcal{X}$ and $\mathcal{Y}$ be finite sets, and let $\mathcal{D}$ be a distribution over $(\mathcal{X} \times \mathcal{Y})^*$. A *sample-based* adversary $\mathcal{A}$ is a potentially-randomized algorithm that, when given as input a sample set $S \leftarrow \mathcal{D}$, produces an output $\mathcal{A}(S) \in \{0,1\}^*$.

Despite its simplicity, our notion of a sample-based adversary captures a wide range of fundamental cryptographic applications, including both unpredictability and indistinguishability ones. Specifically, providing the above distribution $\mathcal{D}$ (which produces the sample sets in Definition 3.1) with access to a typically-keyed cryptographic primitive enables to capture various standard security notions corresponding to the primitive, as we now exemplify.

The most interesting applications in our setting, however, are those in which the produced sample sets correspond to ones that may be realistically used by ML-based cryptanalytic methods. Such sample sets are typically generated by *independently* sampling each pair (x, y) from a given distribution that originates from a *hard-to-compute mapping* between x and y (and, more generally, a hard-to-compute *relation*). For unpredictability applications, this captures, in particular, key recovery for various forms of keyed primitives (e.g., block ciphers, pseudorandom functions and encryption schemes) as well as unforgeability for MACs and signatures. In both cases, sample sets may consist, for example, of pairs $(x, F_{\mathsf{sk}}(x))$, where F is the keyed primitive, and all of the x values are independently sampled from a given distribution (e.g., the uniform distribution, which would correspond to a random message attack). The goal of the adversary is either to extract the key sk (for key recovery) or to produce a new "valid" pair (x^*, y^*) for a value x^* that was not included in sample set (for unforgeability of MACs and signatures).[1]

As somewhat expected, the above applications do not cover adaptive attacks (e.g., ones in which an attacker would choose the x values one by one in an adaptive manner after observing each corresponding output $F_{\mathsf{sk}}(x)$). Indeed, realistic sample sets for ML-based methods are of a rather static flavor, as discussed above, and we concretely exemplify this in Appendix A using those utilized by Gohr [Goh19] and by Wenger, Chen, Charton, and Lauter [WCC$^+$22]. Nevertheless, it is important

[1] Similarly, for indistinguishability applications, realistic ML sample sets naturally arise in various pseudorandomness experiments (such as those used to define the security of weak pseudorandom functions, or hard-core predicates for a one-way function). In these applications, the goal of the adversary is to distinguish between two distributions from which the sample set S is sampled: In one distribution it consists of pairs (x, y) where y is a pseudorandom value computed from x (e.g., the output of a weak pseudorandom function or a hard-core predicate applied to the inverse of x), and in the other distribution it consists of such pairs where y is uniformly distributed.

to note that our framework and results capture arbitrary sample sets that may be generated by any distribution $\mathcal{D}$, and not only those that may be viewed as realistic for ML-based cryptanalysis.

Gradient-based adversaries. Our notion of a gradient-based adversary relies on a corresponding gradient oracle. This oracle, as formally defined in Figure 1, receives as input a sample set S, two functions denoted ℓ and h, as well as a vector of parameters $\vec{\theta}$ for the function h. The first function, ℓ, typically referred to as the *loss function*, is a differentiable function $\ell : \mathbb{R} \times \mathcal{Y} \to \mathbb{R}^+$. The second function, h, typically referred to as the *model*, is a parameterized differentiable function $h : \mathbb{R}^p \times \mathcal{X} \to \mathbb{R}$, where $\vec{\theta} \in \mathbb{R}^p$ is its vector of parameters ($\vec{\theta}$ is typically referred to as the *model parameters*). Generally speaking, the function h represents the adversary's current estimate of a mapping between $\mathcal{X}$ and $\mathcal{Y}$, and the loss function ℓ is used by the oracle to determine the extent to which the estimated mapping is accurate.[2]

The gradient oracle $\boldsymbol{\mathcal{O}_{\mathsf{G}}\left(S, \ell, h, \vec{\theta}\right)}$:

1. Compute and output $\vec{g} \leftarrow \frac{1}{|S|}\sum_{(\vec{x},y)\in S} \nabla_\theta \ell(h(\vec{\theta}, \vec{x}), y)$.

Figure 1: The gradient oracle $\mathcal{O}_{\mathsf{G}}$.

For a given query $\left(S, \ell, h, \vec{\theta}\right)$, the oracle evaluates $\hat{y}_i = h(\vec{\theta}, \vec{x}_i)$ for each sample $(\vec{x}_i, y_i) \in S$, and compares it to y_i using ℓ, i.e., $\ell(\hat{y}_i, y_i)$. Then, the oracle computes the gradient of the evaluation of the loss function with respect to the parameters $\vec{\theta}$. Denote the individual parameters as $\vec{\theta} = (\theta_1, \ldots, \theta_p)$, then the gradient is the vector of partial derivatives with respect to each parameter:

$$\nabla_\theta \ell(h(\vec{\theta}, \vec{x}_i), y_i) = \left(\frac{\partial \ell(h(\vec{\theta}, \vec{x}_i), y_i)}{\partial \theta_1}, \ldots, \frac{\partial \ell(h(\vec{\theta}, \vec{x}_i), y_i)}{\partial \theta_p} \right) .$$

Then, the oracle outputs the average of this gradient over all samples in the set S. Computing the gradient of the loss function, composed with $h(\vec{\theta}, \cdot)$, with respect to the parameters $\vec{\theta}$ provides a measure for the way these parameters may be modified in order to minimize the value of the loss function. In other words, the oracle evaluates the performance of $h(\vec{\theta}, \cdot)$ over the sample set S using the loss function ℓ and then computes the gradient in order to suggest a way to improve this performance. Throughout this work, we do not explicitly require the oracle to verify that all queries consist of differentiable functions. This is since, in all of our results, all issued queries consist of differentiable functions (as explicitly established by our proofs).

Note that although it is not essential for our framework to explicitly force any computational constraints over the functions ℓ and h, we do however assume their computation and derivation can be computed in polynomial time when considering a sufficient level of precision (we would like to avoid, for example, a function h that performs an exhaustive search over an exponentially-large key space). Instead, we include the gradient computation and derivation time in the total running time of a gradient-based adversary $\mathcal{B}$, which includes the time required for expressing and forwarding a certain explicit representation (e.g., an arithmetic circuit) of ℓ, h, and the parameters $\vec{\theta}$ to the oracle $\mathcal{O}_{\mathsf{G}}$.

[2] While it may seem redundant to distinguish between h and $\vec{\theta}$ and to provide the oracle both as input (since $\vec{\theta}$ can contain a function description and h can be a universal function computing the function provided by $\vec{\theta}$), we nevertheless distinguish between the two since this would be useful for defining our next class of adversaries, in which both h and $\vec{\theta}$ admit a particular structure.

Definition 3.2 (Gradient-based adversary)**.** Let $\mathcal{X}$ and $\mathcal{Y}$ be finite sets, and let $\mathcal{D}$ be a distribution over $(\mathcal{X} \times \mathcal{Y})^*$. A *gradient-based* adversary $\mathcal{B}$ is a potentially-randomized algorithm that, when provided with access to the oracle $\mathcal{O}_{\mathsf{G}}(S, \cdot, \cdot, \cdot)$ for a set $S \leftarrow \mathcal{D}$, and given as input $1^{|S|}$, produces an output $\mathcal{B}^{\mathcal{O}_{\mathsf{G}}(S,\cdot,\cdot,\cdot)}(1^{|S|}) \in \{0,1\}^*$.

Definition 3.2 allows the adversary $\mathcal{B}$ to perform arbitrary computations and interact with the oracle $\mathcal{O}_{\mathsf{G}}$ in any way, under the only restriction that the queries must consist of differentiable functions. However, in the most common learning methodology, the learner (adversary in our setting) runs a particular iterative optimization algorithm in order to learn an estimate of some function or mapping. The Gradient Descent (GD) algorithm is the base for the most popularly used optimization algorithms. In this algorithm, given a fixed loss function ℓ and function h, the learner chooses the first oracle query using some heuristic, meaning it chooses initial parameters $\vec{\theta}_0$. Then, the following queries are dependent on a computation of a specific form over the oracle's responses. This computation, known as the *GD update rule*, receives the current query and the oracle's response to it, and defines the next one. More specifically, at each time step t, given the current query $(\ell, h, \vec{\theta}_t)$ and the oracle's response $\vec{g}_t$, the parameters of next query $(\ell, h, \vec{\theta}_{t+1})$ are defined as:

$$\vec{\theta}_{t+1} = \vec{\theta}_t - \eta \cdot \vec{g}_t \ ,$$

where $\eta \in \mathbb{R}$ is known as the step size or learning rate. In other words, the GD algorithm uses the gradients to direct the parameters of h towards a set of parameters that minimizes the loss functions.

Therefore, we additionally consider the following notion of a *GD-based adversary* as a refinement of a gradient-based one: A GD-based adversary $\mathcal{B}_{\mathsf{GD}[T,\ell,h,\eta,\vec{\theta}_0]}$ is a gradient-based adversary that runs a GD algorithm for T iterations, and can then perform any additional computation over the output of the T-th iteration. We denote by $\mathsf{PostProcess}$ such additional computation, which can be viewed either as part of the internal description of the adversary $\mathcal{B}_{\mathsf{GD}[T,\ell,h,\eta,\vec{\theta}_0]}$ or as a supplied parameter. See Figure 2 for the formal description.

The GD-based adversary $\mathcal{B}^{\mathcal{O}_{\mathsf{G}}(S,\cdot,\cdot,\cdot)}_{\mathsf{GD}[T,\ell,h,\eta,\vec{\theta}_0]}$:

```
1. For $t \in \{1, \ldots, T\}$ :
    1.1 Query $\mathcal{O}_{\mathsf{G}}(S, \cdot, \cdot, \cdot)$ with $(\ell, h, \vec{\theta}_{t-1})$ to obtain $\vec{g}_t$.
    1.2 Update $\vec{\theta}_t = \vec{\theta}_{t-1} - \eta \cdot \vec{g}_t$.
2. Output $v = \mathsf{PostProcess}(\vec{\theta}_T)$.
```

Figure 2: The GD-based adversary $\mathcal{B}_{\mathsf{GD}}$.

In Appendix A, we demonstrate that our notion of a GD-based adversary enables us to capture the ML-based cryptanalytic methods that have been studied so far. We exemplify this by focusing on the methods developed by Gohr [Goh19] and by Wenger, Chen, Charton, and Lauter [WCC+22] (as discussed in Section 1.2 and which have already led to additional similar methods), and showing that they indeed fit into our framework.

We emphasize that GD is not the only optimization algorithm used in learning settings, and that there are many others, such as SGD [RM51], ADAM [KB14], and many more [Rud16]. However, in this work, we focus our attention on the GD algorithm, as it is the base algorithm behind many of the popularly used algorithms. This is not a limitation of our work, and all of our results can be adapted to other algorithms as well. In particular, in Section 7.2 we extend our approach to the commonly used *stochastic mini-batch* setting (which also represents the key difference between the GD and SGD algorithms).

Our notions of simulation. We capture the extent to which a gradient-based adversary simulates a sample-based one by measuring the statistical distance between their input-output distributions. This enables to replace any sample-based adversary in a cryptographic experiment with a gradient-based adversary that simulates it, and ensures that the statistical distance between the experiments is bounded by the "closeness" of the simulation.

Definition 3.3 (ϵ-simulation)**.** Let $\mathcal{X}$ and $\mathcal{Y}$ be finite sets, let $\epsilon \geq 0$, and let $\mathcal{A}$ and $\mathcal{B}$ be a sample-based adversary and a gradient-based adversary, respectively. Then, we say that $\mathcal{B}$ *ϵ-simulates* $\mathcal{A}$ with respect to $\mathcal{X} \times \mathcal{Y}$ if for any distribution $\mathcal{D}$ over $(\mathcal{X} \times \mathcal{Y})^*$, it holds that

$$\mathrm{SD}\left(\left(S, \mathcal{A}\left(S\right)\right), \left(S, \mathcal{B}^{\mathcal{O}_{\mathsf{G}}(S,\cdot,\cdot,\cdot)}\left(1^{|S|}\right)\right)\right) \leq \epsilon,$$

where $S \leftarrow \mathcal{D}$ in both distributions.

Definition 3.3 presents a highly intuitive notion of simulation, providing a strong information-theoretic guarantee that enables to replace any sample-based adversary with a gradient-based one that simulates it. Our results in this work, in fact, provide an even stronger guarantee obtained by naturally extending Definition 3.3 to ask for a single "universal" gradient-based algorithm that ϵ-simulates for *any* sample-based algorithm. For this purpose, such a universal gradient-based algorithm $\mathcal{B}$ is provided with oracle access both to the gradient-based oracle $\mathcal{O}_{\mathsf{G}}$ and to the sample-based adversary $\mathcal{A}$ that it simulates.

Definition 3.4 (Black-box ϵ-simulation)**.** Let $\mathcal{X}$ and $\mathcal{Y}$ be finite sets and let $\epsilon \geq 0$. We say that a gradient-based algorithm $\mathcal{B}$ *black-box ϵ-simulates all sample-based adversaries* with respect to $\mathcal{X} \times \mathcal{Y}$ if for any distribution $\mathcal{D}$ over $(\mathcal{X} \times \mathcal{Y})^*$ and for any sample-based adversary $\mathcal{A}$ it holds that

$$\mathrm{SD}\left(\left(S, \mathcal{A}\left(S\right)\right), \left(S, \mathcal{B}^{\mathcal{O}_{\mathsf{G}}(S,\cdot,\cdot,\cdot),\mathcal{A}(\cdot)}\left(1^{|S|}\right)\right)\right) \leq \epsilon,$$

where $S \leftarrow \mathcal{D}$ in both distributions.

Whenever Definitions 3.3 and 3.4 are satisfied by a gradient-based adversary $\mathcal{B}$ for $\epsilon = 0$ (as in the case with our simulator in Section 4), we refer to such an ϵ-simulation as *prefect* simulation.

Measuring the simulation overhead. Finally, for identifying the overhead incurred when simulating a sample-based adversary by a gradient-based one, throughout this work we focus on the following main measures of efficiency for such algorithms $\mathcal{A}$ and $\mathcal{B}$:

- We denote by $T_{\mathcal{A}} = T_{\mathcal{A}}(s, |\mathcal{X}|, |\mathcal{Y}|)$ and $T_{\mathcal{B}} = T_{\mathcal{B}}(s, |\mathcal{X}|, |\mathcal{Y}|, T_{\mathcal{A}})$ the running time of $\mathcal{A}$ and $\mathcal{B}$, respectively, while naturally allowing the running time of $\mathcal{B}$ to depend on that of $\mathcal{A}$. However, when considering a *black-box* simulator $\mathcal{B}$, we measure its running time as a function of only s, $|\mathcal{X}|$ and $|\mathcal{Y}|$, while separately accounting for the number of queries that it issues to $\mathcal{A}$. In order to account for the internal runtime of $\mathcal{B}$, we assume as a standard baseline that simple arithmetic operations (e.g., additions and multiplications) over a small constant number of elements from $\mathcal{X}$ and $\mathcal{Y}$ can be executed in unit cost.
- We denote by $Q_{\mathcal{B}} = Q_{\mathcal{B}}(s, |\mathcal{X}|, |\mathcal{Y}|, T_{\mathcal{A}})$ the number of gradient-oracle queries issued by $\mathcal{B}$ (where, again, we allow the number of queries issued by $\mathcal{B}$ to depend on the running time of $\mathcal{A}$ – although this will not be used by our constructions).

Simulation via neural networks. Neural networks (NN) are a particularly important class of (typically) non-linear differentiable functions and are the most popular family of model functions h used in the ML literature. As discussed above, in our framework the model functions h are assumed to be computable in polynomial time, i.e., their computation can be implemented using Turing machines (TM) running in polynomial time. As such, they can be represented by a bounded-size NN. In a nutshell, this holds as poly-sized TMs can be implemented using poly-sized Boolean circuits [Coo23, Lev73], which in turn can be represented by bounded-sized NNs [SSBD14]. As a result, although being the most common function family in the ML community, in particular in the ML-based cryptanalysis literature, we do not need to explicitly limit the class of functions h to neural networks as any h can be represented as one.

Distinct vs. arbitrary samples. Our gradient-based simulators, which we present throughout the following sections, assume that the sample sets S given as input to the gradient oracle $\mathcal{O}_{\mathsf{G}}(S,\cdot,\cdot,\cdot)$ consist of "distinct" samples. Specifically, letting $S = \{(x_i, y_i)\}_{i\in[s]} \subset \mathcal{X} \times \mathcal{Y}$, our gradient-based simulators assume that $x_i \neq x_j$ for any $i \neq j \in [s]$ (i.e., that the x-values within any sample set are always distinct). This, however, may not be a valid assumption whenever the underlying distribution $\mathcal{D}$ may generate non-distinct samples (say, with some non-negligible probability).

Nevertheless, this assumption does not limit the scope of our framework. This is due to the fact that machine-learning methods naturally allow to preprocess sample sets, where in our case a naive single-pass serialization suffices. That is, any sample set $S = \{(x_i, y_i)\}_{i\in[s]}$ can be easily transformed into a serialized one $S' = \{(i||x_i, y_i)\}_{i\in[s]}$ by concatenating an index to each sample. This increases the bit-length of the x-values from $\log|\mathcal{X}|$ to $\log|\mathcal{X}| + \log s$, which for our results yields only a minor lower-level overhead with no asymptotic effect.[3] Thus, in the remainder of this work, we assume that all sample sets which are given as input to the gradient oracle always consist of distinct samples (and we will not explicitly serialize and deserialize them).

4 Perfect Simulation via DFS-Based Extraction

Equipped with our framework for modeling sample-based and gradient-based adversaries, in this section we establish a general feasibility result by presenting a gradient-based adversary that black-box simulates any sample-based one. Moreover, the simulation is *perfect*, ensuring a completely identical output distribution (see Definition 3.4), and exhibits a *nearly optimal overhead* in terms of the gradient-based adversary's running time (when compared to that of the simulated sample-based adversary).

For simplicity, here we state and prove our result for samples over $\mathcal{X} \times \mathcal{Y}$ where $\mathcal{X} = \{0,1\}^n$ for some integer $n \in \mathbb{N}$ and $\mathcal{Y} = \{0,1\}$. In Section 7 we extend our result to capture the more general setting in which $\mathcal{Y} = \{0,1\}^m$ for some integer $m \in \mathbb{N}$. We prove the following theorem:

Theorem 4.1. *Let $\mathcal{X} = \{0,1\}^n$ for some $n \geq 1$, and let $\mathcal{Y} = \{0,1\}$. There exists a gradient-based adversary $\mathcal{B}$ that black-box perfectly-simulates all sample-based adversaries with respect to $\mathcal{X} \times \mathcal{Y}$, where:*

- *$\mathcal{B}$ runs in time $T_{\mathcal{B}}(|S|, |\mathcal{X}|, |\mathcal{Y}|) = O(|S| \cdot \log^2 |\mathcal{X}|)$.*

[3] Looking ahead, our gradient-based simulators would in fact extract the serialized sample set S' via their oracle queries, and then invoke the sample-based adversary $\mathcal{A}$ providing it with the original sample set S as input. This naturally requires de-serializing each sample, which again yields only a minor lower-level overhead with no asymptotic effect.

- *$\mathcal{B}$ issues $Q_{\mathcal{B}}(|S|, |\mathcal{X}|, |\mathcal{Y}|) = O(|S| \cdot \log|\mathcal{X}|)$ queries to the gradient oracle, and a single query to the simulated sample-based adversary.*

For the more general setting in which $\log|\mathcal{Y}| = m > 1$, for some integer $m \in \mathbb{N}$, our simulator $\mathcal{B}$ runs in time $T_{\mathcal{B}}(|S|, |\mathcal{X}|, |\mathcal{Y}|) = O(|S| \cdot \log|\mathcal{X}| \cdot (\log|\mathcal{X}| + \log|\mathcal{Y}|))$ and issues the same number of queries $Q_{\mathcal{B}}(|S|, |\mathcal{X}|, |\mathcal{Y}|) = O(|S| \cdot \log|\mathcal{X}|)$. We refer the reader to Section 7 for the extended result. In what follows, we first discuss in more detail the overhead of our gradient-based simulator provided by Theorem 4.1. Next, we provide a high-level technical overview of our proof of Theorem 4.1, and then provide its formal proof.

The overhead of our gradient-based simulation. In terms of running time, as our gradient-based simulator issues only a single query to the simulated sample-based adversary, then the running time overhead is simply the simulator's internal running time $\Omega(|S| \cdot \log^2|\mathcal{X}|)$. Note that any sample-based adversary that merely examines the entire sample set S would run in time $\Omega(|S| \cdot \log|\mathcal{X}|)$, and therefore compared to all such sample-based adversaries our overhead is asymptotically optimal within a multiplicative factor of at most $O(\log|\mathcal{X}|)$.

Proof overview. Inspired by the work of Abbe et al. [AKM+21] (as discussed in Section 1.2), our main observation underlying the proof of Theorem 4.1 is that gradient queries are sufficiently expressive for efficiently extracting the entire set of samples. That is, we show that there exists a gradient based-adversary $\mathcal{B}$ that, for any set $S \subseteq \mathcal{X} \times \mathcal{Y}$, can issue $O(|S| \cdot \log|\mathcal{X}|)$ queries to the gradient oracle $\mathcal{O}_{\mathsf{G}}(S, \cdot, \cdot, \cdot)$ and extract the set S. Then, a single query to the simulated sample-based adversary $\mathcal{A}$ provides the output $\mathcal{A}(S)$.

More specifically, our gradient-based simulator (which is formally described in Figure 3 below), is based on the following main observation: For any sample set $S = \{(x_i, y_i)\}_{i \in [s]}$ and for any given prefix z, we can issue a query that enables to distinguish between the case in which there are no samples x_i that are prefixed with z, the case in which there is exactly one sample x_i that is prefixed with z, and the case in which there is more than one sample x_i that is prefixed with z.[4] Thus, beginning with empty string $z = \varepsilon$, this enables our simulator to realize a recursive DFS-based exploration of the sample set S. Specifically, when exploring a path corresponding to a certain prefix z, if no samples are prefixed with z then this path is terminated, if there is exactly one sample prefixed with z then the gradient oracle's response in fact enables to extract the sample, and if there is more than one such sample then the simulator continues recursively with the exploration paths corresponding to the prefixes $z0$ and $z1$. The gradient queries issued by $\mathcal{B}$ enable, in particular, to always correctly distinguish between these three cases, and this ensures that all samples are eventually extracted (and that no false samples are "extracted").

From a more technical perspective, the simulator $\mathcal{B}$ uses the same loss function ℓ across all queries, which is defined as $\ell(\hat{y}, y) = (2 + y) \cdot \hat{y}$. As for our choice of the constant 2, since each $y \in \{0, 1\}$ is a single bit, we have that $y < 2$. Looking ahead, our proof relies on this property to distinguish between the case where a single sample is prefixed by some z to the case where multiple samples are. In Section 7.1 we extend our results to the multi-bit label setting, and show how to adjust the loss function accordingly. Letting $\ell'(h(\vec{\theta}, \vec{x}), y)$ denote the partial derivative of ℓ with respect to $h(\vec{\theta}, \vec{x})$, it holds that

$$\ell'(h(\vec{\theta}, \vec{x}), y) = 2 + y,$$

[4] Recall, as discussed in Section 3, that throughout this work we assume that sample sets always consist of distinct samples, as otherwise straightforward single-pass serialization may be applied.

and therefore the response $\vec{g}$ to any query $(\ell, h, \vec{\theta})$ issued by $\mathcal{B}$ to the gradient oracle $\mathcal{O}_{\mathsf{G}}(S, \cdot, \cdot, \cdot)$ is of the following form (recall Figure 1 for the definition of the gradient oracle):

$$\begin{aligned}\vec{g} &= \frac{1}{|S|} \sum_{(\vec{x},y)\in S} \ell'(h(\vec{\theta}, \vec{x}), y) \cdot \nabla_\theta h(\vec{\theta}, \vec{x}) \\ &= \frac{1}{|S|} \sum_{(\vec{x},y)\in S} (2+y) \cdot \nabla_\theta h(\vec{\theta}, \vec{x}).\end{aligned}$$

At each step of the exploration $\mathcal{B}$ can now tailor the function h, whose gradient $\nabla_\theta h(\vec{\theta}, \vec{x})$ depends on the current prefix z, so that it enables to count the number of samples prefixed with z (and, in case of a single match, to extract it). Specifically, $\mathcal{B}$ chooses h and $\vec{\theta}$ such that $\nabla_\theta h(\vec{\theta}, \vec{x})$ returns $(\vec{x}, 1)$ if $\vec{x}$ is prefixed by z (the additional entry 1 enables counting), and otherwise returns the all-zeros vector. That is, for a current prefix $z = z_1 \dots z_k \in \{0,1\}^k$, and for some vector of parameters $\vec{w} \in \mathbb{R}^{n+1}$, we set $\vec{\theta} = (\vec{w}, z) = ((\vec{w})_1, \dots, (\vec{w})_{n+1}, z_1, \dots, z_k)$ and define z to be a non-differentiable parameter. Note that the exact value of $\vec{w}$ is not important as it is only used for its gradient behaviour, however for consistency we will define it as the all-zeros vector $(0, \dots, 0) \in \mathbb{R}^{n+1}$. We can obtain the desired gradient behavior by defining the function h as follows:

$$\begin{aligned}h(\vec{\theta}, \vec{x}) &= \vec{w} \cdot ((\vec{x})_1, \dots, (\vec{x})_n, 1) \cdot \mathbb{1}_{x_{1,\dots,k}=z} \\ &= (w_1(\vec{x})_1, \dots, w_n(\vec{x})_n, w_{n+1}) \cdot \mathbb{1}_{x_{1,\dots,k}=z} \ ,\end{aligned}$$

and we get:

$$\begin{aligned}\nabla_\theta h(\vec{\theta}, \vec{x}) &= \left(\frac{\partial h(\vec{\theta}, \vec{x})}{\partial w_1}, \dots, \frac{\partial h(\vec{\theta}, \vec{x})}{\partial w_n}, \frac{\partial h(\vec{\theta}, \vec{x})}{\partial w_{n+1}} \right) \\ &= ((\vec{x})_1, \dots, (\vec{x})_n, 1) \cdot \mathbb{1}_{x_{1,\dots,k}=z} \\ &= \begin{cases} (0, \dots, 0) \in \mathbb{R}^{n+1} & \text{if } x_{1\dots k} \neq z \\ ((\vec{x})_1, \dots, (\vec{x})_n, 1) \in \mathbb{R}^{n+1} & \text{otherwise} \end{cases}\end{aligned}$$

Note that although $\vec{\theta}$ contains both the differentiable parameters $\vec{w}$ and the non-differentiable parameters z, for clarity we denote by ∇_θ the gradient with respect to all differentiable parameters (i.e., ∇_θ represents ∇_w). Using the function h and parameters $\vec{\theta}$, as described above, enables us to realize a recursive DFS-based exploration for extracting any sample set S.

Proof of Theorem 4.1. Let $\mathcal{X} = \{0,1\}^n$ for some $n \geq 1$ and $\mathcal{Y} = \{0,1\}$. We show that the gradient-based simulator $\mathcal{B}$ described in Figure 3 black-box perfectly simulates all sample-based adversaries with respect to $\mathcal{X} \times \mathcal{Y}$. In what follows we first prove the correctness of the simulation, and then analyze the running time and query complexity of $\mathcal{B}$.

Let $\mathcal{D}$ be a distribution over $(\mathcal{X} \times \mathcal{Y})^*$. For proving that

$$\mathrm{SD}\left((S, \mathcal{A}(S)), \left(S, \mathcal{B}^{\mathcal{O}_{\mathsf{G}}(S,\cdot,\cdot,\cdot), \mathcal{A}(\cdot)}\left(1^{|S|}\right) \right) \right) = 0 \ ,$$

where $S \leftarrow \mathcal{D}$ in both distributions, we in fact prove a stronger statement. We prove that for any $s \geq 1$ and for any set $S \subseteq (\mathcal{X} \times \mathcal{Y})^s$, the distribution of the output produced by the computation $\mathcal{B}^{\mathcal{O}_{\mathsf{G}}(S,\cdot,\cdot,\cdot), \mathcal{A}(\cdot)}(1^s)$ is identical to the distribution of the output produced by the computation $\mathcal{A}(S)$. For this purpose, it suffices to show that, for any such set S, the algorithm $\mathcal{B}$ always extracts the

```
The black-box gradient-based simulator $\mathcal{B}^{\mathcal{O}_{\mathsf{G}}(S,\cdot,\cdot,\cdot),\mathcal{A}(\cdot)}(1^s)$:
  1. Set $\ell(\hat{y}, y) = (2 + y) \cdot \hat{y}$.
  2. Set $z = \varepsilon$. // empty bit-string
  3. Compute $D =$ CHECK-MATCH$(z)$.
  4. Obtain $v \leftarrow \mathcal{A}(D)$ and output $v$.
The recursive procedure CHECK-MATCH($z$):
  1. Define $k = |z|$ // bit-length of $z$
  2. Define $\vec{w} = (0, \ldots, 0) \in \mathbb{R}^{n+1}$ and set $\vec{\theta} = (\vec{w}, z)$.
  3. Define $h(\vec{\theta}, \vec{x}) = \vec{w} \cdot ((\vec{x})_1, \ldots, (\vec{x})_n, 1) \cdot \mathbb{1}_{x_{1,\ldots,k}=z}$.
  4. Obtain $\vec{g} = \mathcal{O}_{\mathsf{G}}(S, \ell, h, \vec{\theta}) \in \mathbb{R}^{n+1}$
  5. If $(\vec{g})_{n+1} = 0$: // no matches
         Return $\emptyset$.
  6. If $0 < |S| \cdot (\vec{g})_{n+1} < 4$: // single match
         Define $\hat{x} = (1/(\vec{g})_{n+1}) \cdot (\vec{g})_{1 \ldots n}$ and $\hat{y} = |S| \cdot ((\vec{g})_{n+1} - 2)$
         Return $(\hat{x}, \hat{y})$
  7. Else:
         Define $z_0 = z_0 z_1 \ldots z_k 0$ and $z_1 = z_0 z_1 \ldots z_k 1$// concat 0 and 1 to $z$
         Return CHECK-MATCH$(z_0) \cup$ CHECK-MATCH$(z_1)$
```

Figure 3: The black-box gradient-based simulator $\mathcal{B}$.

sample set S via its queries to the gradient oracle (i.e., that in Step 3 of $\mathcal{B}$'s description it holds that $D = S$).

At each invocation of the procedure CHECK-MATCH with some prefix z, the algorithm $\mathcal{B}$ queries the oracle $\mathcal{O}_{\mathsf{G}}(S, \cdot, \cdot, \cdot)$ with $(\ell, h, \vec{\theta})$, and obtains a response $\vec{g}$ computed as follows:

$$\vec{g} = \frac{1}{|S|} \sum_{(\vec{x},y) \in S} \nabla_\theta \ell(h(\vec{\theta}, \vec{x}), y) = \frac{1}{|S|} \sum_{(\vec{x},y) \in S} \ell'(h(\vec{\theta}, \vec{x}), y) \cdot \nabla_\theta h(\vec{\theta}, \vec{x}) \ .$$

Denoting by $S_z \subseteq S$ the set of samples $(\vec{x}, y) \in S$ for which $\vec{x}$ is prefixed by z, we obtain:

$$\begin{aligned}
&\frac{1}{|S|} \sum_{(\vec{x},y)\in S} (2+y) \cdot \nabla_\theta h(\vec{\theta}, \vec{x}) \\
&\quad = \frac{1}{|S|} \left(\sum_{(\vec{x},y)\in S_z} (2+y) \cdot \nabla_\theta h(\vec{\theta}, \vec{x}) + \sum_{(\vec{x},y)\in S\setminus S_z} (2+y) \cdot \nabla_\theta h(\vec{\theta}, \vec{x}) \right) \\
&\quad = \frac{1}{|S|} \left(\sum_{(\vec{x},y)\in S_z} (2+y) \cdot ((\vec{x})_1, \ldots, (\vec{x})_n, 1) \right. \\
&\qquad\qquad \left. + \sum_{(\vec{x},y)\in S\setminus S_z} (2+y) \cdot (0, \ldots, 0) \right) \\
&\quad = \frac{1}{|S|} \sum_{(\vec{x},y)\in S_z} (2+y) \cdot ((\vec{x})_1, \ldots, (\vec{x})_n, 1) \\
&\quad = \frac{1}{|S|} \sum_{(\vec{x},y)\in S_z} ((2+y) \cdot (\vec{x})_1, \ldots, (2+y) \cdot (\vec{x})_n, (2+y)) \ ,
\end{aligned}$$

and thus for the response $\vec{g}$ provided by the oracle it holds that

$$\vec{g} = \frac{1}{|S|} \sum_{(\vec{x},y)\in S_z} ((2+y) \cdot (\vec{x})_1, \ldots, (2+y) \cdot (\vec{x})_n, (2+y)) \, .$$

We now distinguish between the following three cases depending on the size of the set S_z:

- **Case I: $|S_z| = 0$.** In this case $\vec{g} = (0, \ldots, 0) \in \mathbb{R}^{n+1}$, and therefore the procedure CHECK-MATCH returns $\perp$ at Step 5.

- **Case II: $|S_z| = 1$.** In this case, for $S_z = \{(\vec{x}', y')\}$ we get:

$$\begin{aligned}
\vec{g} &= \frac{1}{|S|} \sum_{(\vec{x},y)\in S_z} ((2+y) \cdot (\vec{x})_1, \ldots, (2+y) \cdot (\vec{x})_n, (2+y)) \\
&= \frac{1}{|S|} \cdot \big((2+y') \cdot (\vec{x}')_1, \ldots, (2+y') \cdot (\vec{x}')_n, (2+y')\big) \ ,
\end{aligned}$$

  thus the $(n+1)$-th component of the vector $\vec{g}$ will contain $\frac{1}{|S|} \cdot (2+y')$. As $y' \in \{0, 1\}$ we get that $y' < 2$, and in particular $0 < 2 + y' < 2 + 2 = 4$. Therefore, the procedure CHECK-MATCH extracts and returns the pair $(\vec{x}', y')$ at Step 6.

  - For the extraction of y', we multiply the $(n+1)$-th component of $\vec{g}$ by $|S|$ and subtract 2, and we get

$$\hat{y} = |S| \cdot \left(\frac{1}{|S|} \cdot (2+y') \right) - 2 = y' \ .$$

  - For the extraction of $\vec{x}'$ we use the first n components of the vector $\vec{g}$. By multiplying element-wise by $\frac{|S|}{(2+y')}$ we get

$$\hat{x} = \frac{|S|}{(2+y')} \cdot \left(\frac{1}{|S|} \cdot (2+y') \cdot \vec{x}' \right) = \vec{x}' \ .$$

- **Case III: $|S_z| \geq 2$.** Without loss of generality, we will analyze this case for $|S_z| = 2$. Let $S_z = \{(\vec{x}_1, y_1), (\vec{x}_2, y_2)\}$. We have:

  $$\begin{aligned}\vec{g} &= \frac{1}{|S|} \sum_{(\vec{x},y) \in S_z} ((2+y) \cdot (\vec{x})_1, \ldots, (2+y)) \\ &= \frac{1}{|S|} \left(\left((2+y_1) \cdot (\vec{x}_1)_1 + (2+y_2) \cdot (\vec{x}_2)_1\right), \ldots, (4+y_1+y_2) \right)\end{aligned}$$

  We can see that the the $(n+1)$-th component of $\vec{g}$ now contains $4 + y_1 + y_2 \geq 4$ and therefore the procedure CHECK-MATCH will continue the search over longer prefix values at Step 7.

Thus, we have shown that in each call to CHECK-MATCH we either extract and return one sample, return $\perp$, or return nothing and continue the search recursively. By starting from an empty string z, that matches all samples in S, we extend it bit-by-bit until we reach all prefixes of the sample set. This ensures that we extract the entire set S, thus obtaining $D = S$ and proving the correctness of our algorithm.

After establishing correctness, we now bound the number of queries issued by $\mathcal{B}$ as well as $\mathcal{B}$'s running time. As there are $|S|$ distinct samples, the search will reach at most $2|S|$ leaves of the resulting tree: $|S|$ leaves corresponding to the samples, and an additional $|S|$ leaves corresponding to prefixes that differ in their last bit from a sample. The maximal depth of the DFS tree is at most the bit-length n of the samples, and therefore each time we reach a leaf, we visit n nodes on the way. Although some of these nodes may be shared between different samples (meaning they correspond to prefixes that match more than one sample), there are at most $2|S| \cdot n$ visited nodes.

Each one of these node visits corresponds to one run of CHECK-MATCH, in which the simulator $\mathcal{B}$ issues one oracle query and performs some basic arithmetic computations over the output $\vec{g}$. For issuing the oracle query the simulator encodes a parameter vector $\vec{\theta} = (\vec{w}, z) \in \mathbb{R}^{(n+1)+k}$, where k ranges from 0 to n. Therefore, we bound the number of parameters by $O(n)$. The output $\vec{g}$ is of size $n+1$ and thus in total we get that we perform $O(n)$ basic arithmetic computations in each run of CHECK-MATCH. We conclude that by running CHECK-MATCH for $O(|S| \cdot n)$ times the simulator issues at most $Q_{\mathcal{B}} = O(|S| \cdot n) = O(|S| \cdot \log |\mathcal{X}|)$ queries and runs in time at most $T_{\mathcal{B}} = O((|S| \cdot n) \cdot n) = O(|S| \cdot n^2) = O(|S| \cdot \log^2 |\mathcal{X}|)$ (assuming that basic arithmetic operations over n-bits are values are counted at unit cost).

■

5 Statistical Simulation with Fully-Parallelizable Gradient Queries

In Section 4 we established a general feasibility result by presenting a gradient-based adversary that black-box simulates any sample-based one. Although the presented simulator is highly efficient in terms of both its running time and query complexity, it issues its gradient queries in a sequential manner. Over the years, however, parallelization techniques have played an instrumental role in speeding up cryptanalytic tasks. Such techniques have provided parallel algorithms for a variety of cryptanalytic tasks, with notable examples ranging from classic ones such as discrete logarithm algorithms via parallel variants of Pollard's rho method [Pol78, vOW99, BKN+10, BCC+10, BKK+12] and collision finding algorithms via parallel variants of Wagner's generalized birthday attack [Wag02, Ber07, BLN+09] to a wide variety of parallel algorithms for lattice problems [Mic] (for additional examples see also [Nie12, Bos15] and the references therein). Thus, when simulating a sample-based adversary for a cryptanalytic task that may highly benefit from parallelization techniques, a sequential query pattern as exhibited by our gradient-based adversary presented in Section 4 may introduce an undesirable simulation bottleneck.

In this section we show that our sequential simulation approach can be modified to result in a simulator whose gradient queries are fully parallelizable (i.e., all of its gradient queries can be issued within a single round of parallel queries). This comes at a rather minor cost of statistical simulation instead of perfect simulation (i.e., simulation within statistical distance ϵ for any fixed $\epsilon > 0$ as captured by Definition 3.4) together with a slight increase in the running time and query complexity of the simulator for some ranges of the parameters (depending logarithmically on $1/\epsilon$ and thus enabling to efficiently support negligible values of ϵ). As we show below, for most natural choices of the parameters, the multiplicative overhead compared to our sequential simulator is in fact constant.

As in Section 4, for simplicity here we state and prove our result for samples over $\mathcal{X} \times \mathcal{Y}$ where $\mathcal{X} = \{0,1\}^n$ for some integer $n \in \mathbb{N}$ and $\mathcal{Y} = \{0,1\}$. We prove the following theorem:

Theorem 5.1. *Let $\mathcal{X} = \{0,1\}^n$ for some $n \geq 1$, and let $\mathcal{Y} = \{0,1\}$. For any $\epsilon > 0$ there exists a gradient-based adversary $\mathcal{B}$ that black-box ϵ-simulates all sample-based adversaries with respect to $\mathcal{X} \times \mathcal{Y}$, where:*

- *$\mathcal{B}$ runs in time $T_{\mathcal{B}}(|S|, |\mathcal{X}|, |\mathcal{Y}|) = O((|S| \cdot \log(|S|/\epsilon) \cdot \log |\mathcal{X}|)$.*
- *$\mathcal{B}$ issues $Q_{\mathcal{B}}(|S|, |\mathcal{X}|, |\mathcal{Y}|) = O(|S| \cdot \log(|S|/\epsilon))$ parallel queries to the gradient oracle, followed by a single query to the simulated sample-based adversary.*

Recall that our sequential simulator provided by Theorem 4.1 runs in time $O(|S| \cdot \log^2 |\mathcal{X}|)$ and issues $O(|S| \cdot \log |\mathcal{X}|)$ gradient queries. Thus, in terms of both the running time and query complexity, the performance of our parallel simulator provided by Theorem 5.1 matches that of our sequential simulator within a multiplicative gap of $\log(|S|/\epsilon)/\log |\mathcal{X}|$. Recall that $|S| \leq |\mathcal{X}| = \{0,1\}^n$, and therefore even for an exponentially-small $\epsilon = \exp(-\Omega(n))$ this multiplicative gap is constant.

For the more general setting in which $|\mathcal{Y}| = m > 1$ for some integer $m \in \mathbb{N}$, our simulator $\mathcal{B}$ runs in time $T_{\mathcal{B}}(|S|, |\mathcal{X}|, |\mathcal{Y}|) = O(|S| \cdot \log(|S|/\epsilon) \cdot (\log |\mathcal{X}| + \log |\mathcal{Y}|))$ and issues the same number of queries $Q_{\mathcal{B}}(|S|, |\mathcal{X}|, |\mathcal{Y}|) = O(|S| \cdot \log(|S|/\epsilon))$. We refer the reader to Section 7 for the extended result.

In what follows we first provide a high-level technical overview of our proof of Theorem 5.1 when compared to that of Theorem 4.1, and then provide its formal proof.

Proof overview. Recall that our sequential simulation approach, as detailed in Section 4, relies on the observation that gradient queries are sufficiently expressive for efficiently extracting the entire set of samples. Specifically, we observed that, for any sample set $S = \{(x_i, y_i)\}_{i \in [s]}$ and for any given prefix z, a single gradient query enables to distinguish between the case in which there are no samples x_i that are prefixed with z, the case in which there is exactly one sample x_i that is prefixed with z, and the case in which there is more than one sample x_i that is prefixed with z. We then used this observation to realize a recursive and *completely deterministic* DFS-based exploration of the sample set S, leading to a highly sequential process for extracting all samples (e.g., whenever two samples share a prefix of length ℓ, then our simulator must issue at least ℓ sequential queries in order to extract them).

A natural approach for obtaining a less sequential process is to rely on randomization, and a naive attempt would be to guess a prefix z, and hope that it matches exactly one sample (this is obviously fully parallelizable). Unfortunately, on the one hand, for n-bit samples x_i, guessing a rather short prefix (say, of length $O(\log n)$ bits) may not enable to isolate any sample (e.g., consider a sample set S in which all samples x_i share the same $n/2$-bit prefix). On the other hand, guessing a rather long prefix (say, of length $\Omega(n)$ bits) may lead to an exponentially small probability of even hitting the prefix of any sample. Thus, this naive attempt completely fails.

Our approach relies on a more subtle form of randomization, where instead of guessing a prefix, we guess an output of a hash function $M : \{0,1\}^n \to \{0,1\}^k$ that is independently sampled for each gradient query from a pairwise-independent function family. Recall that the pairwise independence guarantee provided by such a function family is that for any $x_i \neq x_j$ it holds that the random variables $M(x_i)$ and $M(x_j)$ are independent and uniformly distributed over the choice of the function M from the given function family. This guarantees that, even for a short output length $k = O(\log n)$, each sample x_i would be eventually isolated from all other samples, except any some pre-specified failure probability ϵ that would determine the number of queries. Moreover, since the output length is short, we can guess the output $z = M(x_i)$ of the hash function with a sufficiently high probability and include it in the gradient query to enable the extraction of the sample.

From a more technical perspective, for each gradient query our simulator samples a string $z \in \{0,1\}^k$ and a function $M : \{0,1\}^n \to \{0,1\}^k$ from a pairwise independent function family $\mathcal{M}_{n,k}$. For a vector of parameters $\vec{w} \in \mathbb{R}^{n+1}$, we define $\vec{\theta} = (\vec{w}, z, M)$, where we assume M can be encoded using $\ell(n,k) = O(n+k)$ bits, and define z and M to be non-differentiable parameters. We remind that although the exact value of $\vec{w}$ is not important, for consistency we will define it as the all-zeros vector $(0,\dots,0) \in \mathbb{R}^{n+1}$. The function h is defined as follows:

$$\begin{aligned} h(\vec{\theta}, \vec{x}) &= \vec{w} \cdot ((\vec{x})_1, \dots, (\vec{x})_n, 1) \cdot \mathbb{1}_{M(x)=z} \\ &= (w_1(\vec{x})_1, \dots, w_n(\vec{x})_n, w_{n+1}) \cdot \mathbb{1}_{M(x)=z} \,. \end{aligned}$$

The simulator then issues an oracle query $(\ell, h, \vec{\theta})$, for which it holds that

$$\nabla_\theta h(\vec{\theta}, \vec{x}) = \begin{cases} (0,\dots,0) \in \mathbb{R}^{n+1} & \text{if } M(x) \neq z \\ ((\vec{x})_1, \dots, (\vec{x})_n, 1) \in \mathbb{R}^{n+1} & \text{otherwise} \end{cases}$$

As we noted in Section 4, we denote by ∇_θ the gradient with respect to all differentiable parameters in $\vec{\theta}$, in this case this means $\vec{w}$ only. As these queries are now independent of each other, they can all be issued in parallel. Our analysis below shows that, for any $\epsilon > 0$, issuing $T = O(|S| \cdot \log(|S|/\epsilon))$ such queries guarantees that all samples are extracted except with probability ϵ.

Proof of Theorem 5.1. Let $\mathcal{X} = \{0,1\}^n$ for some $n \geq 1$, $\mathcal{Y} = \{0,1\}$, and let $\epsilon > 0$. We show that the gradient-based simulator $\mathcal{B}$ described in Figure 4 black-box ϵ-simulates all sample-based adversaries with respect to $\mathcal{X} \times \mathcal{Y}$. In what follows we first prove the correctness of the simulation, and then analyze the running time and query complexity of $\mathcal{B}$.

Let $\mathcal{D}$ be a distribution over $(\mathcal{X} \times \mathcal{Y})^*$. For proving that

$$\mathrm{SD}\left(\left(S, \mathcal{A}\left(S\right)\right), \left(S, \mathcal{B}^{\mathcal{O}_{\mathsf{G}}(S,\cdot,\cdot,\cdot),\mathcal{A}(\cdot)}\left(1^{|S|}\right)\right)\right) \leq \epsilon,$$

where $S \leftarrow \mathcal{D}$ in both distributions, we in fact prove a stronger statement. We prove that for any $s \geq 1$ and for any set $S \subseteq (\mathcal{X} \times \mathcal{Y})^s$, the output produced by the computation $\mathcal{B}^{\mathcal{O}_{\mathsf{G}}(S,\cdot,\cdot,\cdot),\mathcal{A}(\cdot)}(1^s)$ is distributed as follows: With probability at most ϵ over the internal randomness of $\mathcal{B}$ it outputs $\perp$, and otherwise it is identical to the distribution of the output produced by the computation $\mathcal{A}(S)$. For this purpose, it suffices to show that, for any such set S, with probability at least $1-\epsilon$ over the internal randomness of $\mathcal{B}$, the algorithm $\mathcal{B}$ is able to extract the sample set S via its queries to the gradient oracle.

For every $t \in [T]$ we denote by $z_t \in \{0,1\}^k$ and $M_t \in \mathcal{M}_{n,k}$ the random variables corresponding to the bit-string and the function sampled at iteration t, respectively. For every $i \in [s]$ and for each iteration $t \in [T]$, we say that the sample $(\vec{x}_i, y_i)$ is *isolated* by the t-th iteration if $M_t(\vec{x}_i) = z_t$ and

```
The black-box gradient-based simulator $\mathcal{B}^{\mathcal{O}_{\mathsf{G}}(S,\cdot,\cdot,\cdot),\mathcal{A}(\cdot)}(1^s)$:
  1. Set $k = \lceil \log_2(2s) \rceil$ and $T = \lceil 16s \cdot \ln(s/\epsilon) \rceil$.
  2. Let $\mathcal{M}_{n,k}$ be a pairwise-independent function family, where $M : \{0,1\}^n \to \{0,1\}^k$ for every $M \in \mathcal{M}_{n,k}$.
  3. Set $\ell(\hat{y}, y) = (2 + y) \cdot \hat{y}$.
  4. Set $D = \emptyset$.
  5. For $t \in \{1, \ldots, T\}$: // can be performed in parallel
      5.1. Sample $z \xleftarrow{\$} \{0,1\}^k$ and $M \xleftarrow{\$} \mathcal{M}_{n,k}$.
      5.2. Define $\vec{w} = (0, \ldots, 0) \in \mathbb{R}^{n+1}$ and set $\vec{\theta} = (\vec{w}, z, M)$.
      5.3. Define $h(\vec{\theta}, \vec{x}) = \vec{w} \cdot ((\vec{x})_1, \ldots, (\vec{x})_n, 1) \cdot \mathbb{1}_{M(x)=z}$.
      5.4. Obtain $\vec{g} = \mathcal{O}_{\mathsf{G}}(S, \ell, h, \vec{\theta}_t) \in \mathbb{R}^{n+1}$.
      5.5. If $0 < |S| \cdot (\vec{g})_{n+1} < 4$: // single match
          i. Define $\hat{x} = (1/(\vec{g})_{n+1}) \cdot (\vec{g})_{1 \ldots n}$ and $\hat{y} = |S| \cdot ((\vec{g})_{n+1} - 2)$.
          ii. Update $D = D \cup \{(\hat{x}, \hat{y})\}$.
  6. If $|D| < s$ then output $\perp$, and otherwise obtain $v \leftarrow \mathcal{A}(D)$ and output $v$.
```

Figure 4: The black-box gradient-based simulator $\mathcal{B}$.

for every $j \in [s] \setminus \{i\}$ it holds that $M_t(x_i) \neq M_t(x_j)$. For every $i \in [s]$ we let $\mathsf{Failure}_i$ denote the event in which the sample $(\vec{x}_i, y_i)$ is not isolated by any iteration, and we let $\mathsf{Failure} = \bigcup_{i \in [s]} \mathsf{Failure}_i$.

We will now show that if the event $\mathsf{Failure}$ does not occur, then the gradient-based simulator extracts the entire set S (and otherwise outputs $\perp$). For any iteration $t \in [T]$, the response of the gradient oracle is as follows:

$$\begin{aligned}
\vec{g} &= \frac{1}{|S|} \sum_{(\vec{x},y) \in S} \nabla_\theta \ell(h(\vec{\theta}, \vec{x}), y) \\
&= \frac{1}{|S|} \sum_{(\vec{x},y) \in S} \ell'(h(\vec{\theta}, \vec{x}), y) \cdot \nabla_\theta h(\vec{\theta}, \vec{x}) \\
&= \frac{1}{|S|} \sum_{(\vec{x},y) \in S} (2 + y) \cdot \nabla_\theta h(\vec{\theta}, \vec{x}) \ .
\end{aligned}$$

Denoting by $S_{M,z} \subseteq S$ the set of samples $(\vec{x}, y) \in S$ for which it holds that $M(x) = z$, we obtain

$$
\begin{aligned}
&\frac{1}{|S|} \sum_{(\vec{x},y)\in S} (2+y) \cdot \nabla_\theta h(\vec{\theta}, \vec{x}) \\
&\quad = \frac{1}{|S|} \left(\sum_{(\vec{x},y)\in S_{M,z}} (2+y) \cdot \nabla_\theta h(\vec{\theta}, \vec{x}) + \sum_{(\vec{x},y)\in S\setminus S_{M,z}} (2+y) \cdot \nabla_\theta h(\vec{\theta}, \vec{x}) \right) \\
&\quad = \frac{1}{|S|} \left(\sum_{(\vec{x},y)\in S_{M,z}} (2+y) \cdot ((\vec{x})_1, \dots, (\vec{x})_n, 1) \right. \\
&\qquad\qquad \left. + \sum_{(\vec{x},y)\in S\setminus S_{M,z}} (2+y) \cdot (0, \dots, 0) \right) \\
&\quad = \frac{1}{|S|} \sum_{(\vec{x},y)\in S_{M,z}} (2+y) \cdot ((\vec{x})_1, \dots, (\vec{x})_n, 1) \\
&\quad = \frac{1}{|S|} \sum_{(\vec{x},y)\in S_{M,z}} ((2+y) \cdot (\vec{x})_1, \dots, (2+y) \cdot (\vec{x})_n, (2+y)) \ ,
\end{aligned}
$$

and thus for the response $\vec{g}$ provided by the oracle it holds that

$$
\vec{g} = \frac{1}{|S|} \sum_{(\vec{x},y)\in S_{M,z}} ((2+y) \cdot (\vec{x})_1, \dots, (2+y) \cdot (\vec{x})_n, (2+y)) \, .
$$

We now distinguish between the following three cases depending on the size of the set $S_{M,z}$:

- **Case I: $|S_{M,z}| = 0$.** In this case $\vec{g} = (0, \dots, 0) \in \mathbb{R}^{n+1}$ and therefore the condition in Step 5.5. is not satisfied, and the iteration will end with no samples being extracted.

- **Case II: $|S_{M,z}| = 1$.** In this case it holds that

$$
\begin{aligned}
\vec{g} &= \frac{1}{|S|} \sum_{(\vec{x},y)\in S_{M,z}} ((2+y) \cdot (\vec{x})_1, \dots, (2+y) \cdot (\vec{x})_n, (2+y)) \\
&= \frac{1}{|S|} \cdot \left((2+y') \cdot (\vec{x}')_1, \dots, (2+y') \cdot (\vec{x}')_n, (2+y') \right) \ ,
\end{aligned}
$$

  for $S_{M,z} = \{(\vec{x}', y')\}$. The $(n+1)$-th component of the vector $\vec{g}$ contains $\frac{1}{|S|} \cdot (2+y')$. As $y' \in \{0,1\}$ we get that $y' < 2$, and in particular $0 < 2 + y' < 2 + 2 = 4$. Therefore, the condition in Step 5.5. is satisfied and we will extract the pair $(\vec{x}', y')$ as follows:

  - For the extraction of y', we multiply the $(n+1)$-th component of $\vec{g}$ by $|S|$ and subtract 2, and we get

$$
\hat{y} = |S| \cdot \left(\frac{1}{|S|} \cdot (2+y') \right) - 2 = y' \ .
$$

  - For the extraction of $\vec{x}'$ we use the first n components of the vector $\vec{g}$ which contains $\frac{1}{|S|} \cdot (2+y') \cdot \vec{x}'$. By multiplying element-wise by $\frac{|S|}{(2+y')}$ we get

$$
\hat{x} = \frac{|S|}{(2+y')} \cdot \left(\frac{1}{|S|} \cdot (2+y') \cdot \vec{x}' \right) = \vec{x}' \ .
$$

- **Case III: $|S_{M,z}| \geq 2$.** Without loss of generality, we analyze this case for $|S_{M,z}| = 2$. In this case,

$$\begin{aligned}\vec{g} &= \frac{1}{|S|} \sum_{(\vec{x},y)\in M} ((2+y)(\vec{x})_1, \ldots, (2+y)) \\ &= \frac{1}{|S|} \left(\left((2+y_1) \cdot (\vec{x}_1)_1 + (2+y_2) \cdot (\vec{x}_2)_1 \right), \ldots, (4+y_1+y_2) \right) \ ,\end{aligned}$$

for $S_{M,z} = \{(\vec{x}_1, y_1), (\vec{x}_2, y_2)\}$. We can see that the the $(n+1)$-th component of $\vec{g}$ now contains $4 + y_1 + y_2 \geq 4$ and therefore the condition in Step 5.5. is not satisfied and the iteration will end with no samples being extracted.

We now show that the event $\mathsf{Failure}$ occurs with probability at most ϵ, and this finalizes the correctness of the simulation. From the definition of the events $\{\mathsf{Failure}_i\}_{i\in[s]}$, for every $i \in [s]$ it holds that

$$\begin{aligned}\Pr\left[\mathsf{Failure}_i\right] &= \left(\Pr\left[(\vec{x}_i, y_i) \text{ is not isolated by the first iteration}\right]\right)^T \\ &= \left(\Pr_{z_1, M_1}\left[\, \exists j \in [s] \setminus \{i\} : M_1(x_i) = M_1(x_j) \vee z_1 \neq M_1(x_i) \,\right]\right)^T \\ &= \left(1 - \Pr_{z_1, M_1}\left[\, \nexists j \in [s] \setminus \{i\} : M_1(x_i) = M_1(x_j) \wedge z_1 = M(x_i) \,\right]\right)^T \ ,\end{aligned}$$

where the first equality follows from the independence across all iterations. Using the union bound, we obtain

$$\begin{aligned}&\Pr_{M_1}\left[\, \nexists j \in [s] \setminus \{i\} : M_1(x_i) = M_1(x_j) \,\right] \\ &\qquad = 1 - \Pr_{M_1}\left[\, \exists j \in [s] \setminus \{i\} : M_1(x_i) = M_1(x_j) \,\right] \\ &\qquad \geq 1 - \sum_{j\in[s]\setminus\{i\}} \Pr_{M_1}\left[\, M_1(x_i) = M_1(x_j) \,\right] \\ &\qquad = 1 - \sum_{j\in[s]\setminus\{i\}} \frac{1}{2^k} = 1 - (s-1) \cdot \frac{1}{2^k} \ ,\end{aligned}$$

where the last equality follows from the pairwise-independence of the function family $\mathcal{M}_{n,k}$. Since z_1 is uniformly sampled and is independent of M_1, the probability of a match between z_1 and $M_1(x_i)$, given that there are no collisions can be computed as

$$\Pr_{z_1, M_1}\left[\, z_1 = M_1(x_i) \ \mid \nexists j \in [s] \setminus \{i\} : M_1(x_i) = M_1(x_j) \,\right] = \frac{1}{2^k} \ .$$

Therefore,

$$\Pr\left[\mathsf{Failure}_i\right] \leq \left(1 - \left(1 - \frac{s-1}{2^k}\right) \cdot \frac{1}{2^k}\right)^T = \left(1 - \left(\frac{1}{2^k} - \frac{s-1}{2^{2k}}\right)\right)^T \ ,$$

and our choice of $k = \lceil \log_2(2s) \rceil$ guarantees that

$$\frac{1}{2^k} - \frac{s-1}{2^{2k}} \geq \frac{s}{2^{2k}} \geq \frac{1}{16s} \ ,$$

implying

$$\Pr\left[\mathsf{Failure}_i\right] \leq \left(1 - \frac{1}{16s}\right)^T \leq e^{-T/16s} .$$

We conclude by observing that our choice of $T = \lceil 16s \cdot \ln(s/\epsilon) \rceil$ now implies

$$\Pr\left[\,\mathsf{Failure}\,\right] = \Pr\left[\bigcup_{i \in [s]} \mathsf{Failure}_i\right] \leq s \cdot e^{-T/16s} \leq \epsilon .$$

After establishing correctness, we now bound the number of queries issued by $\mathcal{B}$ as well as $\mathcal{B}$'s running time. In each iteration the simulator issues a single oracle query and performs some basic arithmetic computations over the output $\vec{g} \in \mathbb{R}^{n+1}$. For issuing the oracle query, the simulator encodes the parameter vector $\vec{\theta} = (\vec{w}, z, M) \in \mathbb{R}^{(n+1)+k+\ell(n,k)}$ with $\ell(n,k) = O(n+k)$ being the description length of the pairwise functions. As we defined $k = O(\log |S|)$, we can bound the number of parameters, and correspondingly the runtime of a single iteration, by $O(\log |\mathcal{X}| + \log |S|) = O(\log |\mathcal{X}|)$. By running for $T = O(|S| \cdot \log(|S|/\epsilon))$ iterations we get a total runtime of $T_{\mathcal{B}} = O(|S| \cdot \log(|S|/\epsilon) \cdot \log |\mathcal{X}|)$ and query complexity of $Q_{\mathcal{B}} = O(|S| \cdot \log(|S|/\epsilon))$.

■

6 Statistical Simulation via Gradient Descent

In this section we show that all sample-based adversaries can be simulated not only using general gradient-based methods (as established by Theorems 4.1 and 5.1), but in fact using the particular gradient-descent (GD) algorithm. Recall that whereas our notion of a gradient-based adversary allows querying the gradient oracle in a rather arbitrary manner, our notion of a GD-based adversary forces the application of a specific update rule for each query based on the response provided to the previous one. As discussed in Section 3, this restriction models the GD algorithm, which is considered the most common learning methodology.

Specifically, as detailed in Figure 2, a GD-based algorithm starts by initializing the parameters θ according to some predefined distribution. Then, in each iteration, the algorithm updates these parameters based on the current gradient information $\vec{g}$ (which, in our case, is received from the gradient oracle) for "directing" the update towards parameters that minimize the loss function (and thus correspond to a better estimate). In other words, at every iteration $t \in [T]$, given the current parameters $\vec{\theta}^{(t-1)}$ and the gradient vector $\vec{g}$, the algorithm updates the parameters by computing

$$\vec{\theta}^{(t)} = \vec{\theta}^{(t-1)} - \eta \cdot \vec{g} ,$$

where η is the algorithm's "step size" (also known as the "learning rate") which determines the size of the steps taken in the direction of the gradient.

Our result in this section shows that, even in this significantly more restricted setting, we can construct a GD-based adversary $\mathcal{B}_{\mathsf{GD}}$ that black-box ϵ-simulates any sample-based one. As in Sections 4 and 5, for simplicity here we state and prove our result for samples over $\mathcal{X} \times \mathcal{Y}$ where $\mathcal{X} = \{0,1\}^n$ for some integer $n \in \mathbb{N}$ and $\mathcal{Y} = \{0,1\}$. We prove the following theorem:

Theorem 6.1. *Let $\mathcal{X} = \{0,1\}^n$ for some $n \geq 1$, and let $\mathcal{Y} = \{0,1\}$. For any $\epsilon > 0$ there exists a GD-based adversary $\mathcal{B}_{\mathsf{GD}}$ that black-box ϵ-simulates all sample-based adversaries with respect to $\mathcal{X} \times \mathcal{Y}$, where:*

- *$\mathcal{B}_{\mathsf{GD}}$ runs in time $T_{\mathcal{B}_{\mathsf{GD}}}(|S|, |\mathcal{X}|, |\mathcal{Y}|) = O((|S| \cdot \log(|S|/\epsilon))^2 \cdot \log |\mathcal{X}|)$.*

- $\mathcal{B}_{\mathsf{GD}}$ *issues* $Q_{\mathcal{B}_{\mathsf{GD}}}(|S|, |\mathcal{X}|, |\mathcal{Y}|) = O(|S| \cdot \log(|S|/\epsilon))$ *queries to the gradient oracle, followed by a single query to the simulated sample-based adversary.*

For the more general setting in which $|\mathcal{Y}| = m > 1$ for some integer $m \in \mathbb{N}$, our simulator $\mathcal{B}$ runs in time $T_{\mathcal{B}}(|S|, |\mathcal{X}|, |\mathcal{Y}|) = O((|S| \cdot \log(|S|/\epsilon))^2 \cdot (\log |\mathcal{X}| + \log |\mathcal{Y}|))$ and issues the same number of queries $Q_{\mathcal{B}}(|S|, |\mathcal{X}|, |\mathcal{Y}|) = O(|S| \cdot \log(|S|/\epsilon))$. We refer the reader to Section 7 for the extended result.

In the remainder of this section, we provide an overview of this proof and explaining the main ideas underlying our GD-based simulator described in Figure 5, and then provide its formal proof.

```
The black-box GD-based simulator $\mathcal{B}_{\mathsf{GD}}^{\mathcal{O}_{\mathsf{G}}(S,\cdot,\cdot,\cdot),\mathcal{A}(\cdot)}(1^s)$:
  1. Set $k = \lceil \log_2(2s) \rceil$, $T = \lceil 16s \cdot \ln(s/\epsilon) \rceil$, and $\eta = -s$.
  2. Set $\ell(\hat{y}, y) = (2 + y) \cdot \hat{y}$.
  3. Initialize $\vec{\theta}^{(0)} = (\vec{\theta}_1^{(0)}, \ldots, \vec{\theta}_T^{(0)})$, such that for all $t \in [T]$ we denote by $\vec{\theta}_t^{(0)}$ the set of parameters
     $(\vec{w}_t^{(0)}, z_t, M_t, \kappa_t^{(0)})$ initialized as follows:
       • $\vec{w}_t^{(0)} = (0, \ldots, 0) \in \mathbb{R}^{n+1}$
       • $z_t \in \{0,1\}^k$
       • $M_t \in \mathcal{M}_{n,k}$
       • $\kappa_t^{(0)} = 0$
  4. Define $h((\vec{w}, z, M, \kappa), \vec{x}) = (\vec{w} \cdot ((\vec{x})_1, \ldots, (\vec{x})_n, 1)) \cdot \mathbb{1}_{M(x)=z} + \kappa$.
  5. Define $\tilde{h}(\vec{\theta}, \vec{x}) = \sum_{j=1}^{T} \phi\left(\kappa_{j-1}, \kappa_j, h(\vec{\theta}_j, \vec{x})\right)$.
  6. For $t \in \{1, \ldots, T\}$: // Running GD for $T$ iterations
     6.1 Obtain $\vec{g} = \mathcal{O}_{\mathsf{G}}(S, \ell, \tilde{h}, \vec{\theta}^{(t-1)}) \in \mathbb{R}^{|\vec{\theta}^{(t-1)}|}$
     6.2 Update $\vec{\theta}^{(t)} = \vec{\theta}^{(t-1)} - \eta \cdot \vec{g}$.
  7. Set $D = \emptyset$.
  8. For $t \in \{1, \ldots, T\}$: // decode extracted samples from parameters $\vec{w}_1^{(T)}, \ldots, \vec{w}_T^{(T)}$
     8.1 If $0 < \vec{w}_t^{(T)} < 4$:
         i. Define $\hat{x} = (1/(\vec{w}_t^{(T)})_{n+1}) \cdot (\vec{w}_t^{(T)})_{1 \ldots n}$ and $\hat{y} = (\vec{w}_t^{(T)})_{n+1} - 2$.
         ii. Update $D = D \cup \{(\hat{x}, \hat{y})\}$.
  9. Compute $v \leftarrow \mathcal{A}(D)$ and output $v$.
```

Figure 5: The black-box gradient-based simulator $\mathcal{B}_{\mathsf{GD}}$.

Proof overview. The main challenge underlying the proof of Theorem 6.1 stems from the "non-adaptive" nature of the GD updates. In Sections 4 and 5, we extracted samples after each query, for queries that isolated a single sample. Here, we are allowed to extract samples only after completing the entire execution of the GD algorithm, meaning concluding all T iterations. Towards this goal, we extend our parameters so that they can store all of the gradient oracle's responses, and we extract all samples at once. This requires the size of the parameters to grow from $O(\log |\mathcal{X}|)$ to $O(T \cdot \log |\mathcal{X}|)$. In addition, in our previous constructions we defined the parameters $\vec{\theta}$ to include a vector of additional parameters $\vec{w} \in \mathbb{R}^{n+1}$ that enabled to obtain the desired gradient behavior. As we only used $\vec{w}$ for its gradient vector, any specific choice of $\vec{w}$ could be used, and for consistency we used the all-zeros vector $\vec{w} = (0, \ldots, 0) \in \mathbb{R}^{n+1}$. In the GD setting, we can no longer extract the samples directly from the value of $\vec{g}$, and must write the value of $\vec{g}$ to that of $\vec{w}$ in each iterations. Therefore, here we require $\vec{w}$ to be initialized as the all-zeros vector.

Another main difference compared to Section 5 lies in the selection of the string z and the pairwise independent function M in each of the T iterations. In the gradient-based construction the adversary samples a fresh pair of (z, M) in each iteration and encodes them in the parameters $\vec{\theta}$ so they can be used by the gradient oracle $\mathcal{O}_{\mathsf{G}}$, i.e., $\vec{\theta} = (\vec{w}, z, M)$. In the GD-based setting this is no longer possible, as the GD update rule does not allow the adversary to encode arbitrary new values into the parameters $\vec{\theta}$ in each iteration. As such, the sampling of the z's and M's for all iterations must be made in advance. Namely, our GD-based adversary running for T iterations will sample in advance T pairs (z, M) and encode all of them at once in the initial parameters $\vec{\theta}$. However, in order to use a different set of parameters $(z, M, \vec{w})$ in each iteration, a "clock" mechanism must be introduced, to direct the computation to use only the relevant parameters at each iteration. For this, following Abbe et al. [AKM$^+$21], we include in each set of parameters $(z, M, \vec{w})$ an additional bit κ indicating whether this set was already used in some iteration. We initialize $\kappa = 0$, and after using the corresponding set of parameters, the value of κ is updated to a value greater or equal to 2. At each iteration, we only use the *first* set of parameters for which $\kappa = 0$.

More formally, we construct the model function and model parameters as follows. For any parameter p, we denote by $p^{(t)}$ the value of the parameter at the end of iteration t. Then, for every $t \in [T]$ the adversary initializes a set of parameters $\vec{\theta}_t^{(0)} = (\vec{w}_t^{(0)}, z_t, M_t, \kappa_t^{(0)})$ such that $\vec{w}_t^{(0)} = (0, \ldots, 0) \in \mathbb{R}^{n+1}, z_t \in \{0,1\}^k$, $M_t \in \mathcal{M}_{n,k}$, and $\kappa_t^{(0)} = 0$. Similarly to the construction in Figure 4, we define the parameters z and M to be non-differentiable, and as such their value does not change between different iterations. We get that the entire set of parameters at initialization time is of the form:

$$\vec{\theta}^{(0)} = \left(\vec{\theta}_1^{(0)}, \ldots, \vec{\theta}_T^{(0)}\right) .$$

As the GD update rule requires the subtraction of the gradient vector $\vec{g}$ from the vector of parameters $\vec{\theta}$, we modify our notation of the gradient. In our previous results we denoted by ∇_θ the gradient with respect to only the differentiable parameters in $\vec{\theta}$, so that when $\vec{\theta}$ is composed of p_{diff} differentiable parameters and $p_{\mathsf{non\text{-}diff}}$ non-differentiable ones , we get that $|\vec{g}| = p_{\mathsf{diff}}$. Here we will denote by ∇_θ the gradient with respect to the differentiable parameters, and in positions corresponding to non-differentiable parameters we will return 0, and we get $|\vec{g}| = p_{\mathsf{diff}} + p_{\mathsf{non\text{-}diff}}$.

Let $\phi : \mathbb{R}^3 \to \mathbb{R}$ be a differentiable function such that

$$\phi(\alpha_1, \alpha_2, \alpha_3) = \begin{cases} \alpha_3 & \alpha_1 \geq 2 \wedge \alpha_2 = 0 \\ 0 & \alpha_1 = \alpha_2 \\ * & \text{otherwise} \end{cases} ,$$

where the "$*$" symbol may correspond to any arbitrary values that ensure the continuity of the function, required for it to be differentiable. Then, we define the differentiable model $\tilde{h}$ as follows:

$$\tilde{h}(\vec{\theta}, \vec{x}) = \sum_{j=1}^{T} \phi\left(\kappa_{j-1}, \kappa_j, h(\vec{\theta}_j, \vec{x})\right) ,$$

where h is the differentiable function defined as in Figure 4, and we set $\kappa_0 = 2$. We prove that this construction indeed provides the clock mechanism described above, such that when running GD for

T iterations we get that in each iteration we initiate an independent evaluation of h on a fresh set of parameters $\vec{\theta}_t$.

It is important to note that while we extended the construction of our fully-parallelizable gradient-based adversary, the construction here is not necessarily parallelizable. This is due to GD being an *iterative* algorithm in its nature, where a fully-parallelizable solution is less intuitive. The construction in Figure 4 can be viewed as a one-step parallelizable variant of GD. The DFS-based construction described in Figure 3 does not fit the GD-based construction as all "guesses" must be made in advance and in a non-adaptive manner.

Proof of Theorem 6.1. For proving the correctness of the construction described in Figure 5 we need to first prove the correctness of the clock mechanism. That is, to show that at each iteration $t \in [T]$, only a single set of unused parameters is utilized, and all other parameters remain unmodified. To conclude the correctness, we will need to also show that after running GD for T iterations using the proposed construction, we get that for each pair $(\vec{x}, y) \in S$ and with some probability, there exists a $t \in [T]$ such that $\vec{w}_t^{(T)}$ "encodes" this pair, and we can extract it as described in the gradient-based construction provided in Section 5.

For the clock mechanism, we show that after every iteration $t \in [T]$ of GD we get:

- **Condition 1: future parameters.** For every $t < j \leq T$: $\vec{w}_j^{(t)} = \vec{w}_j^{(0)} = 0^{n+1}$, and $\kappa_j^{(t)} = \kappa_j^{(0)} = 0$. In other words, this means that by the end of iteration t, parameters that have yet to be used remain in their initial value.

- **Condition 2: past parameters.** For every $0 < j < t$: $w_j^{(t)} = w_j^{(j)}$, and $\kappa_j^{(t)} \geq 2$. In other words, this means that the value of parameters used by previous iterations remains unchanged by the end of iteration t.

- **Condition 3: current parameters.** For $j = t$ we have $\kappa_j^{(t)} \geq 2$ and

$$\vec{w}_j^{(t)} = \begin{cases} (0, \ldots, 0) \in \mathbb{R}^{n+1} & \text{when } |S_{z_t, M_t}| = 0 \\ \sum_{(\vec{x}, y) \in S_{z_t, M_t}} (2 + y) \cdot ((\vec{x})_1, \ldots, (\vec{x})_n, 1) & \text{when } |S_{z_t, M_t}| \geq 1 \end{cases}$$

  where $S_{z_t, M_t} \subseteq S$ is defined to be the set of all samples $(\vec{x}, y) \in S$ for which $M_t(x)$ match z_t, i.e., $\forall (\vec{x}, y) \in S_{z_t, M_t}$: $M_t(x) = z_t$. In other words, the parameters used at iteration t will be modified by its end so that the value of κ_j be set to a value greater or equal to 2 and the value of $\vec{w}_j$ contains the encoding of samples matched at this step.

Let $\mathcal{M}_{n,k}$ be any pairwise-independent function family, where $M : \{0,1\}^n \to \{0,1\}^k$ for every $M \in \mathcal{M}_{n,k}$, and let $S_{z,M} = \{(\vec{x}, y) \in S | M(x) = z\} \subseteq S$. For proving the statement above, we provide the following lemma:

Lemma 6.2. There exists a differentiable model $h(\vec{\theta}, \cdot)$, with $\vec{\theta} = (\vec{w}, z, M, \kappa)$, such that running GD for one step, from an initialization $\vec{\theta}^{(0)} = (\vec{w}^{(0)}, z, M, \kappa^{(0)})$ satisfying $\vec{w}^{(0)} = (0, \ldots, 0) \in \mathbb{R}^{n+1}$, $\kappa^{(0)} = 0$, $z \in \{0,1\}^k$ and $M \in \mathcal{M}_{n,k}$, where z, M are non-differentiable parameters, yields parameters $\vec{\theta}^{(1)} = (\vec{w}^{(1)}, z, M, \kappa^{(1)})$ such that we get:

1. $\vec{w}^{(1)} = \begin{cases} (0, \ldots, 0) \in \mathbb{R}^{n+1} & \text{when } |S_{z,M}| = 0 \\ \sum_{(\vec{x}, y) \in S_{z,M}} (2 + y) \cdot ((\vec{x})_1, \ldots, (\vec{x})_n, 1) \in \mathbb{R}^{n+1} & \text{when } |S_{z,M}| \geq 1 \end{cases}$

2. $\kappa^{(1)} \geq 2$.

Proof of Lemma 6.2. We define a differentiable model similar to the one defined in Figure 4:

$$h(\vec{\theta},\vec{x}) = (\vec{w}\cdot((\vec{x})_1,\dots,(\vec{x})_n,1))\cdot \mathbb{1}_{M(x)=z} + \kappa \ .$$

As before, we get that the gradient with respect to $\vec{w}$ is:

$$\nabla_{\vec{w}} h(\vec{\theta},\vec{x}) = \begin{cases} (0,\dots,0) & \text{if } M(x)\neq z \\ ((\vec{x})_1,\dots,(\vec{x})_n,1) & \text{otherwise} \end{cases} \in \mathbb{R}^{n+1}$$

Therefore, by setting the learning rate $\eta = -s$, using the loss function ℓ defined in Figure Algorithm 4, and performing one step of GD we get:

$$\begin{aligned}
\vec{w}^{(1)} &= \vec{w}^{(0)} - \eta\cdot\frac{1}{s}\cdot\sum_{(\vec{x},y)\in S}\ell'(h(\vec{\theta},\vec{x}),y)\nabla_{\vec{w}}h(\vec{\theta},\vec{x}) \\
&= \sum_{(\vec{x},y)\in S}(2+y)\cdot\nabla_{\vec{w}}h(\vec{\theta},\vec{x}) \\
&= \sum_{(\vec{x},y)\in S_{z,M}}(2+y)\cdot\nabla_{\vec{w}}h(\vec{\theta},\vec{x}) + \sum_{(\vec{x},y)\in S\setminus S_{z,M}}(2+y)\cdot\nabla_{\vec{w}}h(\vec{\theta},\vec{x}) \\
&= \sum_{(\vec{x},y)\in S_{z,M}}(2+y)\cdot((\vec{x})_1,\dots,(\vec{x})_n,1) \ .
\end{aligned}$$

Similarly, we have:

$$\frac{\partial h(\vec{\theta},\vec{x})}{\partial\kappa} = 1 \ ,$$

and we get:

$$\kappa^{(1)} = \kappa^{(0)} - \eta\cdot\frac{1}{s}\cdot\sum_{(\vec{x},y)\in S}\ell'(h(\vec{\theta},\vec{x}),y)\cdot\frac{\partial h(\vec{\theta},\vec{x})}{\partial\kappa} = \sum_{(\vec{x},y)\in S}\ell'(h(\vec{\theta},\vec{x}),y)\cdot\frac{\partial h(\vec{\theta},\vec{x})}{\partial\kappa} = \sum_{(\vec{x},y)\in S}(2+y)\geq 2 \ .$$

Therefore, after one GD step, $w^{(1)}$ either encodes all the pairs $(\vec{x},y)$ that are matched by z and M, or remains all zeros. As we have shown previously, in cases where only one pair matches, i.e., $|S_{z,M}|=1$, we can extract the full value of the pair from the encoding $(2+y)\cdot((\vec{x})_1,\dots,(\vec{x})_n,1)$. Regardless of $M(x)$ matching or not matching z, $\kappa^{(1)}$ would be greater or equal to 2.

■

By induction on t we get:

- For $t=1$, from the initialization we get that for every $j\in[T]$: $\kappa_j^{(0)}=0$. Therefore, for every $j>1=t$ we get that:

  $$\phi\left(\kappa_{j-1}^{(t-1)},\kappa_j^{(t-1)},h(\vec{\theta}_j^{(t-1)},\vec{x})\right) = \phi\left(0,0,h(\vec{\theta}_j^{(t-1)},\vec{x})\right) = 0 \ .$$

  As such, the gradient w.r.t $w_j^{(t-1)}$ and $\kappa_j^{(t-1)}$ is zero and from the GD update rule Condition 1, described above, holds. Condition 2 holds as there are no possible j values in this range. As for the case where $j=t=1$, we get:

  $$\begin{aligned}
  \phi\left(\kappa_{j-1}^{(t-1)},\kappa_j^{(t-1)},h(\vec{\theta}_j^{(t-1)},\vec{x})\right) &= \phi\left(\kappa_0^{(0)},\kappa_1^{(0)},h(\vec{\theta}_1^{(0)},\vec{x})\right) \\
  &= \phi\left(2,0,h(\vec{\theta}_1^{(0)},\vec{x})\right) = h(\vec{\theta}_1^{(0)},\vec{x}) \ .
  \end{aligned}$$

  By applying Lemma 6.2 we get that Condition 3 holds.

- Fix some $t > 0$ and assume all the conditions holds for t. We will now show they holds for $t+1$ as well. By the assumption we have $\forall j > t \; \kappa_j^{(t)} = 0$ and $\forall j \leq t \; \kappa_j^{(t)} \geq 2$. We get that:
  - For $t+1 < j < T$ we get $\kappa_{j-1}^{(t)} = \kappa_j^{(t)} = 0$ and therefore:

    $$\phi\left(\kappa_{j-1}^{(t)}, \kappa_j^{(t)}, h(\vec{\theta}_j^{(t)}, \vec{x})\right) = 0 \; ,$$

    and the gradient w.r.t $w_j^{(t)}$ and $\kappa_j^{(t)}$ is zero so Condition 1 holds.
  - For $1 < j < t+1$ we get $\kappa_{j-1}^{(t)} = \kappa_j^{(t)} \geq 2$ and therefore:

    $$\phi\left(\kappa_{j-1}^{(t)}, \kappa_j^{(t)}, h(\vec{\theta}_j^{(t)}, \vec{x})\right) = 0 \; ,$$

    and the gradient w.r.t $w_j^{(t)}$ and $\kappa_j^{(t)}$ is zero so Condition 2 holds.
  - For $j = t+1$ we get $\kappa_{j-1}^{(t)} = \kappa_t^{(t)} \geq 2$ and $\kappa_j^{(t)} = \kappa_{t+1}^{(t)} = 0$, therefore:

    $$\phi\left(\kappa_{j-1}^{(t)}, \kappa_j^{(t)}, h(\vec{\theta}_j^{(t)}, \vec{x})\right) = h(\vec{\theta}_{t+1}^{(t)}, \vec{x}) \; .$$

    By applying Lemma 6.2 we get that Condition 3 holds.

We have shown that the clock mechanism works as described, in the sense that after running for T iterations we get that for each $t \in [T]$ the final parameters $\vec{w}_t^{(T)}$ contains an encoding of the samples that were matched in iteration t. The extraction process in this case is the same as in Figure 4, and therefore we do not repeat it. The same holds of the query complexity, which is equivalent to that of Figure 4 following the same analysis.

As for the number of parameters, here we suffer from an increased size of $\vec{\theta}$ in comparison to that of the gradient-based adversary described in Section 5, as we have to store all the parameters that will be used at once, and we cannot re-use parameters as in previous results. Here, our parameters are composed of T tuples of the form $\vec{\theta}_t = (\vec{w}, z, M, \kappa)$. As we have $\vec{w} \in \mathbb{R}^{n+1}$, $z \in \{0,1\}^k$, $M = l(n,k) = O(n+k)$, and $\kappa \in \{0, \ldots, 3|S|\} \in \{0,1\}^{\lceil \log(3|S|+1) \rceil}$ we get in total $T \cdot (n + 1 + k + l(n,k) + \lceil \log(3|S|+1) \rceil) = O\left(T \cdot (n + k + \log(|S|))\right)$ parameters. We defined $T = O(|S| \log(|S|/\epsilon))$ and $k = O(\log|S|) = O(\log|\mathcal{X}|)$, so we get $O\left(|S| \cdot \log(|S|/\epsilon) \cdot \log|\mathcal{X}|\right)$ parameters. Thus, in total we get a runtime of $T_{\mathcal{B}_{\mathsf{GD}}} = O((|S| \cdot \log(|S|/\epsilon))^2 \cdot \log|\mathcal{X}|)$.

■

7 Extensions of Our Results

In this section we provide two important extensions of our results. First, in Section 7.1 we extend our proofs to consider samples over $\mathcal{X} \times \mathcal{Y}$ where $\mathcal{Y} = \{0,1\}^m$ for some integer $m \in \mathbb{N}$ (recall that our proofs have so far considered single-bit labels, i.e., the case in which $m = 1$). Then, in Section 7.2 we extend our approach to the *stochastic mini-batch* setting, capturing common gradient-based methods in which the gradient-based information is averaged over random smaller subsets of the sample set instead of over the entire sample set.

7.1 Supporting Multi-Bit Labels

As discussed in Sections 4, 5 and 6, our proofs considered for simplicity the case of single-bit labels. That is, our proofs considered samples over $\mathcal{X} \times \mathcal{Y}$ where $\mathcal{X} = \{0,1\}^n$ for some integer $n \in \mathbb{N}$ and $\mathcal{Y} = \{0,1\}$. Here, we show how to extend our proofs to consider the case of multi-bit labels where $\mathcal{Y} = \{0,1\}^m$ for some integer $m \in \mathbb{N}$. For concreteness, in what follows we focus on extending the proof of Theorem 4.1, and note that similar extensions may be applied to the proofs of Theorems 5.1 and 6.1.

In this more general case, the structure of our gradient-based simulator remains unmodified, and we are only required to generalize its loss function ℓ, the model functions h with which it queries the gradient oracle and the manner in which it extracts the samples from the oracle's responses. In turn, this leads to increasing the running time $T_{\mathcal{B}}$ of our simulator, without increasing the number $Q_{\mathcal{B}}$ of gradient queries they issue. Specifically, the running time of the simulator provided by Theorem 4.1 increases from $T_{\mathcal{B}}(|S|, |\mathcal{X}|, |\mathcal{Y}|) = O(|S| \cdot \log^2 |\mathcal{X}|)$ to $T_{\mathcal{B}}(|S|, |\mathcal{X}|, |\mathcal{Y}|) = O(|S| \cdot \log |\mathcal{X}| \cdot (\log |\mathcal{X}| + \log |\mathcal{Y}|))$. Note that, assuming $m = \log |\mathcal{Y}| = O(\log |\mathcal{X}|) = n$, this does not introduce any asymptotic overhead. First, as for the loss function ℓ, recall that for single-bit labels we defined $\ell(\hat{y}, y) = (2+y) \cdot \hat{y}$. In this case, the partial derivative of ℓ with respect to $\hat{y}$, denoted for clarity by $\ell'(\hat{y}, y)$, satisfies

$$\ell'(\hat{y}, y) = (2+y) \in \mathbb{R} \ .$$

Denote by $\vec{2} = (2, \ldots, 2) \in \mathbb{R}^m$. More generally, for m-bit labels, we let

$$\ell(\vec{\hat{y}}, \vec{y}) = \left(\vec{2} + \vec{y}\right) \cdot \vec{\hat{y}} = \left((2 + (\vec{y})_1) \cdot (\vec{\hat{y}})_1, \ldots, (2 + (\vec{y})_m) \cdot (\vec{\hat{y}})_m\right) \in \mathbb{R}^m \ ,$$

and it thus holds that

$$\begin{aligned} \ell'(\vec{\hat{y}}, \vec{y}) &= \left(\frac{\partial \ell(\vec{\hat{y}}, \vec{y})}{\partial (\vec{\hat{y}})_1}, \ldots, \frac{\partial \ell(\vec{\hat{y}}, \vec{y})}{\partial (\vec{\hat{y}})_m}\right) \\ &= ((2 + (\vec{y})_1), \ldots, (2 + (\vec{y})_m)) \\ &= \left(\vec{2} + \vec{y}\right) \in \mathbb{R}^m \end{aligned}$$

Second, as for the model functions h with which we query the gradient oracle as part of the procedure CHECK-MATCH(z), recall that for single-bit queries their gradient with respect to θ was defined to satisfy

$$\nabla_\theta h(\vec{\theta}, \vec{x}) = \begin{cases} (0, \ldots, 0) \in \mathbb{R}^{n+1} & \text{if } \vec{x}_{1\ldots k} \neq z \\ ((\vec{x})_1, \ldots, (\vec{x})_n, 1) \in \mathbb{R}^{n+1} & \text{otherwise} \end{cases}$$

More generally, for m-bit labels, we let their gradient[5] with respect to θ satisfy

$$\nabla_\theta h(\vec{\theta}, \vec{x}) = \begin{cases} \begin{pmatrix} 0 & \ldots & 0 \\ & \ddots & \\ 0 & \ldots & 0 \end{pmatrix} \in \mathbb{R}^{m \times (n+m)} & \text{if } \vec{x}_{1\ldots k} \neq z \\ \begin{pmatrix} (\vec{x})_1 & \ldots & (\vec{x})_n & 1 & 0 & \ldots & 0 \\ (\vec{x})_1 & \ldots & (\vec{x})_n & 0 & 1 & \ldots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ (\vec{x})_1 & \ldots & (\vec{x})_n & 0 & 0 & \ldots & 1 \end{pmatrix} \in \mathbb{R}^{m \times (n+m)} & \text{otherwise} \end{cases}$$

[5] This is in fact denoted the Jacobian of h with respect to θ. The term Gradient corresponds to the vector of partial derivatives of scalar-valued functions, while the Jacobian corresponds to vector-valued functions.

Denote by $(\vec{v}||\vec{u})$ the concatenation of two vectors $\vec{v}$ and $\vec{u}$, and for every $j \in [m]$ denote by $\vec{e}_j \in \{0,1\}^m$ the vector with 1 in its jth entry and 0's elsewhere. Thus, we can describe our desired gradient behavior as

$$\nabla_\theta h(\vec{\theta}, \vec{x}) = \begin{pmatrix} \vec{x}||\vec{e}_1 \\ \vec{x}||\vec{e}_2 \\ \vdots \\ \vec{x}||\vec{e}_m \end{pmatrix} \cdot \mathbb{1}_{\vec{x}_{1\ldots k}=z} \in \mathbb{R}^{m \times (n+m)} .$$

Equipped with the above loss function ℓ and model h for each iteration, our simulator obtains a response $\vec{g}$ from the gradient oracle, where

$$\begin{aligned} \vec{g} &= \frac{1}{|S|} \sum_{(\vec{x},\vec{y}) \in S} \nabla_\theta \ell(h_{\vec{\theta}}(\vec{x}), \vec{y}) \\ &= \frac{1}{|S|} \sum_{(\vec{x},\vec{y}) \in S} \ell'(h_{\vec{\theta}}(\vec{x}), \vec{y}) \cdot \nabla_\theta h_{\vec{\theta}}(\vec{x}) \\ &= \frac{1}{|S|} \sum_{(\vec{x},\vec{y}) \in S} \left(\vec{2} + \vec{y}\right) \cdot \nabla_\theta h_{\vec{\theta}}(\vec{x}) . \end{aligned}$$

Denoting by $S_z \subseteq S$ the set of samples $(\vec{x}, \vec{y}) \in S$ for which $\vec{x}$ is prefixed by z, we obtain

$$\begin{aligned} \vec{g} &= \frac{1}{|S|} \sum_{(\vec{x},\vec{y}) \in S_z} \ell'(h(\vec{\theta}, \vec{x}), \vec{y}) \cdot \nabla_\theta h(\vec{\theta}, \vec{x}) \\ &= \frac{1}{|S|} \sum_{(\vec{x},\vec{y}) \in S_z} \left(\vec{2} + \vec{y}\right) \cdot \begin{pmatrix} \vec{x}||\vec{e}_1 \\ \vec{x}||\vec{e}_2 \\ \vdots \\ \vec{x}||\vec{e}_m \end{pmatrix} \\ &= \frac{1}{|S|} \cdot \sum_{(\vec{x},\vec{y}) \in S_z} \left(\underbrace{(\vec{x})_1 \cdot \sum_{j=1}^{m} (2 + (\vec{y})_j), \ldots, (\vec{x})_n \cdot \sum_{j=1}^{m} (2 + (\vec{y})_j),}_{(\vec{2}+\vec{y}) \cdot (\vec{x}, \ldots, \vec{x})^\top} \right. \\ &\qquad\qquad\qquad\qquad \left. \underbrace{(2 + (\vec{y})_1), \ldots, (2 + (\vec{y})_m)}_{(\vec{2}+\vec{y}) \cdot (\vec{e}_1, \ldots, \vec{e}_m)^\top} \right) \in \mathbb{R}^{n+m} \end{aligned}$$

Recall that, in the proof of Theorem 4.1, a sample $(\vec{x}, y)$ was extracted only when the condition $0 < |S| \cdot \vec{g}_{n+1} < 4$ was satisfied in Step 6 of the procedure CHECK-MATCH. We now show that this holds also in the more general case. First, we observe that

$$(\vec{g})_{n+1} = \frac{1}{|S|} \cdot \sum_{(\vec{x},\vec{y}) \in S_z} (2 + (\vec{y})_1) ,$$

and we distinguish between the following three cases depending on the size of the set S_z:

- **Case I: $|S_z| = 0$.** In this case we get that $\vec{g} = (1/|S|) \cdot (0, \ldots, 0) \in \mathbb{R}^{n+m}$, and in particular $|S| \cdot \vec{g}_{n+1} = 0$. Therefore, the condition described above does not hold and no extraction will be performed.
- **Case II: $|S_z| = 1$.** In this case, for $S_z = \{(\vec{x}', \vec{y}')\}$ we get:
$$|S| \cdot \vec{g}_{n+1} = \sum_{(\vec{x},\vec{y}) \in S_z} (2 + (\vec{y})_1) = 2 + (\vec{y'})_1 \ .$$
As $\vec{y'} \in \{0,1\}^m$ we get that $(\vec{y'})_1 < 2$, and in particular:
$0 < 2 + (\vec{y'})_1 < 2 + 2 = 4$. Therefore, the condition holds and the pair $\{(\vec{x}', y')\}$ can be extracted as we describe below.
- **Case III: $|S_z| \geq 2$.** Without loss of generality, we will analyze this case for $|S_z| = 2$. Let $S_z = \{(\vec{x}_1, \vec{y}_1), (\vec{x}_2, \vec{y}_2)\}$. We have:
$$\begin{aligned} |S| \cdot \vec{g}_{n+1} &= \sum_{(\vec{x},\vec{y}) \in S_z} (2 + (\vec{y})_1) \\ &= (2 + (\vec{y}_1)_1) + (2 + (\vec{y}_2)_1) \\ &= 4 + (\vec{y}_1)_1 + (\vec{y}_2)_1 \ . \end{aligned}$$
Therefore we get that $|S| \cdot \vec{g}_{n+1} \geq 4$, and as such, the condition above does not hold, and no sample will be extracted.

The extraction process for the above Case II slightly differs from that of Section 4. Following the notation above, let $\{(\vec{x'}, \vec{y'})\}$ be the sample to be extracted, and we extract it as follows:

- **Extraction of $\vec{x'}$.** Here we first compute the following term C:
$$C = |S| \cdot \sum_{j=n+1}^{n+m} (\vec{g})_j = |S| \cdot \sum_{j=n+1}^{n+m} \frac{1}{|S|} \left(2 + (\vec{y'})_{j-n}\right) = \sum_{j'=1}^{m} (2 + (\vec{y'})_{j'}) \ ,$$
and then compute
$$\vec{x''} = \frac{|S|}{C} \cdot (\vec{g})_{1 \ldots n} \ .$$
Then, for every $i \in [n]$ it holds that
$$\begin{aligned} (\vec{x''})_i &= \frac{|S|}{C} \cdot (\vec{g})_i \\ &= \frac{|S|}{C} \cdot \left(\frac{1}{|S|} \cdot (\vec{x'})_i \cdot \sum_{j=1}^{m} \left(2 + (\vec{y'})_j\right) \right) \\ &= \frac{|S|}{C} \cdot \left(\frac{1}{|S|} \cdot (\vec{x'})_i \cdot C \right) \\ &= (\vec{x'})_i \ . \end{aligned}$$
- **Extraction of $\vec{y'}$.** The extraction is computed as follows:
$$\vec{y''} = (|S| \cdot (\vec{g})_{n+1 \ldots n+m}) - \vec{2} \ .$$
Then, for every $j \in [m]$ it holds that
$$(\vec{y''})_j = \left(|S| \cdot \left(\frac{1}{|S|} \cdot (2 + (\vec{y'})_j) \right) \right) - 2 = (2 + (\vec{y'})_j) - 2 = (\vec{y'})_j$$

7.2 The Stochastic Mini-Batch Setting

Throughout this work we have defined the gradient oracle $\mathcal{O}_{\mathsf{G}}$ such that, given a query $(S, \ell, h, \vec{\theta})$, it averages the gradient of the loss function ℓ composed with the model function $h(\vec{\theta}, \cdot)$ over the *entire* sample set. This setting is referred to in the ML community as the *full-batch* setting.

In a wide variety of ML use cases, it is common for the learner to optimize the parameters $\vec{\theta}$ of h over many iterations using a very large set S. It is, therefore, computationally expensive to compute the gradient over the entire set S at each time step. Thus, in order to avoid such a significant cost, it is more common to operate in a *stochastic mini-batch* manner. In this setting, given a query $(S, \ell, h, \vec{\theta})$, the oracle $\mathcal{O}_{\mathsf{G}}$ first uniformly samples a batch $B \subset S$, where $|B| < |S|$, and computes the averaged gradient only over this subset. This variant is known in the ML community as stochastic gradient descent (SGD). One particularly interesting sub-case of this setting is the *single-sample* setting, in which the batch is composed of a single sample (i.e.,$|B| = 1$).

In this section we show that our framework and results are not limited to the full-batch setting, and both capture and apply to the mini-batch setting as well. We show this by extending our framework to consider an appropriately-modified *mini-batch* gradient oracle, and then by adapting our black-box simulator from Section 5 accordingly.

The mini-batch gradient-oracle. We say that a function $b : \mathbb{N} \to \mathbb{N}$ is a *mini-batch* function if for any $s \geq 1$ it holds that $1 \leq b(s) \leq s$. For any such function $b(\cdot)$, we define the *b-mini-batch gradient oracle* $\mathcal{O}_{\mathsf{G}[b]}$ as follows: On input $(S, \ell, h, \vec{\theta})$, the oracle samples a subset (a "mini-batch") $B \subseteq S$ of size $b(|S|)$ uniformly among all such subsets, and then computes the following gradient vector with respect to this batch (instead of with respect to the entire sample set S as in Section 3):

$$\vec{g} = \frac{1}{|B|} \sum_{(\vec{x},y) \in B} \nabla_\theta \ell(h(\vec{\theta}, \vec{x}), y) \ .$$

This enables us to accordingly extend our notion of black-box simulation (recall Definition 3.4):

Definition 7.1 (Black-box ϵ-simulation – Mini-batch setting)**.** Let $\mathcal{X}$ and $\mathcal{Y}$ be finite sets, let $\epsilon > 0$, and let $b : \mathbb{N} \to \mathbb{N}$ be a mini-batch function. We say that a gradient-based algorithm $\mathcal{B}$ *black-box ϵ-simulates all sample-based adversaries in the b-batch setting* with respect to $\mathcal{X} \times \mathcal{Y}$ if for any sample-based adversary $\mathcal{A}$, integer $s \geq 1$, and distribution $\mathcal{D}$ over $(\mathcal{X} \times \mathcal{Y})^s$ it holds that

$$\mathrm{SD}\left((S, \mathcal{A}(S)), (S, \mathcal{B}^{\mathcal{O}_{\mathsf{G}[b]}(S,\cdot,\cdot,\cdot), \mathcal{A}(\cdot)}(1^s)\right) \leq \epsilon,$$

where $S \leftarrow \mathcal{D}$ in both distributions.

Equipped with our adapted notions, we prove the following theorem that extents Theorem 5.1 to the mini-batch setting. Note that for the case in which $b(s) = s$ (i.e., the case in which the b-batch gradient oracle is in fact the standard full-batch one) the statement of Theorem 7.2 coincides with that of Theorem 5.1 as expected.

Theorem 7.2. *Let $\mathcal{X} = \{0,1\}^n$ for some $n \geq 1$, and let $\mathcal{Y} = \{0,1\}$. For any $\epsilon > 0$ and batch function $b : \mathbb{N} \to \mathbb{N}$, there exists a gradient-based adversary $\mathcal{B}$ that black-box ϵ-simulates all sample-based adversaries in the b-batch setting with respect to $\mathcal{X} \times \mathcal{Y}$, where:*

- *$\mathcal{B}$ runs in time $T_{\mathcal{B}}(|S|, |\mathcal{X}|, |\mathcal{Y}|) = O((|S|^2/b) \cdot \log(|S|/\epsilon) \cdot \log|\mathcal{X}|)$.*
- *$\mathcal{B}$ issues $Q_{\mathcal{B}}(|S|, |\mathcal{X}|, |\mathcal{Y}|) = O((|S|^2/b) \cdot \log(|S|/\epsilon))$ parallel queries to the b-batch gradient oracle, followed by a single query to the simulated sample-based adversary.*

Proof of Theorem 7.2. The proof is nearly identical to that of Theorem 5.1 with the exception that the number of iterations $T = \lceil 16s \cdot \ln(s/\epsilon) \rceil$ in the description of the gradient-based simulator described in Figure 4 is replaced with $T = \lceil 16 \cdot (s^2/b) \cdot \ln(s/\epsilon) \rceil$, where $b = b(s)$.

Equipped with these parameters, for every iteration $t \in [T]$ we denote by B_t the set of sample indices chosen by the gradient oracle for responding to t-th iteration's query. As in the proof of Theorem 5.1, we also denote by $z_t \in \{0,1\}^k$ and $M_t \in \mathcal{M}_{n,k}$ the random variables corresponding to the bit-string and the function sampled at iteration t, respectively. Letting $S = \{(x_i, y_i)\}_{i \in [s]}$, then for every $i \in [s]$ and for each iteration $t \in [T]$ we now say that the sample (x_i, y_i) is *isolated* by the t-th iteration if $(x_i, y_i) \in B_t$, $M_t(x_i) = z_t$, and for every $j \in [s] \setminus \{i\}$ it holds that $M_t(x_i) \neq M_t(x_j)$. For every $i \in [s]$ we let $\mathsf{Failure}_i$ denote the event in which the sample (x_i, y_i) is not isolated by any iteration, and we let $\mathsf{Failure} = \bigcup_{i \in [s]} \mathsf{Failure}_i$.

As in the proof of Theorem 5.1, if the event $\mathsf{Failure}$ does not occur, then the gradient-based simulator extracts the entire set S (and otherwise outputs $\perp$). Thus, we are only left with showing that the event $\mathsf{Failure}$ occurs with probability at most ϵ. From the definition of the events $\{\mathsf{Failure}_i\}_{i \in [s]}$, for every $i \in [s]$ it holds that

$$\begin{aligned}\Pr[\mathsf{Failure}_i] &= \left(\Pr[(x_i, y_i) \text{ is not isolated by the first iteration}]\right)^T \\ &= \left(1 - \Pr_{B_1, z_1, M_1}\left[\begin{array}{c} i \in B_1 \wedge z_1 = M(x_i) \\ \wedge \nexists j \in [s] \setminus \{i\} : M_1(x_i) = M_1(x_j) \end{array}\right]\right)^T ,\end{aligned}$$

where the above equality follows from the independence across all iterations. Using the union bound, it holds that

$$\begin{aligned}&\Pr_{M_1}[\, \nexists j \in [s] \setminus \{i\} : M_1(x_i) = M_1(x_j) \,] \\ &\qquad = 1 - \Pr_{M_1}[\, \exists j \in [s] \setminus \{i\} : M_1(x_i) = M_1(x_j) \,] \\ &\qquad \geq 1 - \sum_{j \in [s] \setminus \{i\}} \Pr_{M_1}[\, M_1(x_i) = M_1(x_j) \,] \\ &\qquad = 1 - \sum_{j \in [s] \setminus \{i\}} \frac{1}{2^k} \\ &\qquad = 1 - (s-1) \cdot \frac{1}{2^k} ,\end{aligned}$$

where the last equality follows from the pairwise independence of the function family $\mathcal{M}_{n,k}$. Since z_1 is uniformly sampled and is independent of M_1, the probability of a match between z_1 and $M_1(x_i)$, given that there are no collisions can be computed as

$$\Pr_{z_1, M_1}[\, z_1 = M_1(x_i) \mid \nexists j \in [s] \setminus \{i\} : M_1(x_i) = M_1(x_j) \,] = \frac{1}{2^k} .$$

In addition, since B_1 is sampled independently of M_1 and z_1, then the probability of the event $i \in B_1$ is b/s even when conditioning on a match between z_1 and $M_1(x_i)$ and on having no collisions. Therefore, Therefore,

$$\begin{aligned}\Pr[\mathsf{Failure}_i] &\leq \left(1 - \left(1 - \frac{s-1}{2^k}\right) \cdot \frac{1}{2^k} \cdot \frac{b}{s}\right)^T \\ &= \left(1 - \left(\frac{1}{2^k} - \frac{b-1}{2^{2k}}\right) \cdot \frac{b}{s}\right)^T ,\end{aligned}$$

and our choice of $k = \lceil \log_2(2s) \rceil$ guarantees that

$$\frac{1}{2^k} - \frac{s-1}{2^{2k}} \geq \frac{s}{2^{2k}} \geq \frac{1}{16s} ,$$

implying

$$\Pr\left[\mathsf{Failure}_i\right] \leq \left(1 - \frac{b}{16s^2}\right)^T \leq e^{-T \cdot b/16s^2} .$$

We conclude by observing that our choice of $T = \lceil 16 \cdot (s^2/b) \cdot \ln(s/\epsilon) \rceil$ now implies

$$\Pr\left[\,\mathsf{Failure}\,\right] = \Pr\left[\bigcup_{i \in [s]} \mathsf{Failure}_i\right] \leq s \cdot e^{-T \cdot b/16s^2} \leq \epsilon .$$

■

References

[ABW10] B. Applebaum, B. Barak, and A. Wigderson. Public-key cryptography from different assumptions. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 171–180, 2010.

[ADH+19] S. Arora, S. Du, W. Hu, Z. Li, and R. Wang. Fine-grained analysis of optimization and generalization for overparameterized two-layer neural networks. In *International Conference on Machine Learning*, pages 322–332. PMLR, 2019.

[AKM+21] E. Abbe, P. Kamath, E. Malach, C. Sandon, and N. Srebro. On the power of differentiable learning versus PAC and SQ learning. *Advances in Neural Information Processing Systems*, 34:24340–24351, 2021.

[AS20] E. Abbe and C. Sandon. Poly-time universality and limitations of deep learning. *arXiv preprint arXiv:2001.02992*, 2020.

[BB22] A. Baksi and A. Baksi. Machine learning-assisted differential distinguishers for lightweight ciphers. *Classical and Physical Security of Symmetric Key Cryptographic Algorithms*, pages 141–162, 2022.

[BCC+10] D. J. Bernstein, H. Chen, C. Cheng, T. Lange, R. Niederhagen, P. Schwabe, and B. Yang. ECC2K-130 on NVIDIA GPUs. In *Progress in Cryptology – INDOCRYPT '10*, pages 328–346, 2010.

[Ber07] D. J. Bernstein. Better price-performance ratios for generalized birthday attacks. Workshop Record of SHARCS'07: Special-purpose Hardware for Attacking Cryptographic Systems, 2007.

[BGL+21] Z. Bao, J. Guo, M. Liu, L. Ma, and Y. Tu. Conditional differential-neural cryptanalysis. *IACR Cryptol. ePrint Arch.*, 2021:719, 2021.

[BGP+21] A. Benamira, D. Gerault, T. Peyrin, and Q. Q. Tan. A deeper look at machine learning-based cryptanalysis. In *Advances in Cryptology – EUROCRYPT '21*, pages 805–835. Springer, 2021.

[BHY$^+$23] G. Buzaglo, N. Haim, G. Yehudai, G. Vardi, Y. Oz, Y. Nikankin, and M. Irani. Deconstructing data reconstruction: Multiclass, weight decay and general losses. *arXiv preprint arXiv:2307.01827*, 2023.

[BKK$^+$12] J. W. Bos, M. E. Kaihara, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. Solving a 112-bit prime elliptic curve discrete logarithm problem on game consoles using sloppy reduction. *Int. J. Appl. Cryptogr.*, 2(3):212–228, 2012.

[BKN$^+$10] J. W. Bos, T. Kleinjung, R. Niederhagen, and P. Schwabe. ECC2K-130 on Cell CPUs. In *Progress in Cryptology – AFRICACRYPT '10*, pages 225–242, 2010.

[BLN$^+$09] D. J. Bernstein, T. Lange, R. Niederhagen, C. Peters, and P. Schwabe. FSBday: Implementing Wagner's generalized birthday attack against the SHA-3 round-1 candidate FSB. In *Progress in Cryptology – INDOCRYPT '09*, pages 18–38, 2009.

[BLY$^+$23] Z. Bao, J. Lu, Y. Yao, and L. Zhang. More insight on deep learning-aided cryptanalysis. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 436–467. Springer, 2023.

[Bos15] J. W. Bos. Parallel cryptanalysis. Summer school on real-world crypto and privacy, Croatia, 2015. Slides available at https://summerschool-croatia.cs.ru.nl/2015/ParallelCryptanalysis.pdf.

[BSS$^+$13] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The simon and speck families of lightweight block ciphers. Cryptology ePrint Archive, Paper 2013/404, 2013. https://eprint.iacr.org/2013/404.

[CLE$^+$19] N. Carlini, C. Liu, Ú. Erlingsson, J. Kos, and D. Song. The secret sharer: Evaluating and testing unintended memorization in neural networks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 267–284, 2019.

[CN11] Y. Chen and P. Q. Nguyen. Bkz 2.0: Better lattice security estimates. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 1–20. Springer, 2011.

[Coo23] S. A. Cook. The complexity of theorem-proving procedures. In *Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook*, pages 143–152. 2023.

[CSY$^+$23] Y. Chen, Y. Shen, H. Yu, and S. Yuan. A new neural distinguisher considering features derived from multiple ciphertext pairs. *The Computer Journal*, 66(6):1419–1433, 2023.

[Dan20] A. Daniely. Neural networks learning and memorization with (almost) no over-parameterization. *Advances in Neural Information Processing Systems*, 33:9007–9016, 2020.

[Din14] I. Dinur. Improved differential cryptanalysis of round-reduced speck. In *Selected Areas in Cryptography–SAC 2014: 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers 21*, pages 147–164. Springer, 2014.

[DLL$^+$19] S. Du, J. Lee, H. Li, L. Wang, and X. Zhai. Gradient descent finds global minima of deep neural networks. In *International conference on machine learning*, pages 1675–1685. PMLR, 2019.

[DPS23] L. Ducas, E. Postlethwaite, and J. Sotáková. Salsa Verde versus the actual state of the art. CRYPTO '23 Rump Session Talk. Available at https://crypto.iacr.org/2023/rump/crypto2023rump-paper13.pdf, 2023.

[GLN22] A. Gohr, G. Leander, and P. Neumann. An assessment of differential-neural distinguishers. Cryptology ePrint Archive, Paper 2022/1521, 2022. https://eprint.iacr.org/2022/1521.

[Goh19] A. Gohr. Improving attacks on round-reduced SPECK32/64 using deep learning. In *Advances in Cryptology – CRYPTO '19*, pages 150–179. Springer, 2019.

[HRC21a] Z. Hou, J. Ren, and S. Chen. Cryptanalysis of round-reduced simon32 based on deep learning. Cryptology ePrint Archive, Paper 2021/362, 2021. https://eprint.iacr.org/2021/362.

[HRC21b] Z. Hou, J. Ren, and S. Chen. Improve neural distinguisher for cryptanalysis. Cryptology ePrint Archive, Paper 2021/1017, 2021. https://eprint.iacr.org/2021/1017.

[HVY$^+$22] N. Haim, G. Vardi, G. Yehudai, O. Shamir, and M. Irani. Reconstructing training data from trained neural networks. *Advances in Neural Information Processing Systems*, 35:22911–22924, 2022.

[HZR$^+$16] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[JGH18] A. Jacot, F. Gabriel, and C. Hongler. Neural tangent kernel: Convergence and generalization in neural networks. *Advances in neural information processing systems*, 31, 2018.

[JKM20] A. Jain, V. Kohli, and G. Mishra. Deep learning based differential distinguisher for lightweight cipher present. Cryptology ePrint Archive, Paper 2020/846, 2020. https://eprint.iacr.org/2020/846.

[KB14] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[Kea98] M. Kearns. Efficient noise-tolerant learning from statistical queries. *Journal of the ACM (JACM)*, 45(6):983–1006, 1998.

[Kha93] M. Kharitonov. Cryptographic hardness of distribution-specific learning. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 372–381, 1993.

[KS09] A. R. Klivans and A. A. Sherstov. Cryptographic hardness for learning intersections of halfspaces. *Journal of Computer and System Sciences*, 75(1):2–12, 2009.

[KV89] M. J. Kearns and L. G. Valiant. Cryptographic limitations on learning boolean formulae and finite automata. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 433–444, 1989.

[Lev73] L. A. Levin. Universal sequential search problems. *Problemy peredachi informatsii*, 9(3):115–116, 1973.

[LLS$^+$22] J. Lu, G. Liu, B. Sun, C. Li, and L. Liu. Improved (related-key) differential-based neural distinguishers for simon and simeck block ciphers. *arXiv preprint arXiv:2201.03767*, 2022.

[LSW$^+$23a] C. Li, J. Sotakova, E. Wenger, Z. Allen-Zhu, F. Charton, and K. Lauter. Salsa verde: a machine learning attack on learning with errors with sparse small secrets. *arXiv preprint arXiv:2306.11641*, 2023.

[LSW$^+$23b] C. Li, J. Sotáková, E. Wenger, M. Malhou, E. Garcelon, F. Charton, and K. Lauter. Salsa picante: a machine learning attack on lwe with binary secrets. *arXiv preprint arXiv:2303.04178*, 2023.

[Mic] D. Micciancio. Parallel algorithms for lattice problems. https://cseweb.ucsd.edu/~daniele/LatticeLinks/Parallel.html.

[Nie12] R. Niederhagen. Parallel Cryptanalysis. PhD thesis, Eindhoven University of Technology, 2012. http://polycephaly.org/thesis/niederhagen-thesis-printed.pdf.

[Pol78] J. M. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, 32:918–924, 1978.

[Reg05] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, pages 84–93, 2005.

[Riv91] R. L. Rivest. Cryptography and machine learning. In *Advances in Cryptology – ASIACRYPT '91*, pages 427–439, 1991.

[RM51] H. Robbins and S. Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[Rud16] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[SRS17] C. Song, T. Ristenpart, and V. Shmatikov. Machine learning models that remember too much. In *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*, pages 587–601, 2017.

[SSBD14] S. Shalev-Shwartz and S. Ben-David. Understanding machine learning: From theory to algorithms. Cambridge university press, 2014.

[SWL$^+$24] S. Stevens, E. Wenger, C. Y. Li, N. Nolte, E. Saxena, F. Charton, and K. Lauter. SALSA FRESCA: Angular embeddings and pre-training for ML attacks on learning with errors. Cryptology ePrint Archive, Paper 2024/150, 2024.

[SZB21] M. J. Song, I. Zadik, and J. Bruna. On the cryptographic hardness of learning single periodic neurons. *Advances in neural information processing systems*, 34:29602–29615, 2021.

[SZM21] H.-C. Su, X.-Y. Zhu, and D. Ming. Polytopic attack on round-reduced simon32/64 using deep learning. In *Information Security and Cryptology: 16th International Conference, Inscrypt 2020, Guangzhou, China, December 11–14, 2020, Revised Selected Papers*, pages 3–20. Springer, 2021.

[Val84] L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[vOW99] P. C. van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.

[VSP+17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[Wag02] D. A. Wagner. A generalized birthday problem. In *Advances in Cryptology – CRYPTO '02*, pages 288–303, 2002.

[WCC+22] E. Wenger, M. Chen, F. Charton, and K. E. Lauter. Salsa: Attacking lattice cryptography with transformers. *Advances in Neural Information Processing Systems*, 35:34981–34994, 2022.

[YK21] T. Yadav and M. Kumar. Differential-ml distinguisher: Machine learning based generic extension for differential cryptanalysis. In *International Conference on Cryptology and Information Security in Latin America*, pages 191–212. Springer, 2021.

[ZBH+17] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning requires rethinking generalization. In *International Conference on Learning Representations*, 2017.

[ZW22] L. Zhang and Z. Wang. Improving differential-neural distinguisher model for des, chaskey, and present. *arXiv preprint arXiv:2204.06341*, 2022.

[ZWw22] L. Zhang, Z. Wang, and B. wang. Improving differential-neural cryptanalysis. Cryptology ePrint Archive, Paper 2022/183, 2022. https://eprint.iacr.org/2022/183.

A Positioning Previous Work Within Our Framework

As we have discussed in Section 1.2, there has been a significant interest in ML-based cryptanalysis, especially in the context of block ciphers and lattice-based cryptography. Most notably, this includes the seminal work of Gohr [Goh19], that was the first to introduce differential-neural distinguishers, and the work of Wenger, Chen, Charton, and Lauter [WCC+22], that trained an ML model for attacking the LWE problem. In order to have a better intuition of the generality of our framework, proposed and formalized in Section 3, we will now show how these two notable works can fit into our framework. More specifically, we will explain how the attackers proposed in each work can fit into our definition of the GD-based adversary, described in Figure Algorithm 2. For this, we will define what does $\mathcal{X}$ and $\mathcal{Y}$ represent, and correspondingly the set S. In addition, we will define the adversary's parameters T, ℓ, h, η, the initialization of the parameters $\vec{\theta}_0$, as well as the computation performed in the PostProcess step.

We note that while in our work we focused on the full-batch settings, these two works utilize the more commonly used mini-batch setting. In the mini-batch setting, it is common to treat iterations in terms of *epochs*, where each epoch corresponds to the number of iterations required to complete one pass over the entire sample set. For aligning with our notation, we will describe by T, the number of GD iterations, the total number of iterations, i.e., the number of epochs times the number of iterations in each epoch. The latter depends on the size of the set and the batch size. Put differently, $T = E \times \lceil |S|/b \rceil$, for E epochs and a batch size of b. In addition, we focused on the GD optimization algorithm, while Gohr and Wenger et al. used the ADAM optimization algorithm, which is a variant of the GD algorithm. For simplicity we will replace ADAM with GD in this discussion.

Differential-neural distinguishers. In the case of the differential-neural distinguishers, suggested by Gohr [Goh19], a neural network was trained to distinguish between *real* ciphertext pairs, i.e., those that correspond to plaintext pairs with specific input difference, and *random* ciphertext pairs, i.e., those that correspond to plaintext pairs with some other input difference. This work focused on a round reduced version of Speck32/64 [BSS+13] - a variant of the iterative block cipher Speck, corresponding to 32-bit block size and 64-bit keys. Speck32/64 consists of 22 rounds, however Gohr analyze a round-reduced variant of up to 8 rounds.

For framing the work of Gohr in the context of our framework we define the finite set $\mathcal{X} \in \{0,1\}^{32} \times \{0,1\}^{32}$ to represent ciphertext pairs (C_0, C_1). More specifically, for a uniformly sampled key $K \in \{0,1\}^{64}$ and a uniformly sampled plaintext pair $(P_0, P_1) \in \{0,1\}^{32} \times \{0,1\}^{32}$, the ciphertext pair (C_0, C_1) is obtained by encrypting each plaintext for 8 rounds using the key K. The finite set $\mathcal{Y} \in \{0,1\}$ represents the *label* y of each ciphertext pair, indicating whether the ciphertext pair corresponds to a plaintext pair with a predefined input difference $\Delta_{in} = 0x0040/0000$ or not. More specifically, for a ciphertext pair $(C_0, C_1) \in \mathcal{X}$ we get:

$$y = \begin{cases} 1 & \text{if } P_0 \oplus P_1 = \Delta_{in} \\ 0 & \text{otherwise} \end{cases},$$

where (P_0, P_1) is the corresponding plaintext pair. The set $S \subseteq (\mathcal{X} \times \mathcal{Y})^s$ corresponds to a set of s ciphertext pairs and their corresponding label.

Following the procedure in Gohr's work, we define h to represent a ResNet-based neural network [HZR+16]. We omit the exact details of the network's architecture and refer the reader to [Goh19]. The model was trained for $T = 360,000$ iterations (corresponding to 200 epochs with a batch size of $5,000$), using the mean square error (MSE) as the loss function ℓ, including some additional regularization terms. As for the learning rate η, a cyclic scheduling was used, defining a different learning rate value at each iteration, we omit this discussion as well for clarity. The parameters $\vec{\theta}_0$ are randomly initialized.

Although Gohr proposed an additional partial key recovery attack based on the trained model, it was not the core of the work, and therefore we will treat the *accuracy* of the trained model as the output of the procedure, i.e., the prediction success rate over unseen ciphertext pairs. Therefore, starting from $\vec{\theta}_0$, the model is trained for T iterations. Then, in the $\mathsf{PostProcess}$ step, the adversary evaluates the performance of $h(\vec{\theta}_T, \cdot)$ over a subset of samples $S_{test} \subset S$ that were separated from S before training. In such case, the model would have trained on $S_{train} = S \setminus S_{test}$. The output v would be the accuracy computed with respect to the set S_{test}:

$$v = \frac{1}{|S_{test}|} \sum_{((C_0,C_1),y) \in S_{test}} \mathbb{1}_{h(\vec{\theta}_T,(C_0,C_1))=y}$$

We note that the notion of treating the accuracy as the output of the adversary aligns with our simulation notation, as in this case it is natural to compare the accuracy achieved by a GD-based adversary to that of a sample-based one.

We also note that Gohr additionally proposed to enhance the capabilities of neural distinguishers by following a hybrid approach and allowing the adversary to interact with both a sample-based oracle and a gradient-based oracle. This approach is not directly captured by our framework. However, we do allow a computationally light pre-processing step with direct access to samples, which we used for serializing samples, as well as a post-processing step that does not have access to samples. Allowing the post-processing step to also access samples and allowing more significant preprocessing will indeed capture hybrid attackers as well.

Learning with errors. In this section we will show how the method proposed by Wenger et al. [WCC+22] can be framed in the context of our GD-based adversary. Although there has been three follow-up works of the same research group [LSW+23b, LSW+23a, SWL+24], for clarity we focus on the method proposed in the first work.

This line of work focus on the Learning With Errors (LWE) problem, where given a dimension n, integer modulus q, and a secret vector $s \in \mathbb{Z}_q^n$, the LWE problem is to find the secret s given multiple noisy inner products of form $b = a \cdot s + e \mod q$, where $a \in \mathbb{Z}_q^n$ is a random vector, and e is a small error value (i.e., noise), sampled from a narrow centered Gaussian distribution.

The method proposed by Wenger et al. trains a model to predict the value of b given a. Then, the secret recovery is perform by querying the trained model using a specific form of queries, i.e., pairs of the form (a, b).

Thus, we defined the finite set $\mathcal{X}$ to represent the values of a, i.e., $\mathcal{X} \in \{0,1\}^{n \times \log q}$. The authors focus on binary secrets, and therefore given a predefined secret $s \in \{0,1\}^{n \times \log q}$, we define the finite set $\mathcal{Y}$ to represent the values of $b = a \cdot s + e \mod q$, i.e., $\mathcal{Y} \in \{0,1\}^{\log q}$. We note that the authors experimented with different bases for encoding the integer values. Here, for simplicity, we assume they are encoded with base 2, i.e., binary representation. The set $S \subseteq (\mathcal{X} \times \mathcal{Y})^s$ corresponds to a set of s pairs of the form $(a, b = a \cdot s + e \mod q)$.

The function h is defined to be a sequence-to-sequence transformer based neural network [VSP+17], the parameters $\vec{\theta}_0$ are randomly initialized, the loss function ℓ is defined to be the cross-entropy loss function, the learning rate η is set to 5×10^{-5}, and the number of iterations is roughly $T = 585,000$. This is an estimation as the model was trained on mini-batches of size 128, and each epoch consists of $300,000$ samples, therefore leading to around 2343 iterations per epoch. The model is trained for around 250 epochs, but can terminate earlier if a high enough accuracy is obtained.

After training is concluded, Wenger et al. performs a key recovery attack using queries to the trained model $h(\vec{\theta}_T, \cdot)$. We will not describe the actual algorithm, and refer the reader to [WCC+22]. Therefore, we define the PostProcess step to be the evaluation of this key recovery algorithm, and thus the output value v will represent the estimate of the key outputted by this procedure.

We note that follow-up works [LSW+23b, LSW+23a, SWL+24] improve upon the results of Wenger et al. by adding a preproccessing step, in which the LWE samples are processed by BKZ [CN11], a lattice reduction algorithm. Similar to our discussion in Section 3, we allow preprocessing in our framework, for example to serialize the samples. Therefore, the additional BKZ-based preproccessing step can be captured by our framework as well.