Salsa Picante: A Machine Learning Attack On LWE with Binary Secrets

Cathy Yuanchen Li[*]
cathyli@uchicago.edu
Meta AI
Seattle, USA

Jana Sotáková[*]
ja.sotakova@gmail.com
Meta AI
Seattle, USA

Emily Wenger
ewenger@uchicago.edu
The University of Chicago
Chicago, USA

Mohamed Malhou
mohamed.malhou@polytechnique.edu
Meta AI
Paris, France

Evrard Garcelon
evrard.garcelon@gmail.com
Meta AI
Paris, France

François Charton[†]
fcharton@meta.com
Meta AI
Paris, France

Kristin Lauter[†]
klauter@meta.com
Meta AI
Seattle, USA

ABSTRACT

Learning With Errors (LWE) is a hard math problem underpinning many proposed post-quantum cryptographic (PQC) systems. The only PQC *Key Exchange Mechanism* (KEM) standardized by NIST [13] is based on module LWE, and current publicly available PQ Homomorphic Encryption (HE) libraries are based on ring LWE [2]. The security of LWE-based PQ cryptosystems is critical, but certain implementation choices could weaken them. One such choice is sparse binary secrets, desirable for PQ HE schemes for efficiency reasons. Prior work Salsa [51] demonstrated a machine learning-based attack on LWE with sparse binary secrets in small dimensions ($n \leq 128$) and low Hamming weights ($h \leq 4$). However, this attack assumes access to millions of eavesdropped LWE samples and fails at higher Hamming weights or dimensions.

We present Picante, an enhanced machine learning attack on LWE with sparse binary secrets, which recovers secrets in much larger dimensions (up to $n = 350$) and with larger Hamming weights (roughly $n/10$, and up to $h = 60$ for $n = 350$). We achieve this dramatic improvement via a novel *preprocessing step*, which allows us to generate training data from a linear number of eavesdropped LWE samples ($4n$) and changes the distribution of the data to improve transformer training. We also improve the secret recovery methods of Salsa and introduce a novel cross-attention recovery mechanism allowing us to read off the secret directly from the trained models. While Picante does not threaten NIST's proposed LWE standards, it demonstrates significant improvement over Salsa and could scale further, highlighting the need for future investigation into machine learning attacks on LWE with sparse binary secrets.

[*]Co-first authors.
[†]Co-senior authors, corresponding author: `fcharton@meta.com`

CCS '23, November 26–30, 2023, Copenhagen, Denmark
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3623076

CCS CONCEPTS

• **Security and privacy → Cryptanalysis and other attacks**; • **Computing methodologies → Machine learning**.

KEYWORDS

machine learning, post-quantum cryptography, cryptanalysis

ACM Reference Format:
Cathy Yuanchen Li, Jana Sotáková, Emily Wenger, Mohamed Malhou, Evrard Garcelon, François Charton, and Kristin Lauter. 2023. Salsa Picante: A Machine Learning Attack On LWE with Binary Secrets. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623076

1 INTRODUCTION

The race for post-quantum cryptography (PQC) is well underway. A large-scale quantum computer could solve the hard math problems underpinning most deployed public-key cryptographic systems, like RSA [42], in polynomial time. Small-scale quantum computers have already been built. Consequently, new post-quantum cryptographic systems were proposed and considered for standardization by US National Institute of Standards and Technology (NIST) in the 5-year PQC competition. In July 2022, NIST standardized 4 schemes from the PQC competition [13]. The only key encapsulation mechanism selected—CRYSTALS-Kyber [6]—and one of the three signature schemes—CRYSTALS-Dilithium [25]—are based on the mathematical hardness assumption known as Learning With Errors (LWE) [41]. LWE is also used in proposed PQ homomorphic encryption schemes [2].

LWE works as follows: given an integer modulus q, a dimension n, and a secret vector $\mathbf{s} \in \mathbb{Z}_q^n$, the *Learning With Errors problem* is to recover $\mathbf{s}$ given many random vectors and their noisy inner products with $\mathbf{s}$. These noisy inner products are computed by taking random vector $\mathbf{a} \in \mathbb{Z}_q^n$ and producing $b := \mathbf{a} \cdot \mathbf{s} + e \mod q$, where e is an "error" term sampled from a narrow discrete Gaussian distribution (i.e. taking small values). The adversary is then given the *samples* $(\mathbf{a}, b)$ and attempts to use these to recover $\mathbf{s}$.

The basic LWE problem is assumed to be hard for both classical and quantum adversaries [9, 32, 35, 40, 41]. Variants of LWE, like module-LWE or ring-LWE—on which the NIST-standards and HE schemes are based—add structure to the basic LWE problem, making them potentially easier than LWE. Classical attacks on LWE and its variants typically rely on algebraic techniques for lattice reduction to recover the secret $\mathbf{s}$ from pairs $(\mathbf{a}, b)$ [14, 30]. The error e added to $\mathbf{a} \cdot \mathbf{s}$ to compute b adds noise, making algebraic solutions difficult. Fundamentally, the LWE hardness assumption is that it is hard to learn from noisy data, rendering LWE secret recovery computationally expensive.

On the other hand, the whole field of machine learning (ML) depends on the fact that it is possible to train machines to learn from noisy data. Recent advances in model architectures (e.g. [49]) and training techniques have allowed ML models to glean meaningful trends even from noisy unstructured data. Although LWE samples $(\mathbf{a}, b)$ are noisy, they are highly structured, a fact that ML models can exploit for learning. Thus, it is worthwhile to investigate whether ML-based attacks can enable LWE secret-recovery.

Prior work, Salsa [51], provided an initial proof-of-concept for ML-based LWE attacks. Salsa demonstrated the feasibility of recovering sparse binary secrets, attractive for example in HE applications, for LWE problems with relatively small parameters. Given many LWE samples, Salsa trains ML models to learn the underlying structure of the LWE problem, then leverages the trained model to recover the LWE secret. If the models learn to predict b from $\mathbf{a}$ (even with low accuracy), Salsa can recover the secret $\mathbf{s}$.

Although promising, Salsa has significant limitations. For the largest dimension $n = 128$, Salsa can only recover Hamming weight $h \leq 3$. In comparison, real-world LWE schemes with binary secrets (for homomorphic encryption) start at dimension $n = 512$ or $n = 1024$. Salsa also requires millions of LWE samples $(\mathbf{a}, b)$ for model training, but a real-world attacker would likely only have access to a few samples. Making Salsa's approach realistic requires scaling up the parameters of solvable LWE problems (n, h, and modulus q) while reducing the number of required samples.

Contributions. In this work, we propose Picante, an enhanced ML-based attack on the LWE hardness assumption. Picante leverages basic principles of the original Salsa attack [51]—transformer training, secret recovery—while introducing several novel techniques. This enhanced attack allows recovery of high-dimensional binary secrets with Hamming weight roughly $n/10$ or beyond, requiring only $4n$ samples for training. Table 1 shows the largest Hamming weights we recover for each dimension.

As in Salsa [51, Table 4], we observe that it is easier to learn from vectors with a skewed distribution on the entries. Therefore, we introduce a data preprocessing step that uses lattice-reduction methods to produce samples with smaller coefficients. In contrast to Salsa, Picante starts with a linear number of samples, $m = 4n$, and uses a novel subsampling procedure to generate many more LWE matrices for model training, deduplicated by the aforementioned preprocessing step. These design choices produce numerous non-duplicate samples with skewed entries, and we show that transformers can learn from such data better than from a large set of LWE samples without preprocessing. We also show that, compared to using data preprocessed on independent LWE pairs, the model learns equally well or better using preprocessed data on matrices subsampled from a linear number of LWE pairs.

Dimension	80	150	200	256	300	350
$\log q$	7	13	17	23	27	32
highest h	9	13	22	31	33	60
# possible secrets	2^{32}	2^{61}	2^{97}	2^{133}	2^{147}	2^{227}

Table 1: Salsa Picante's highest recovered secret Hamming weights h. The bottom row lists the approximate number of possible secrets for each n/h combination, for comparison with brute force guessing attacks.

Specifically, this work makes the following contributions:

- **Linear number of samples**: our method only requires a linear number of samples, $m = 4n$ in practice.
- **Data preprocessing:** we preprocess data with classical lattice reduction techniques (e.g. LLL [30] or BKZ [14]), using small block size. This helps transformers learn.
- **Novel secret recovery:** we recover secret bits from the trained transformer, using its *cross-attention* mechanism.

We also improve the data encoding method, introduce rounding to reduce the size of the vocabulary the transformer must learn, improve the distinguisher secret recovery method, and introduce novel combined secret recovery methods. Finally, we compare Picante's performance to classical LWE attacks.

Problem complexity. Instances of cryptographic problems like LWE fall broadly into three buckets:

- easy (solvable via exhaustive search);
- medium-to-hard (requiring significant/unrealistic resources even for best known attacks);
- standardized (believed secure).

Picante considers **medium-to-hard** problems and outperforms some lattice reduction attacks such as uSVP. For example, Picante recovers secrets when $n = 256$ and $h = 31$. In this setting, there are 2^{133} possible binary secrets, so brute force attacks are impossible. Picante succeeds in 70 hours with simple parallelization, while uSVP attacks run on the same machines succeed in 230 – 240 hours (see Table 15). In dimension $n = 350$, $h = 60$ (2^{227} binary secrets), Picante recovers sparse binary secrets in ≈ 250 hours, whereas the uSVP attacks did not recover secrets.

Overall, Picante demonstrates a significant improvement over Salsa, further validating the possibility of ML-based attacks on LWE with sparse binary secrets. Picante cannot (yet) break LWE schemes standardized by NIST, which use larger dimension, smaller moduli q, and more general secret distributions. But it has the potential to scale to these. Further research should explore and expand this line of work.

2 BACKGROUND AND RELATED WORK

Before presenting Picante, we first provide an overview of lattice cryptography and LWE, existing attacks on LWE, and information relevant to the machine learning techniques used in our attack.

2.1 Lattice-based cryptography

Lattice-based cryptography is a major field in post-quantum cryptography. Three out of the four schemes selected by NIST [13] are lattice-based, and two are based on a variant of LWE [41].

Lattices. An n-dimensional integer lattice is the set of all integer linear combinations of n linearly independent vectors in $\mathbb{Z}^n$. More formally, given n vectors $\mathbf{v}_1, \dots, \mathbf{v}_n \in \mathbb{Z}^n$, a lattice is the integer span $\Lambda = \Lambda(\mathbf{v}_1, ..\mathbf{v}_n) := \{\sum_{i=1}^n a_i \mathbf{v}_i \mid a_i \in \mathbb{Z}\}$. The vectors $\mathbf{v}_1, \dots, \mathbf{v}_n$ are called a *basis* for the lattice Λ. The lattice Λ inherits a norm simply by restriction of the Euclidean norm from $\mathbb{R}^n$ to Λ: any vector $\mathbf{v} \in \Lambda$ has norm $\|\mathbf{v}\| = \sqrt{\mathbf{v} \cdot \mathbf{v}}$.

Hard Lattice Problems. Lattices give rise to several *hard* problems—problems for which the best known algorithms require exponential time in the dimension n for both classical and quantum computers. The most famous and widely-studied is the *Shortest Vector Problem* (SVP): for a lattice Λ, find a nonzero vector $\mathbf{v} \in \Lambda$ with minimal norm. Currently, the best algorithms for SVP take exponential space and time in n [33]. This makes lattices attractive building blocks for post-quantum cryptography.

Learning with Errors (LWE). Many lattice-based cryptographic schemes leverage the "Learning with Errors" problem, which is defined as follows. Fix a lattice dimension n, modulus q, number of samples m and a narrow Gaussian probability distribution χ. The "Learning with Errors" (LWE) problem is to recover a secret vector $\mathbf{s} \in \mathbb{Z}_q^n$ given a collection of m noisy samples $(\mathbf{a}_\mathbf{i}, b_i)$, where $\mathbf{a}_1, \dots, \mathbf{a}_\mathbf{m} \leftarrow_R \mathbb{Z}_q^n$ are random vectors, and $b_i = \mathbf{a}_\mathbf{i} \cdot \mathbf{s} + e_i \bmod q$ are noisy inner products. The $e_i \in \mathbb{Z}_q$ are sampled independently from the error distribution χ. A *LWE instance* is given by a matrix $(\mathbf{A}, \mathbf{b}) \in \mathbb{Z}_q^{m \times n} \times \mathbb{Z}_q^m$, where $\mathbf{A}$ is uniformly random in $\mathbb{Z}_q^{m \times n}$ and $\mathbf{b} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e} \bmod q$ is a column vector. The vector $\mathbf{s} \in \mathbb{Z}_q^n$ is the secret vector, and $\mathbf{e} \in \mathbb{Z}_q^m$ is an error vector with entries sampled from the probability distribution χ. We call any of the pairs $(\mathbf{a}_i, b_i)$, or equivalently any row of the matrix $(\mathbf{A}, \mathbf{b})$, an LWE *sample*.

Hardness of LWE. In 2005, Regev demonstrated a worst-case quantum reduction from the SVP to LWE [41]. Regev also showed that LWE-based cryptographic schemes were far more efficient than existing lattice cryptography methods. Later work demonstrated that LWE is classically as hard as worst-case SVP-like problems [9, 32, 35]. Hence, LWE is considered a solid foundation for (post-quantum) lattice cryptography.

Real-world LWE-based cryptographic schemes. LWE-based schemes are not only standardized for Post-Quantum Cryptography [6, 25] and Homomorphic Encryption [2], but also allow for a range of cryptographic constructions beyond key exchange and signatures, including group signatures, secret sharing, and multi-party computation. The NIST standardization competition received 23 entries proposing schemes based on lattice assumptions such as LWE. In CRYSTALS-Kyber [6], the dimension is $n = k \times 256$ for $k = 2, 3, 4$, where k is a parameter of Kyber's module-LWE scheme. The LWE-based signature scheme CRYSTALS-Dilithium [25] uses similar size of n. Both use secret vectors with small integer coordinates, centered around 0. Another LWE-based NIST submission, LIZARD, suggests LWE dimensions n from 544 to 736 [17, Table 2].

Homomorphic encryption schemes in publicly available libraries such as SEAL use dimension $n = 512$ only for small computations, and generally require dimensions $n = 1024, 2048$ and other powers of 2 up to 2^{15}. HE implementations commonly use binary or ternary secrets for efficiency (see [2]), and many implementations propose using a sparse (binary) secret with Hamming weight $h << n$. For instance, HEAAN uses $n = 2^{15}$, $q = 2^{628}$, ternary secret and Hamming weight 64 [15]. For more on the use of sparse binary secrets in LWE, see [3, 16, 20]. We focus on the case of a binary secret with Hamming weight h and error distribution χ, a centered Gaussian with $\sigma = 3$. $\sigma = 3.2$ is the typical choice for homomorphic encryption [2, 16, 20, 48].

2.2 Attacks on LWE

The LWE problem is assumed to be exponentially hard to solve with classical [9, 32, 35] or quantum [40] algorithms. Due to LWE's prominence as a hard problem in post-quantum cryptography, a significant body of work has been devoted to attacking it.

Classical Attacks. Most existing classical attacks on LWE leverage *lattice reduction* techniques, which reduce the problem to recovering the shortest vector in a lattice. The LLL [30] algorithm runs in polynomial time in the dimension of the lattice (the optimized *fplll* [22] implementation runs in time $O(n^4 \log(q)^2)$), but it recovers an exponentially bad approximation to the shortest vector. LLL can be improved using the *block Korkine-Zolotarev* method (BKZ) by Schnorr [45] and Schnorr-Euchner [46]. The BKZ algorithm finds shortest vectors in projected lattices of dimension $k < n$, where k is referred to as the *block size*. The BKZ approach relies on an exponential time sub-algorithm applied for increasing block sizes, but can recover shorter vectors than LLL. The main 3 attacks used to estimate secure parameters for lattice-based cryptography are: the uSVP, dual, decoding attacks, all of which require finding a short vector in a particular lattice arising from the LWE instance. The uSVP attack uses Kannan's embedding [26] to embed the problem into a lattice such that the (unique) shortest vector reveals the secret s. For concrete choices for this embedding, see [12]. The Homomorphic Encryption Standard [2, Section 2.1.2] describes the uSVP, dual, and decoding attacks in detail.

Salsa: a machine learning based attack. Salsa [51] demonstrated the possibility of training machine learning (ML) models to attack LWE for sparse binary secrets. Salsa trained universal transformers to predict b from input $\mathbf{a}$ and developed secret recovery techniques to extract the secret that is implicitly learned by the model. Salsa successfully recovered secrets for LWE problems with dimension $n \leq 128$ and Hamming weight ≤ 4.

2.3 Machine learning preliminaries

Here we provide a brief background and intuition behind the ML techniques used in our attack, Picante.

ML basics. The generic goal of machine learning is to compute a model $\mathcal{M}$ that maps an input $x \in \mathcal{X}$ to an output $y \in \mathcal{Y}$. The model $\mathcal{M}$ is computed via a *supervised training process*, during which it is shown samples (x', y') such that $x' \in \mathcal{X}' \subset \mathcal{X}$ and

$y' \in \mathcal{Y}' \subset \mathcal{Y}$. During this training process, the parameters θ of $\mathcal{M}$ are iteratively updated to minimize a predefined *loss function*, $l(\mathcal{M}, x', y', \tilde{y})$, where $\tilde{y}$ is the model's predicted output given input x' (e.g. $\mathcal{M}(x') = \tilde{y}$), and y' is the ground truth output.

In Picante, a model $\mathcal{M}$ is trained using LWE samples ($\mathbf{a}$, b) as (x, y) pairs. Hence, models are given an input vector $\mathbf{a}$ and asked to predict the LWE output b, where $b = \mathbf{a} \cdot \mathbf{s} + e$. We use the cross-entropy loss function, a common choice in ML model training. For our model, we use the well-known *transformer* architecture.

Transformers. Transformers were introduced in [49] for natural language processing (NLP) and machine translation. In recent years, they have been applied to a wide range of problems, from text and image generation [37–39] to image processing [10] and speech recognition [23], where they now achieve state-of-the-art performance [24, 50]. Transformers have also been proposed for problems in mathematics, like symbolic integration [29], theorem proving [36], and numerical computations [11]. Transformers process sequences of tokens (in NLP, sequences of words, making up sentences). They combine a multi-head attention mechanism [7] that takes care of relations between different tokens in the sequence, essentially "decorrelating" it, and a fully-connected neural network (FCNN), which processes the decorrelated sequences. More details about our transformers are in §4.2.

3 INTRODUCING SALSA PICANTE

Before diving into the details of Picante's methodology, we first present a high level overview of the attack. Salsa Picante builds upon Salsa and progresses in three stages: (1) data preprocessing, (2) model training, and (3) secret recovery (see Figure 1).

Each run of Salsa Picante targets LWE for a fixed dimension n, modulus q, binary secret s with Hamming weight h, and error distribution χ with $\sigma = 3$. Salsa Picante requires m *original LWE pairs*, sharing the same secret s. These are of the form $(\mathbf{a_i}, b_i)$, with $b_i = \mathbf{a_i} \cdot \mathbf{s} + e_i$. In real world situations, these samples must be collected. In experimental settings we choose $m = 4n$ and generate these samples randomly. After these parameters are fixed, the attack proceeds via the following three stages:

(1) Data preprocessing. During this step, n LWE pairs are randomly selected from the set of original samples and stacked into an $n \times n$ matrix $\mathbf{A}$ and vector $\mathbf{b}$ of length n. The matrix $\mathbf{A}$ is processed using a basis-reduction algorithm (BKZ), and the same linear operations are performed on $\mathbf{b}$. This creates reduced LWE pairs with smaller norms but larger errors. This step is repeated to produce 2^{22} reduced LWE pairs.

(2) Model Training. The reduced LWE samples $(\mathbf{a}, b)$ are encoded as sequences of numbers, represented in base B, and used to train a transformer model $\mathcal{M}$. The model $\mathcal{M}$ learns to predict b from $\mathbf{a}$. Model training proceeds in *epochs*, each using 2 million samples. The 4 million training data are shuffled randomly every 2 epochs.

(3) Secret Recovery. At the end of each epoch, Salsa Picante attempts to recover the secret using 3 techniques: direct, distinguisher, and cross-attention. The methods are used separately and can be combined to provide more secret guesses. Secret guesses are evaluated. Model training stops if the secret is recovered; else, another epoch begins.

4 ATTACK METHODOLOGY

Now, we provide a detailed description of Picante, which progresses in the three stages outlined above.

4.1 Stage 1: data preprocessing

(1.1) Collect LWE samples. The Picante attack begins by collecting a set of LWE samples with fixed parameters, as described in §3. Salsa assumed the attacker had access to $4,000,000 \approx 2^{22}$ LWE pairs ($\mathbf{A}$, $\mathbf{b}$) with the same secret, since transformers, the model architecture used in both Salsa and Salsa Picante, typically train on millions of examples. However, access to this many samples is unrealistic in practice. To mitigate this, Picante introduces TinyLWE, a technique that only requires $m = 4n$ LWE pairs – linear in the dimension n. Thus, the attack collects the $4n$ pairs and then runs TinyLWE.

(1.2) Recombine to expand LWE sample set. The goal of TinyLWE is to produce the large set of 4 million samples required to train our models, from a small initial set of $m = 4n$ LWE pairs. Prior work [5, 51] observed that a set of m LWE pairs $(\mathbf{a}_1, b_1) \ldots (\mathbf{a}_m, b_m)$ can always be expanded by considering the linear combinations $(\mathbf{a}, b) = (\sum_i c_i \mathbf{a}_i, \sum_i c_i b_i)$, with $c_i \in \mathbb{Z}$ and $\sum_i |c_i|$ small. We could, therefore, generate a large set of samples from a small initial set of LWE pairs by creating many such linear combinations.

Unfortunately, LWE error is amplified by linear combinations: $e' = b - \mathbf{a} \cdot \mathbf{s} = \sum_i c_i e_i$, with e_i the error in the original LWE sample. Assuming that the c_i are centered, the standard deviation of error grows as the square root of the number of terms in the combination ($\sqrt{m}$ in the general case) times the standard deviation of the distribution of c_i (which is $\sqrt{C/3}$ if we assume the c_i are uniformly distributed in $[-C, C]$). In addition, the initial LWE error is further amplified by the data reduction step. So although the transformers used in Picante can handle noisy data, generating samples via linear combinations would bring error to a level where training and secret recovery becomes very difficult.

Instead of linear combinations, Picante uses *subsampling*. Subsets of n out of the m original LWE samples, $(\mathbf{a}_{j_1}, b_{j_1}), \ldots (\mathbf{a}_{j_n}, b_{j_n})$, are randomly selected, and arranged in a matrix $\mathbf{A}$, with rows $\mathbf{a}_{j_1}$ to $\mathbf{a}_{j_n}$. Because the original LWE pairs are merely copied into $\mathbf{A}$, the associated noisy inner products have the same error distribution as the original samples. This technique produces up to $\binom{4n}{n}$ unique matrices ($\approx 9.48^n \cdot 0.46/\sqrt{n}$)). In Picante, we use subsampling to generate about $2^{21}/n$ matrices, which, after the reduction step described next, results in about 2^{22} reduced LWE pairs. Subsampled matrices often have rows in common, but we experimentally observe that, after reduction, there are almost no duplicate vectors. For $n = 80$, we counted one duplicate in 50,000 examples; for $n \geq 150$, we found no duplicates in 4 million examples.

Note that *subsampling* is different from *batching*. Subsampling allows us to generate a training set of 4 millions examples from only $4n$ LWE samples. This is accomplished in the preprocessing step, which performs data reduction on subsets of the $4n$ original samples. Batching, on the other hand, takes place during training, when computing the gradients of the loss function, that are then used to optimize the model. Instead of computing gradients on a

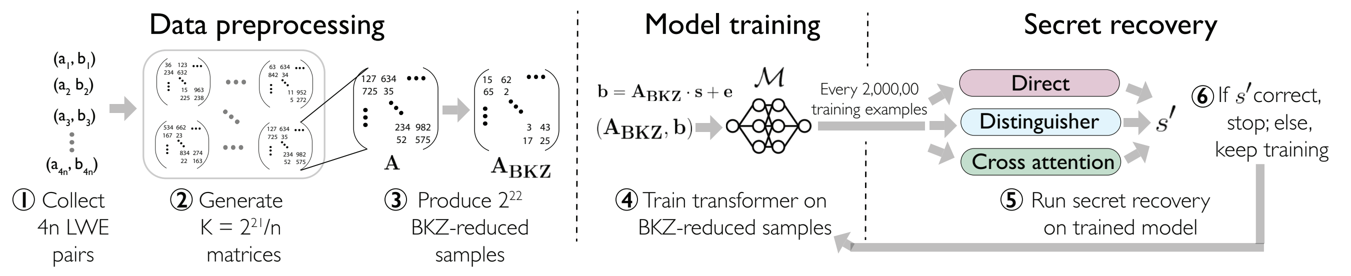
Data preprocessing
Model training
Secret recovery
① Collect 4n LWE pairs
② Generate K = 2^21/n matrices
③ Produce 2^22 BKZ-reduced samples
④ Train transformer on BKZ-reduced samples
⑤ Run secret recovery on trained model
⑥ If s′ correct, stop; else, keep training
Every 2,000,00 training examples
Direct
Distinguisher
Cross attention
b = A_BKZ · s + e
(A_BKZ, b)
A
A_BKZ

Figure 1: An end-to-end overview of Salsa Picante's attack methodology.

single training example, batching averages them over many examples, allowing for faster training and better estimate of gradients. Picante uses batches of 128 examples.

(1.3) Reduce samples. After sampling and recombination, the attack runs a final *reduction* step to make the LWE samples more amenable to model training. The motivation for this step comes from experimental observations in Salsa. Salsa could only recover binary secrets with low Hamming weights: up to 4 non-zero bits in the secret. However, the Salsa authors observed [51, Table 4] that, if the coordinates of the samples $\mathbf{a}$ used to train the model were bounded by αq, with $\alpha < 0.6$, binary secrets with Hamming weights up to 15 could be fully or partially recovered for $n = 50$. We confirmed this result for larger dimensions, and different restrictions on $\mathbf{a}$ (e.g. $a_i \leq \alpha q$ for different α). Unfortunately, in practical settings, the coordinates of $\mathbf{a}$ are sampled from a uniform distribution over $\mathbb{Z}_q$, making this technique useless for real world attacks.

Picante turns this observation into a practical attack technique by leveraging existing *lattice reduction* methods. Such methods reduce the size of the coordinates of LWE samples naturally, yielding the same effect as the Salsa $\mathbf{a}$-limiting technique. We find experimentally that reducing LWE samples via these methods before model training allows recovery of secrets with much higher Hamming weights. The reduction technique proceeds as follows.

Given n LWE samples, stored as the rows of a $n \times n$ matrix $\mathbf{A}$, and a corresponding vector $\mathbf{b}$ of noisy inner products with a fixed secret $\mathbf{s}$, we can create a matrix $\mathbf{A}'$ with smaller entries than $\mathbf{A}$ by applying standard basis-reduction algorithms like LLL [30] and BKZ [45] to Λ, the n-dimensional lattice defined by the rows of $\mathbf{A}$. In Picante, we run BKZ (from the *fplll* package [22]) on the matrix:

$$\begin{bmatrix} \omega \cdot \mathbf{I}_n & \mathbf{A}_{n\times n} \\ 0 & q \cdot \mathbf{I}_n \end{bmatrix},$$

with $\omega \in \mathbb{Z}$ an error penalization parameter, discussed below. Since the BKZ reduction is a change of basis, it is a linear transformation, which we can represent as $\begin{bmatrix} \mathbf{R}_{2n\times n} & \mathbf{C}_{2n\times n} \end{bmatrix}$. The BKZ reduction can be written as a matrix multiplication

$$\begin{bmatrix} \mathbf{R}_{2n\times n} & \mathbf{C}_{2n\times n} \end{bmatrix} \begin{bmatrix} \omega \cdot \mathbf{I}_n & \mathbf{A}_{n\times n} \\ \mathbf{0} & q \cdot \mathbf{I}_n \end{bmatrix} = \begin{bmatrix} \omega \cdot \mathbf{R} & \mathbf{RA} + q\mathbf{C} \end{bmatrix},$$

with matrices $\mathbf{R}$ and $\mathbf{C}$ chosen so that $\begin{bmatrix} \omega \cdot \mathbf{R} & \mathbf{RA} + q\mathbf{C} \end{bmatrix}$ has $2n$ rows with small norms. The matrix $q\mathbf{C}$ adds an integer multiple of q to each entry in $\mathbf{RA}$, so that all entries are in the range $(-q/2, q/2)$.

Applying the linear transformation $\mathbf{R}$ to $\mathbf{b}$, we create a new LWE instance $(\mathbf{RA}, \mathbf{Rb})$ with the same secret $\mathbf{s}$ and smaller coordinates $\mathbf{RA}$ but *a different error distribution.* Let $\mathbf{e} = \mathbf{b} - \mathbf{A} \cdot \mathbf{s}$ be the initial LWE error. After reduction, the error becomes $\mathbf{e}' = \mathbf{Rb} - (\mathbf{RA}) \cdot \mathbf{s} = \mathbf{R}(\mathbf{b} - \mathbf{A} \cdot \mathbf{s}) = \mathbf{Re}$. Thus, as $\mathbf{R}$ entries grow, LWE error is amplified. All computations are performed mod q.

Error amplification can be controlled by the error penalization parameter ω. Recall that BKZ computes $\mathbf{R}$ and $\mathbf{C}$ so that the norms of the rows of $\begin{bmatrix} \omega \cdot \mathbf{R} & \mathbf{RA} + q\mathbf{C} \end{bmatrix}$ are small. A large ω encourages small entries in the rows of $\mathbf{R}$ but hinders the norm reduction of $\mathbf{RA} + q\mathbf{C}$, and therefore limits the reduction of $\mathbf{A}$ coordinates. The choice of ω controls a trade-off between the amount of reduction of $\mathbf{a}$ we can achieve, and the amount of additional noise which gets injected in the transformed samples. In practice, we set $\omega = 15$.

When more than n pairs are available (e.g. the million of pairs produced by Step 1.2), they are divided into batches of n and processed as above. Thus, n LWE pairs are transformed into a matrix $\mathbf{RA}$ with $2n$ rows, which produces $\approx 2n$ reduced LWE samples (for $n \geq 256$, we observe about 1% zero rows; this fraction is larger for smaller n).

Note on reduction algorithm choice. We experimented with two standard basis-reduction algorithms: LLL and BKZ. Note that our objective is not to find the shortest vector in the lattice defined by $\mathbf{A}$ (the traditional goal of LLL/BKZ), but to transform $\mathbf{A}$ into a matrix with smaller coefficients. Experimentally, we find that BKZ with small block size ($\beta = 16 - 20$) achieves better reduction than LLL. BKZ speed-ups, such as BKZ2.0 [14], do not seem to result in improved reduction. For BKZ, the block sizes needed to achieve reduction in Picante are significantly smaller than the block sizes that would be required to perform a lattice-reduction attack on problems of the same dimensions (see also § 7).

4.2 Stage 2: model training

After the data is prepared, the attack enters the model training stage. Although there are no sub-stages to model training, here we break down the model training step into several components: data encoding, model architecture choice, and the training itself.

Encoding LWE pairs. Prior to training, Picante encodes the LWE samples (i.e. $(\mathbf{a}, b)$ pairs) as sequences of tokens that the transformer can process. After encoding, the integer coordinates of $\mathbf{a}$ and b are represented as two digit numbers in base B (with $B \geq \sqrt{q}$). Our experiments with different values of B (see § 6.2) suggest that large values of B, which limit the most significant digit of a_i and b to a small number of values (i.e. $B \approx q/k$ with k small), allow for better performance. In our experiments, we use $B = \lfloor q/k \rfloor$ with $k = 2 \cdot \lceil \frac{n}{100} \rceil + 2$.

This creates a problem for large dimensions. The large values of q and B (for $n \geq 20$ we have $q > 100,000$ and $B > 16,600$)

result in large token vocabularies, which are difficult to learn for a transformer trained on 4 million LWE pairs only. To mitigate this, we encode the lowest digits of **a** and b into B/r *buckets* of size r (i.e. integer divide them by r). The value r is chosen so that the overall vocabulary size $B/r < 10,000$ (see Table 2). The use of buckets helps train models for large n but it also causes a loss of precision in the values of **a** and b. We believe the impact on performance is limited, because the low bits of **a** and b that are rounded off by buckets are those most corrupted by LWE error.

Model architecture. As noted previously, PICANTE uses a transformer model architecture. This architecture, summarized in Figure 2, is strongly inspired by SALSA [51]. Following [49], it uses a sequence-to-sequence (seq2seq) model [18], composed of two transformer stacks – an encoder and a decoder – connected by a cross-attention mechanism. The encoder processes the input sequence, the coordinates of **a**, represented as sequences of digits. The discrete input tokens are first projected over a high-dimensional space (we use dimension $d = 1024$) by a Linear Embedding Layer with trainable weights (i.e. embedding is learned during training). The resulting sequence is then processed by a single-layer transformer: a self-attention layer with 4 attention heads, and a FCNN with one hidden layer of 4096 neurons.

The decoder is an auto-regressive model. It predicts the next token in the output sequence, given already decoded output and the input sequence processed by the encoder. Initially, the decoder is given a beginning of sequence token (BOS), and predicts b_1^*, the first digit of b. It is then fed the sequence BOS, b_1^*, and decoding proceeds until the end-of-sequence token (EOS) is output.

Decoder input tokens are encoded as 512-dimensional vectors via a trainable embedding (which also decodes transformer output). The decoder has two layers. First, a shared layer (as in [21]), which is iterated through 8 times, feeds layer output back into its input. This recurrent process is controlled by a copy-gate mechanism [19], which decides whether a specific token should be processed by the shared layer or just copied *as is*, skipping the next iteration. After 8 iterations, the output of the shared layer is fed into a "regular" transformer layer. Finally, a linear layer processes the decoder output and computes the probabilities that any word in the vocabulary is the next token. The largest probability is selected via a softmax function (a differentiable counterpart of the max function).

Decoder layers are connected to the encoder via a *cross-attention* mechanism with 4 attention heads. In each head, the output of the encoder $E = (E_i)_{i\in\mathbb{N}_l}$ (with l the input sequence length) is multiplied by two trainable matrices, W_K and W_V, yielding the *Keys* $K = W_K E$ and *Values* $V = W_V E$. The 512-dimensional vector to be decoded, D, is multiplied by a matrix W_Q, yielding the *Query* $Q = W_Q D$. The l scores are calculated from the query and keys:

$$\text{scores}(E, D) = \text{Softmax}((W_Q D)(W_K E)^T).$$

The scores measure how important each encoder input element is when decoding D (i.e. computing b). The cross-attention value for this head is the dot product of scores and values. The values of different heads are then processed by a final linear layer. Cross-attention scores quantify the relation between input positions and output values. PICANTE uses them to recover the secret bit by bit.

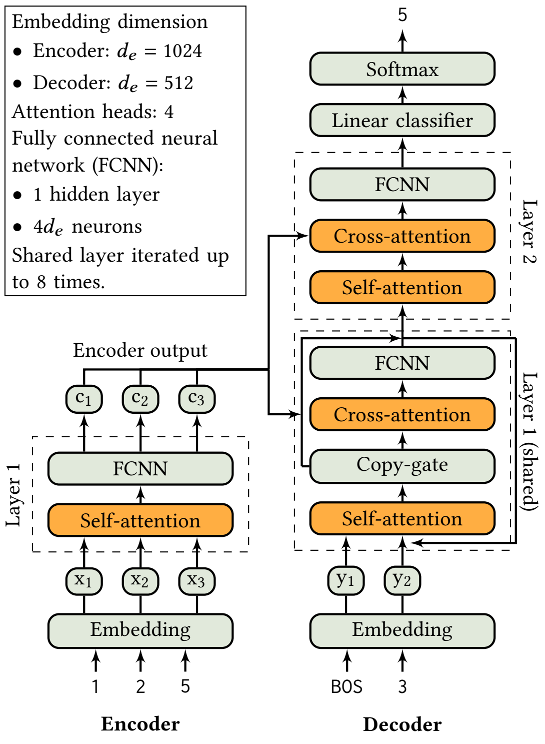

Figure 2: Our transformer architecture.

Model training. After encoding the samples, the attacker trains the transformer $\mathcal{M}$ to predict b from **a**. PICANTE frames this as a supervised multi-classification problem, i.e. minimizing the loss:

$$\min_{\theta\in\Theta} \sum_{i=1}^{N} \sum_{j=1}^{K} \sum_{k=1}^{V} \mathbf{1}[y_i[j] = k-1] \frac{e^{\mathcal{M}(x_i)[j,k]}}{\sum_{k'=1}^{V} e^{\mathcal{M}(x_i)[j,k']}}, \tag{1}$$

where $\mathcal{M}(x_i) \in \mathbb{R}^{K\times V}$ are model logits evaluated at x_i, $\theta \in \Theta$ are the model parameters, N the training sample size, $K = 2$ the output sequence length and $V = B/r$ the vocabulary size.

Solving (1) requires minimizing the cross entropy between model predictions $\mathcal{M}(\mathbf{a})$ and the ground truth b, over all tokens in the output sequence. Alternatively, one could define this as a regression problem, but we believe classification is better adapted to the modular case. Prior works confirm that reformulating regression as classification leads to state-of-the-art performance [1, 43, 44, 47].

Training proceeds via batches of $n_b = 128$ examples. The cross-entropy loss $\mathcal{L}(\mathcal{M}, \mathbf{a}, b)$ is computed over all batch examples, and gradients $\nabla\mathcal{L}$ are calculated with respect to the model parameters (via *back-propagation*). Model parameters are then updated using the Adam optimizer [27], by $\text{lr}\nabla\mathcal{L}$. The learning rate, lr is set to 10^{-5}, except during the 1000 first optimizer steps, where it is increased linearly from 10^{-8} to 10^{-5}. Every 2 million examples (an *epoch*), model performance is evaluated on a held-out sample, and PICANTE attempts to recover the secret. If it fails, another epoch begins.

4.3 Stage 3: secret recovery

After every training epoch, Picante attempts secret recovery. The intuition behind secret recovery is that if a model $\mathcal{M}$ can predict b from $\mathbf{a}$ with higher-than-chance accuracy, then $\mathcal{M}$ must somehow "know" the secret key $\mathbf{s}$, and we can recover $\mathbf{s}$ from $\mathcal{M}$. Salsa Picante uses 3 methods—*cross attention*, *direct recovery*, and *distinguisher* —for recovery. These can be combined for greater accuracy.

In this section, we assume that the attacker knows the Hamming weight h of the secret to be recovered. This is the only part of the attack where this assumption is made. If h is not known, then secret recovery is run for increasing values of h until the secret is found.

Cross-Attention. In this novel recovery method, Picante guesses the secret from the parameters of $\mathcal{M}$ by leveraging the cross-attention scores of the first decoder layer (see Figure 2 and § 4.2). Intuitively, the cross-attention score measures the relevance of input tokens (i.e. coordinates of $\mathbf{a}$) for the computation of b. Since $b = \mathbf{a} \cdot \mathbf{s} + e$, the coordinates of $\mathbf{a}$ that correspond to the 0 bits of $\mathbf{s}$ have no impact on b. On the other hand, the coordinates associated to the 1 bits in $\mathbf{s}$ have an impact proportional to their value. Therefore, high cross-attention scores should be found for the input positions that correspond to 1s in the secret.

To run this method, Picante evaluates the trained transformer on a test set of $10,000$ reduced LWE samples and sums the cross-attention scores of all heads. Since $\mathbf{a}$ has n coordinates encoded on 2 tokens, this produces a $2n$-dimensional vector, from which the odd positions are kept (i.e. the high digits of $\mathbf{a}$ coordinates), generating an n-dimensional score vector V. A secret guess $\mathbf{s}'$ is then produced by setting the h largest coordinates of V to one, and the rest to 0.

Direct recovery. Picante uses the same direct recovery method as Salsa. This technique leverages trained transformers' ability to generalize on inputs not seen during training. The trained model is evaluated on special vectors $\mathbf{a}$ with one non-zero coordinate: $\mathbf{a} = K\mathbf{e}_i$ with $\mathbf{e}_i$ the i-th standard basis vector and $K \in \mathbb{Z}_q$. For these vectors, since $b = \mathbf{a} \cdot \mathbf{s} + e$, and e is small, $b \approx 0$ if the i-th bit in the secret $s_i = 0$, and $b \approx K$ if $s_i = 1$ (see [51] for details). In practice, different K_j are chosen, and the transformer is run on $K_j \cdot \mathbf{e}_i$ for $i = 1, \dots, n$, identifying potential 1-bits in the secret as above and producing a secret guess for each K_j.

To obtain a score for each bit to be used in combination methods, for each index i, we sum the resulting values of $\mathcal{M}(K_j \cdot \mathbf{e}_i)$ (or, equivalently, take the mean). We then guess the secret $\mathbf{s}'$ by assuming that the h largest coordinates are 1 and the rest are 0.

Distinguisher. Picante's version of distinguisher recovery improves upon that of Salsa. The general idea is that if the i-th bit of the secret $s_i = 0$ and $\mathbf{e}_i$ is the i-th standard basis vector, then the model should predict close values for $\mathbf{a}$ and $\mathbf{a} + K \cdot \mathbf{e}_i$. Salsa's distinguisher took a LWE sample $(\mathbf{a}, b)$ and compared b with the model prediction $b' = \mathcal{M}(\mathbf{a} + K \cdot \mathbf{e}_i)$ for some random $K \in \mathbb{Z}_q$. This presupposed relatively high model accuracy, i.e. $\mathcal{M}(\mathbf{a}) \approx b$, which rarely happens in practice. In Picante, $b' = \mathcal{M}(\mathbf{a} + K \cdot \mathbf{e}_i)$ is compared to $\mathcal{M}(\mathbf{a})$ instead of to b. The rest of the method is unchanged, other than implementation improvements. The secret is guessed by setting the h highest-scoring secret bits to 1, and the rest to 0.

This improved method has two benefits. First it exploits trained model consistency without requiring prediction accuracy. In practice, this means recovery can happen earlier during training, when model prediction accuracy is low. Second, it does not need additional LWE samples $(\mathbf{a}, b)$ (as was the case in Salsa), and can be run from randomly generated $\mathbf{a}$. This reduces the number of LWE samples necessary for the attack.

This recovery method relies on a large number of model inferences, which can make it very slow for large dimension. To increase its speed, we use the same $\mathbf{a}$ across all secret bits s_i, halving the number of inferences relative to those required in Salsa.

Combined secret recovery. Each recovery method outputs a score for every bit in the secret. The secret guess is computed by setting the h bits with the largest scores to 1 (and the other bits to 0). By combining the scores from different methods, we create four additional techniques, which can sometimes can recover secrets when individual methods fail. The combined methods are as follows:

- *Aggregated rank.* The bit scores produced by each method are sorted from largest to smallest, and replaced by their rank. The h bits with the highest ranks (Highest Rank) or highest summed ranks (Sum Rank) are set to 1.
- *Aggregated normalized scores.* The bit scores produced by each method are normalized to $[0, 1]$. The h bits with the maximum normalized scores (Max Normalized) or the highest sum of normalized scores (Sum Normalized) are set to 1.

These combination rules essentially amount to setting secret bits to 1 for bit positions where all, some, or any of the secret recovery methods have a high score. We use aggregated scores from all subsets of the secret recovery methods. Other combination rules could be considered. These mixing techniques are cheap to implement, because they do not require additional model inferences.

Checking correctness. Recovery methods make guesses $\mathbf{s}'$ about the (unknown) secret $\mathbf{s}$. The test for whether $\mathbf{s}' = \mathbf{s}$ was introduced in Salsa. On a test sample of N_{test} LWE pairs $(\mathbf{a}_i, b_i)_{1 \leq i \leq N_{\text{test}}}$, compute $b'_i = \mathbf{a}_i \cdot \mathbf{s}'$, and consider the distribution r of $r_i = b'_i - b_i \mod q$. If $\mathbf{s}' = \mathbf{s}$, then $r \approx e$, the LWE error, with standard deviation σ. If $\mathbf{s}' \neq \mathbf{s}$, then r will be approximately uniformly distributed over $\mathbb{Z}_q$, with standard deviation $\sigma' = q/\sqrt{12}$. By estimating σ' on a large set of samples, one can verify $\mathbf{s}' = \mathbf{s}$ to any confidence level.

This test can be performed on the original set of LWE samples collected by the attacker, e.g. with $N_{\text{test}} = m = 4n$. In §A.2 we statistically analyze this verification technique and demonstrate that this sample size is sufficient for all lattice dimensions $n \geq 80$.

5 SALSA PICANTE'S PERFORMANCE

We now evaluate Picante's performance over a variety of parameter settings for lattice dimension, modulus size, Hamming weight, and number of samples. All Picante experiments are based on the following choices, with the exact parameters used in our experiments listed in Table 2. Other details are in §4.

5.1 Experimental settings

- For each n, the modulus q is selected after consulting Table 1 in [12]. We set our q such that $\log_2 q$ is smaller than the smallest successful lattice-reduction attack reported there (see Table 2). A smaller q makes attacks more difficult.

- The error in the original LWE samples is sampled from a discrete Gaussian distribution, centered at 0, and with $\sigma = 3$, a common choice when LWE is used in homomorphic encryption [2, 3].
- We consider binary secrets with sparsity $h/n \approx 10\%$ or larger. For each n where we evaluate PICANTE, we show results for seven different Hamming weights $h \approx n/10$ to confirm repeatability.
- The attack starts with a set of $4n$ randomly generated samples $(\mathbf{a}, b)$ with fixed n, q, sparse binary $\mathbf{s}$, and σ.
- For the BKZ reduction step (PICANTE stage 1.3), we use $\omega = 15$ as the error penalization parameter for all n. Block size and the LLL parameter δ in *fplll* are set to 20 and 0.99 for all $n \leq 200$. To keep preprocessing times reasonable, we decrease these values for $n = 256$, 300 and 350 (see Table 8).
- For each n and q, we perform the preprocessing step on random matrices A once and use that reduced data for experiments with different secrets.

n	q	$\log_2 q$	δ	β	base	r
80	113	7	0.99	20	29	1
150	6421	13	0.99	20	1071	1
200	130769	17	0.99	20	21795	2^2
256	6139999	23	0.96	18	767500	2^7
300	94056013	27	0.96	16	11757002	2^{11}
350	3831165139	32	0.96	14	383116514	2^{16}

Table 2: PICANTE parameters. n: dimension, q: modulus, δ: delta-LLL (BKZ), β: block-size (BKZ), base: encoding base, r: bucket size for encoding.

5.2 Overall Performance

A summary of PICANTE's results is shown in Table 3, which records PICANTE's success for various dimensions n, modulus q, and Hamming weight h. We run multiple experiments for each parameter setting, and report the number of successes/attempts, as well as the model training epochs at which successful secret recoveries occurred. For example, in dimension $n = 350$, we recovered a secret with $h = 60$ in one out of five trials (each trial has a different secret). In that case, the recovery happened in training epoch 38.

Effect of dimension n. For dimensions up to 300, PICANTE consistently recovers LWE secrets with sparsity $h/n \approx 10\%$, a significant improvement over SALSA. For $n = 350$, PICANTE can recover secrets with Hamming weight $h = 60$, sparsity $\approx 17\%$. We believe this improved performance is due to the preprocessing parameters used for $n = 350$ (§6.1). This suggests that harder LWE problems, with dimension $n = 350$ but smaller q, could be solved with this architecture for $h \approx 0.1n$.

For all dimensions, PICANTE succeeds for smaller values of the modulus q than those for which the concrete, classical lattice attacks in [12, Table 1] can recover secrets with BKZ blocksize roughly 40. In our experiments, we use a fixed q for each dimension, to avoid running the costly preprocessing step multiple times. Evaluating performance at varying q for a fixed n is important future work.

Effect of Hamming weight h. For each dimension n, we evaluate PICANTE on secrets with a range of Hamming weights. For each n, there is a "cutoff" Hamming weight, above which PICANTE did not successfully recover the secret in these runs. This is expected,

$n, \log_2 q$	Hamming weight h						
80, 7	4	5	6	7	8	**9**	10
success	3/5	3/5	2/5	2/5	1/20	1/20	0/20
epoch	2,5,6	0,1,4	0,3	0,8	3	4	
150, 13	9	10	11	12	**13**	14	15
success	4/5	2/5	3/5	1/5	1/20	0/20	0/20
epoch	1,1,3,6	2,2	8,8,11	9	13		
200, 17	17	18	19	20	21	**22**	23
success	3/5	2/5	3/5	1/5	2/5	2/20	0/20
epoch	1,1,8	2,11	2,3,9	7	7,10	12,17	
256, 23	26	27	28	29	30	**31**	32
success	4/5	1/5	1/5	3/5	3/5	4/20	0/20
epoch	2,3,4,7	10	5	5,9,11	17,20,32	6,12,26,27	
300, 27	28	29	30	31	32	**33**	34
success	2/5	2/5	1/5	1/5	2/5	1/5	0/20
epoch	6,7	6,13	11	11	21,31	39	
350, 32	54	55	56	57	58	59	**60**
success	2/5	1/5	1/5	1/5	1/5	1/5	1/5
epoch	10,20	10	46	42	39	18	38

Table 3: Success rates and number of epochs. Highest recovered Hamming weight for each dimension n is in bold.

because increasing Hamming weight makes the problem more difficult. Table 3 presents the cutoff value in bold, along with the number of successfully recovered secrets for each Hamming weight.

Required training duration. Figure 3 shows the number of epochs needed for secret recovery for $80 \leq n \leq 300$ and different values of h. Whereas 66% of successful secret recoveries occurred during the first 10 epochs, the number of epochs before recovery increases with n and h. For $n = 80$, about 75% of successful experiments succeed by epoch 4. 8 epochs are needed for 75% of experiments to succeed for $n = 150$, and 13 epochs for $n = 200, 256, 300$.

For a given dimension, the number of epochs required for secret recovery varies a lot from one experiment to another. For dimension 256 and Hamming weight 31, different secrets need between 6 and 27 epochs. For dimension 350, a secret with Hamming weight 59 is recovered after 18 epochs, while secrets with $h = 58$ and 60 need 39 and 38.

We believe that, for a given secret $\mathbf{s}$, certain distributions of the coordinates of $\mathbf{a}$ help the transformer learn $\mathbf{s}$. The proportion of such points $\mathbf{a}$ in the training sample varies for different secrets, making some harder to recover, and necessitating longer training.

Another explanation for the variations in training length is the random initialization of transformer parameters, discussed in § 6.4. For a given secret, running several experiments, with different initializations, may reduce the number of epochs required for recovery.

Required LWE sample size. PICANTE relies on the TINYLWE subsampling technique introduced in §4.1 to recover secrets from only $4n$ initial LWE samples. By comparison, SALSA used 4 million LWE samples. Table 4 compares the performance of PICANTE (using TINYLWE), with an equivalent attack using 2.2 million collected

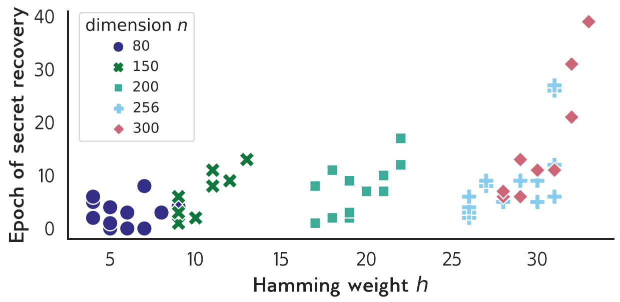

Figure 3: Number of training epochs before secret recovery for $80 \leq n \leq 300$**. For different dimensions and Hamming weights. We omit** $n = 350$ **results for space reasons.**

LWE samples, which we call LWE. For both sampling approaches —TinyLWE and LWE—we run the reduction step described in §4.1 (Stage 1.3) before performing model training and secret recovery.

Dimension	80	150	200	256	300
TinyLWE max h	9	13	22	31	33
LWE max h	9	12	21	32	32

Table 4: TinyLWE vs LWE. Values: highest h **recovered for each** n**.**

There is little difference between the highest Hamming weight of recovered secrets for TinyLWE and LWE (Table 4). In fact, TinyLWE sometimes recovers larger Hamming weights than LWE. Thus, we conclude that TinyLWE, while greatly reducing the LWE samples needed for the attack, has no impact on performance. More detailed comparisons can be found in Table 16 in Appendix A.1.

5.3 Resources needed for Picante

The total cost of Picante is the sum of the resources needed to preprocess data, train the model, and recover the secret.

n $\log_2 q$	80 7	150 13	200 17	256 23	300 27	350 32
Cost per matrix (CPU.hrs)	0.01	3	16	52	106	194
Matrices needed	34,800	14,600	10,800	8,300	7,100	6,000

Table 5: Resources needed for preprocessing. Total resources needed to produce 2^{22} **reduced samples is the work required for processing** $2^{21}/n$ **matrices. Processing can be fully parallelized, so total time required is the number of CPU hours needed for one matrix.**

Data preprocessing is the most resource intensive part of Picante. To generate 2^{22} reduced samples, $2^{21}/n$ matrices must be reduced (one $n \times n$ matrix produces $2n$ reduced samples, see § 4.1). As the dimension increases, the number of matrices needed scales down linearly. To avoid the exponential cost of BKZ-reduction [45], we fix the block size to at most $\beta = 20$ so that the preprocessing step scales as a polynomial in n and $\log q$. In practice, to save resources, we choose smaller β for larger dimensions. Parameter choices for preprocessing are discussed further in §6.1 below.

Table 5 reports the preprocessing resources (in cpu·hours) required for each n. It is important to note that our preprocessing step is fully parallelizable. Using as many CPUs as the number of matrices needed ($2^{21}/n$), the preprocessing step can be performed in the time required to reduce one matrix (e.g. 194 hours for $n = 350$).

Model training and secret recovery. The cost of training and recovery is proportional to the number of training epochs needed to recover the secret. Table 6 reports the average duration of one training epoch and associated secret recovery. All models use the same number of parameters, batch size (128) and epoch size (2 million examples), and are trained on one NVIDIA V100 GPU.

Training time increases with dimension. This is expected, as the length of input sequences is $2n$, i.e. linear in the dimension, and training is slower on long sequences. For secret recovery, the time required for each method is dominated by the number of transformer inferences needed, multiplied by the time required for each inference. The cross-attention method uses a constant number of inferences, direct recovery uses $15n$ inferences, and distinguisher recovery $200n$. Like training, the time for a single inference scales linearly with n because of increasing sequence length. Overall, cross-attention recovery scales linearly with n, and direct and distinguisher scale quadratically. In our experiments, secret recovery accounts for less than 10% of the total time. We report the cost of training and recovery per epoch on one GPU, but both training and recovery time could be significantly reduced by parallelizing across many GPUs.

n $\log q$	80 7	150 13	200 17	256 23	300 27	350 32
Training	42	52	68	82	92	105
Secret Recovery	1	2	3	5	7	8

Table 6: Training and recovery time per epoch (minutes). All models are trained on a single NVIDIA V100 GPU with batch size 128).

6 ADDITIONAL RESULTS

Now, we consider the effect of different experimental choices on Picante's performance. This enables us to better understand the conditions under which Picante succeeds or fails.

6.1 Data preprocessing

Through extensive experimentation, we observed that data preprocessing is critical to enabling the transformer to learn and allowing recovery of secrets with larger Hamming weight. Preprocessing changes the distribution of both the size of the entries of $\mathbf{A}$ modulo q (shown for $n = 150$ in Figure 4) and the norm of its rows (see Figure 5). Note that the goal of preprocessing is **not** to obtain the shortest vector in the lattice like the classical uSVP, decoding, and dual attacks. Rather, its goal is to skew the distribution to make it more amenable to machine learning-based attacks.

Choosing preprocessing parameters. To determine the amount of preprocessing needed for optimal Picante performance, we set various targets for the standard deviation of the entries of the rows of $\mathbf{A}$. For random matrices $\mathbf{A}_{rand}$, the standard deviation of the entries is $\text{std}(\mathbf{A}_{rand}) \approx q/\sqrt{12} \approx 0.29q$. After preprocessing, the standard deviation of the entries are smaller, e.g., for $n = 256$, $\text{std}(\mathbf{A}) \approx 0.1q$. Empirically, we observed that reducing the ratio

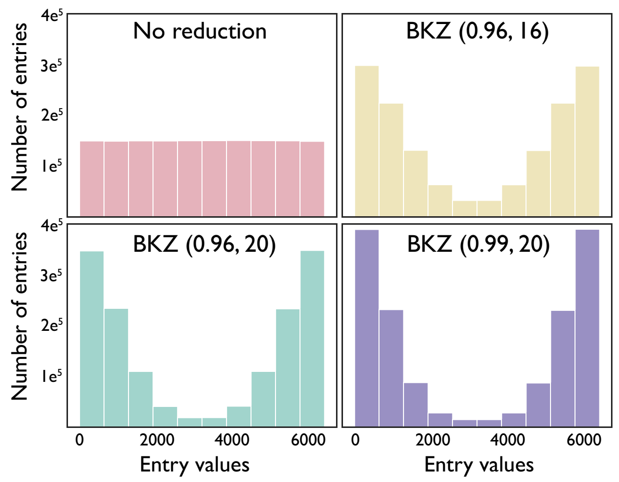

Figure 4: Distribution of sample entry values as strength of norm reduction increases ($n = 150$, $q = 6421$). BKZ parameters: BKZ (β, δ).

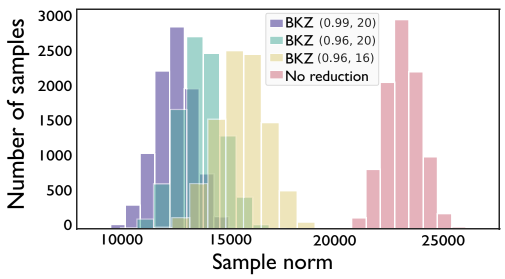

Figure 5: Distribution of sample norms as strength of norm reduction increases ($n = 150$). BKZ parameters listed as BKZ (β, δ).

std($\mathbf{A}$)/std($\mathbf{A}_{rand}$) allows us to recover secrets with higher hamming weights. In practice, we select parameters for our BKZ preprocessing step in *fplll* by first processing a single matrix $\mathbf{A}$ with various choices for β and δ and observing the resulting standard deviation. To prepare a whole dataset for training, we preprocess the $2^{21}/n$ matrices with parameters that reach a low standard deviation in a reasonable time; when we did not recover secrets with the target $\approx 10\%$ density for $n = 350$, we repeated the data generation with stronger parameters.

Relationship between preprocessing and recovered h. To examine how preprocessing parameters affect Picante's secret recovery, we ran four sets of experiments with values of β and δ ranging from no preprocessing at all to the parameters used in Picante. Table 7 shows statistics of the data distribution and the largest h recovered for these experiments. As preprocessing strength increases, for dimension $n = 150$ (middle columns), the weight h of recovered secrets increases up to 12. As shown in the last two columns of Table 7, for $n = 350$, stronger preprocessing parameters decrease the standard deviation of the entries and enable recovery of larger weight h secrets (up to $h = 60$).

Relationship between preprocessing and n. Table 8 presents the highest Hamming weight recovered and standard deviations of reduced $\mathbf{A}$ for different n. We observe that, even though we decreased the block size and δ to reduce preprocessing time for larger dimensions ($n = 256, 300$ and 350), as long as the standard deviation of the entries is low enough, Picante recovers secrets with $> 10\%$ sparsity.

$n, \log_2 q$	150, 13				350, 32	
δ	-	0.96	0.96	0.99	0.93	0.96
β	-	16	20	20	14	14
highest h	-	**5**	**8**	**12**	**25**	**60**
std($\mathbf{A}$)/std($\mathbf{A}_{rand}$)	1	0.667	0.578	0.526	0.331	0.253
norm($\mathbf{A}$)/norm($\mathbf{A}_{rand}$)	1	0.669	0.581	0.528	0.332	0.253
cost / matrix (hours)	0	0.5	0.9	3.1	152	194
time out (hours)	-	1	2	5.5	-	-

Table 7: Highest weight h secret recovered for varying δ and/or β ($n = 150, 350$). std(A): standard deviation of A's coefficients post-reduction; norm(A): average norm of A's rows post-reduction.

n	$\log_2 q$	δ	β	std($\mathbf{A}$)/std($\mathbf{A}_{rand}$)	max h
80	7	0.99	20	0.78	9
150	13	0.99	20	0.53	13
200	17	0.99	20	0.40	22
256	23	0.96	18	0.33	31
300	27	0.96	16	0.32	33
350	32	0.96	14	0.25	60

Table 8: Preprocessing parameters for BKZ in *fplll* for Picante's best secret recoveries. β : block size, δ : LLL_DELTA.

base	$h = 9$	$h = 10$	$h = 11$	$h = 12$
$81 \approx \sqrt{q}$	1/5	0/5	0/5	0/5
$402 \approx q/16$	3/5	2/5	0/5	0/5
$803 \approx q/8$	3/5	2/5	2/5	2/5
$1071 \approx q/6$	**4/5**	**2/5**	**3/5**	**1/5**
$1606 \approx q/4$	3/5	2/5	0/5	0/5

Table 9: Secret recovery rate for different bases B. $n = 150$, $q = 6421$. Picante parameters and results are in bold.

6.2 Encoding base

We explore how B, the base used to encode b and the coordinates of $\mathbf{a}$ (§4.2) during model training, affects Picante performance. Table 9 presents the impact of different base choices on secret recovery for $n = 150$, $q = 6421$. For a small modulus like this, no buckets are needed, i.e. $r = 1$. To keep input sequences short, all integers modulo q should be encoded in two tokens, i.e. $B \geq \sqrt{q}$. However, values of B close to $\sqrt{q}$ result in worse secret recovery. Recovery rates are highest when $B = \lfloor q/k \rfloor$ with $k = 6$ or 8.

Table 10 presents a similar study of base B and bucket size r for larger $n = 256$ and $q = 6139999$. When q is this large, using base $B = \lfloor q/k \rfloor$ would result in a vocabulary that is too large for the transformer to learn efficiently. Hence, we tokenize the less significant digits in buckets of size r, as described in §4.2. Small values of $B \approx \sqrt{q}$ and large values $B = q/4$ do not seem to result in good secret recovery. For $B = q/8$ and $q/16$, bucket sizes $r = 128$ and $r = 512$ have comparable performance.

6.3 Model architecture

All Picante experiments use the same model architecture (see § 4.2 for details). However, Salsa reported improved performance for

Encoding base B	r	Hamming weight h: 26	27	28	29
$2478 \approx \sqrt{q}$	1	0/5	0/5	0/5	0/5
$383750 \approx q/16$	32	4/5	1/5	1/5	0/5
	128	3/5	2/5	1/5	3/5
	512	4/5	2/5	1/5	3/5
$767500 \approx q/8$	32	4/5	1/5	1/5	1/5
	128	**4/5**	**1/5**	**1/5**	**3/5**
	512	4/5	2/5	1/5	3/5
$1535000 \approx q/4$	32	0/5	0/5	0/5	0/5
	128	0/5	0/5	0/5	0/5
	512	1/5	0/5	0/5	0/5

Table 10: Secret recovery rates for different bases B and bucket sizes r. $n = 256$, $q = 6139999$. Picante parameters are in bold.

larger n with larger models, specifically increased embedding dimensions. Also, the number of attention heads used in Picante, 4 in the encoder and decoder, is low, compared to common transformer architectures. Most transformers with 512 dimensions use 8 heads. Thus, we explore the impact of larger dimensions and number of heads in the encoder and decoder, on secret recovery (Table 11) for $n = 350$. As the table shows, increasing dimension and heads do *not* result in better performance. This contrasts with results in NLP, where performance usually increases with model size. We believe this is because the LWE problem differs from traditional NLP tasks.

embedding size encoder/decoder	number of attention heads encoder/decoder/cross attention: **4/4/4**	4/4/8	4/4/16	8/8/8	8/8/16
1024 / 512	**60**	58	58	-	-
1024 / 768	-	58	60	57	55
1280 / 512	-	60	58	58	58

Table 11: Effect of architecture on Picante's performance. Data shown is the highest Hamming weight recovered for $n = 350$. Picante's parameters are in bold.

6.4 Model initialization

Transformer parameters are randomly initialized before training, and these initial values may impact the performance. This is known as the "lottery ticket" phenomenon: models sometimes learn better, or faster, with different initial parameter values. We explore this effect in 4 experiments for $n = 200$ and $h = 19$. Each experiment in Table 12 has a different secret; for each secret we train 20 transformers, each initialized with a different seed. In experiment 1, the secret is recovered for all 20 seeds, at epoch 2 to 7. For experiments 2 and 3, the secret is recovered about 3/4 of the time, between 6 and 25 epochs. In experiment 4, the secret is never recovered.

This sheds light on results from §5.2. There, we observed significant variance in the number of epochs needed for secret recovery, for given n and h. Initialization seems to be an important factor and suggests a possible improvement to Picante when significant compute resources are available. By training several transformers with different initializations on the same data, Picante's chances of secret recovery improve, as does training speed—training can stop for all models once one recovers the secret.

Experiment	Success	Mean epoch	Min, max epochs
1	20/20	4.2	2, 7
2	12/20	12.1	8, 25
3	15/20	9.1	6, 16
4	0/20	-	-

Table 12: Effect of model initialization on secret recovery. $n = 200$, $h = 19$.

6.5 Secret recovery methods

Picante leverages 4 secret recovery methods (§ 4.3): distinguisher, direct, cross-attention, and combined. The first three methods output *bit scores* and secret guesses $\mathbf{s'}$. The *bit scores* rank the likelihood of individual secret bits having value 1. The *combined secret recovery* method allows Picante to create additional secret guesses by aggregating the scores of the previous methods. Table 13 reports the successes/attempts of all secret recovery methods for fixed n/q and varying h. We only report the method(s) that succeed first: we terminate each experiment after successful recovery. We say that the *combined* method is successful if and only if it recovered the secret when no individual method could. If an individual method succeeds, the combined method typically succeeds as well.

Two trends are evident in Table 13. First, the direct recovery method is outperformed by other methods as n increases. It works well at $n = 80$, but for $n \geq 256$, it is either slower than other methods or fails to recover the secret. Recall that direct recovery works when for every bit i of the secret, the model prediction $\mathcal{M}(K \cdot \mathbf{e}_i)$ corresponds to the secret bit: large when the secret bit is 1 and small otherwise. This happens with lower probability as n grows. Second, the combined recovery method performs better as n increases. Probably for larger n, individual methods cannot glean information about all secret bits, but each gains some information about some bits. Thus, combining their scores may allow additional recoveries.

7 COMPARISON TO EXISTING LWE ATTACKS

Finally, we compare Picante's performance against classical lattice attacks. This is a difficult task, given both the significant differences in methodology between Picante and existing attacks, as well as the lack of reported concrete running times for existing attacks. In practice, the training stage of Picante takes less time than preprocessing the data (see Table 5 and Table 6 in § 5.3), so we focus on comparing the cost of preprocessing with the cost of *classical lattice reduction attacks* such as uSVP, decoding, and dual attacks. As Picante uses the *fplll* package for lattice reduction algorithms, we compare the running times of Picante with the uSVP attack, run using *fplll*.

The LWE Estimator software package [5] is used to estimate the cost of classical lattice reduction attacks. The LWE Estimator uses theoretical formulas and heuristic estimates to predict which block size will be required for BKZ to recover the secret for a given lattice parameter size. These estimates are widely used to set parameters and estimate security at parameter sizes for which it is impossible to actually run these classical attacks (they would not terminate in our lifetimes). Concrete running times for actual successful attacks can be found in a few places in the literature, e.g. in [4, 8, 12, 28], and we find those useful for comparison here. In particular, for dimensions $n \leq 200$, we chose values of $\log q$ strictly smaller than

$n, \log_2 q$	Hamming weight h						
80, 7	4	5	6	7	8	9	10
success	3/5	3/5	2/5	2/5	1/20	1/20	0/20
Distinguisher	3/5	3/5	2/5	2/5	1/20	1/20	0/20
Direct	2/5	1/5	1/5	1/5	1/20	1/20	0/20
Cross-attention	3/5	1/5	2/5	0/5	1/20	0/20	0/20
Combined	0/5	0/5	0/5	0/5	0/20	0/20	0/20
150, 13	9	10	11	12	13	14	15
success	4/5	2/5	3/5	1/5	1/20	0/20	0/20
Distinguisher	2/5	1/5	2/5	0/5	0/20	0/20	0/20
Direct	2/5	0/5	0/5	0/5	1/20	0/20	0/20
Cross-attention	1/5	0/5	1/5	1/5	0/20	0/20	0/20
Combined	0/5	1/5	0/5	0/5	0/20	0/20	0/20
200,17	17	18	19	20	21	22	23
success	3/5	2/5	3/5	1/5	2/5	2/20	0/20
Distinguisher	2/5	1/5	2/5	1/5	0/5	0/20	0/20
Direct	0/5	0/5	1/5	0/5	0/5	0/20	0/20
Cross-attention	1/5	0/5	0/5	0/5	0/5	1/20	0/20
Combined	0/5	1/5	1/5	0/5	2/5	1/20	0/20
256, 23	26	27	28	29	30	31	32
success	4/5	1/5	1/5	3/5	3/5	4/20	0/20
Distinguisher	3/5	1/5	1/5	1/5	1/5	0/20	0/20
Direct	0/5	0/5	0/5	0/5	0/5	0/20	0/20
Cross-attention	2/5	0/5	1/5	3/5	2/5	2/20	0/20
Combined	1/5	0/5	0/5	0/5	0/5	2/20	0/20
300, 27	28	29	30	31	32	33	34
success	2/5	2/5	1/5	1/5	2/5	1/5	0/20
Distinguisher	2/5	0/5	0/5	0/5	0/5	0/5	0/20
Direct	0/5	0/5	0/5	0/5	0/5	0/5	0/20
Cross-attention	1/5	2/5	1/5	0/5	1/5	1/5	0/20
Combined	0/5	0/5	0/5	1/5	1/5	0/5	0/20
350, 32	54	55	56	57	58	59	60
success	2/5	1/5	1/5	1/5	1/5	1/5	1/5
Distinguisher	0/5	0/5	0/5	0/5	1/5	0/5	0/5
Direct	0/5	0/5	0/5	0/5	0/5	0/5	0/5
Cross-attention	0/5	1/5	0/5	0/5	0/5	0/5	1/5
Combined	2/5	0/5	1/5	1/5	0/5	1/5	0/5

Table 13: Secret recovery successes/attempts for Picante's four recovery methods. For each (n, q, h) setting, we report the number of secrets Picante recovers out of attempted attacks ("success" row) and the number of recoveries by each individual method ("Distinguisher" through "Combined" rows). If two methods succeed in the same training epoch, we report both successes, so individual recoveries may exceed the number of total successes. Combined method successes are only reported when all other methods fail.

those used in [12]; for dimensions $n = 256, 300$ and 350, we use much smaller values of $\log q$ than [28]: for instance, for $n = 350$, we use $\log q = 32$, much smaller than the value $\log q = 52$ in [28].

We present here 3 ways to quantify, estimate, and compare with pure lattice reduction attacks: the LWE estimator, concrete timings for running the uSVP attack at small sizes, and theoretical and heuristic formulas.

n	q	best attack	cost	block size
80	113	BDD	$2^{48.0}$	$\beta = 63$
150	6421	BDD	$2^{42.7}$	$\beta = 44$
200	130769	BDD	$2^{41.8}$	$\beta = 41$
256	6139999	uSVP/BDD	$2^{41.8}$	$\beta = 40$
300	94056013	uSVP	$2^{41.9}$	$\beta = 40$
350	3831165139	uSVP	$2^{42.0}$	$\beta = 40$

Table 14: LWE Estimator [5] estimates for Picante's most successful recoveries (see Table 1). Cost: number of operations in $\mathbb{Z}_q$.

LWE Estimator. Table 14 presents the block size and estimated cost for classical attacks, to compare against Picante's successful secret recoveries (using the highest h achieved by Picante). LWE Estimator [5] costs are listed in terms of the number of operations in $\mathbb{Z}_q$, the cost of which can be approximated by $(\log q)^2$. For example, for $n = 256$, the LWE estimator predicts that the uSVP attack should succeed with block size 40 and cost about $2^{41.8}$ operations in $\mathbb{Z}_q$. Picante uses block size 18.

Concrete running times. Table 15 presents concrete running times for the following attack: We run the primal uSVP attack, using Kannan's embedding and BKZ2.0 [14] with different block sizes. The dimension for the Kannan's embedding is determined as in [12]. We choose block sizes close to the block size predicted by the LWE Estimator and compare Picante against attacks with similar success probability. We ran the classical attacks for dimensions up to $n = 256$ with the block size predicted by the LWE Estimator ($\beta = 40$). For $n = 300, 350$, already the first loop in BKZ takes longer than 3 days; the full attack was taking too long to run.

The uSVP attack was run using the *fplll* package on the same machine as the norm-reduction step of Picante. We did not use any optimization for either of the attacks. We see that for $n = 80$ and $h = 9$, Picante with $\beta = 20$ achieves similar success to uSVP with $\beta = 60$. The uSVP attack takes about 10 hours to succeed; for this $n = 80$, $h = 9$ setting, the time spent on data preprocessing for Picante is negligible with enough parallelization and the training (run on 1 GPU) for successful recoveries took 5 epochs of about 0.7 hours each. The time spent by Picante is therefore about 4 hours. For $n = 256$ and $h = 31$, the uSVP attack with block size $\beta = 35$ (smaller than predicted by the LWE estimator) took a minimum of 231 hours to succeed; Picante with sufficient parallelization needs 52 hours for data pre-processing and a minimum of 10 hours for training, so the total time is about 62 hours.

These timings are rough estimates. Optimizations to lattice-reduction for the uSVP attack could also speed up the data pre-processing of Picante. We did not include any possible savings from parallelizing training and secret recovery methods (see §5.3).

Theoretical analysis. Denote by $BKZ(d, \beta)$ the (classical) cost of BKZ reduction in dimension d with block size β. It can be estimated [2] as $2^{0.292\beta+c} << BKZ(d, \beta) < 8d \cdot 2^{0.292\beta+c}$, where the cost $SVP(\beta) = 2^{0.292\beta+c}$ is the cost of the SVP oracle in dimension β (a major step in the BKZ-reduction algorithm). The constant c depends on the attack model—16.4 for sieving and 0 for others. The upper bound arises from the estimated 8 runs (full loops) of the BKZ-reduction, and hence $8d\ SVP(\beta)$ oracle calls, needed.

The uSVP attack solves the shortest vector problem in dimension $d > n$; Picante applies the BKZ-reduction to lattices of dimension $d = 2n$ but keeps the block size close to constant ($\beta = 20$, and

$n, \log_2 q$	h	PICANTE					uSVP attack with BKZ 2.0 and early-abort			
				Preprocessing		*Training*				
		β	success	CPU.hrs per matrix	# matrices	CPU.hrs per epoch	β	success	success time (CPU.hrs)	fail time (CPU.hrs)
80, 7	6, 7	20	4/10	0.01	34800	0.7	60	2/10	8, 12	7.8
							65	6/10	9, 10, 14, 18, 40, 85	72.1
	8, 9	20	2/40	0.01	34800	0.7	55	0/10	—	2.4
							60	1/10	12	6.9
150, 13	9,10	20	6/10	3.1	14600	0.9	50	5/10	26, 30, 31, 35, 35	22.9
							55	8/10	19, 19, 23, 23, 23, 23, 28, 28	22.7
	11, 12	20	4/10	3.1	14600	0.9	50	2/10	22, 39	9.5
							55	4/10	14, 19, 24, 33	5.6
200, 17	18, 19	20	5/10	16	10800	1.2	45	6/10	12, 13, 18, 21, 21, 21	7.7
	20, 21	20	3/10	16	10800	1.2	45	3/10	13, 21, 25	12.9
256, 23	26, 27	18	5/10	52	8300	1.5	40	4/10	203, 221, 243, 265	189.1
	28, 29	18	4/10	52	8300	1.5	35	7/10	238, 246, 249, 269, 284, 303, 348	241.9
	30, 31	18	4/10	52	8300	1.5	35	5/10	231, 255, 263, 330, 336	171.7

Table 15: Concrete running times (CPU.hrs) for uSVP attacks and corresponding PICANTE costs. Training cost includes secret recovery time. PICANTE uses BKZ, the uSVP attack uses BKZ2.0 [14], see the discussion in § 4.1. In all uSVP attacks, we use $\omega = round(\sqrt{2}\sigma) = 4$. Legend: CPU.hrs: CPU hours. fail time: average time for the failed experiments.

decreases the block size in larger dimensions for efficiency). We choose this because the cost of BKZ reduction scales exponentially with the block size. While we do not know if we can use constant block size $\beta = 20$ for all dimensions, we expect our block size to grow slower than block sizes required for lattice-reduction attacks.

High level comparison. PICANTE compares with classical lattice reduction attacks as follows: PICANTE succeeds in recovering the secret vector *using much smaller block size than pure lattice reduction attacks, at the expense of processing many more matrices (2.2 million/n matrices).* Because this step is run in parallel, PICANTE recovers secrets faster than the uSVP attack but uses many more CPUs for parallel processing. As the dimension increases and/or $\log q$ decreases, we expect the advantage of PICANTE to grow, due to the exponential cost of the lattice reduction attacks based on BKZ. Future work may produce a more efficient way to preprocess the data or reduce the amount of data needed for training.

8 DISCUSSION

Our attack, PICANTE, demonstrates a dramatic improvement over SALSA, the only prior work on attacking LWE with Machine Learning. SALSA pioneered the use of ML models in cryptanalysis of LWE, but only recovered secrets for small LWE problems. In contrast, PICANTE successfully recovers LWE binary secrets with sparsity up to 10%, for dimensions up to 350. It does so using only $4n$ LWE samples, a realistic assumption in practice. PICANTE's performance is competitive with that of known state-of-the-art attacks on LWE, particularly when sufficient compute resources are available, as PICANTE's novel data preprocessing step can be parallellized.

Mastermind. One way to think about the role of our novel preprocessing step is in analogy with the game Mastermind. In Mastermind, a secret made up of 4 pegs of 6 possible colors is hidden from the guesser. The guesser makes queries of 4 pegs of different colors, and query responses indicate how many pegs matched the color and/or position of secret pegs. Binary secret LWE can be thought of as Mastermind with n positions and 2 colors, ignoring error.

In PICANTE, the trained model serves as an engine for answering queries about the secret. Consider two extreme types of queries. If you submit a vector with all entries constant, $(f, f, ..., f)$, (i.e. very low entropy), you only get the Hamming weight—no information about the position of the 1s. On the other hand, the Direct secret recovery approach makes queries of the form $(0, ..., 0, K_i, 0, ..., 0)$, which gives information only about the i^{th} bit of the secret. Submitting queries with random entries (maximal entropy) does not clearly give any particular type of information.

PICANTE's preprocessing step reduces the entropy of LWE samples, making it more likely that queries such as those in the Direct secret recovery method bear some similarity to the training samples. So in some sense our approach is ML for Mastermind (or Wordle).

Scaling to larger Hamming weights. SALSA only recovered secrets with Hamming weights $h = 3$ or 4. PICANTE recovers larger h (up to 31 for $n = 256$ and 60 for $n = 350$), but recovering even larger h (general binary secrets) is an important challenge for future work. As it stands, the PICANTE attack can be countered by using general binary secrets. One way to scale to larger h is to improve the preprocessing step. The more the variance of the training set's coordinates is reduced, the higher h secrets PICANTE recovers.

Our intuition regarding the relationship between preprocessing and recoverable h is as follows (see [31] for a detailed analysis). Given a secret $\mathbf{s}$ with Hamming weight h and dimension n, and a vector $\mathbf{a}$ with coordinates uniformly sampled over $(-q/2, q/2)$, the dot product $\mathbf{a} \cdot \mathbf{s}$ is a sum of h uniform random variables with mean $\mu = 0$ and standard deviation $\sigma = q/\sqrt{12}$. Thus, as h grows, the distribution of $\mathbf{a} \cdot \mathbf{s}$ is roughly normal with $\mu = 0$ and $\sigma = nq\sqrt{h/12}$. Thus, in 68% of cases, the value of $\mathbf{a} \cdot \mathbf{s}$ will remain within one σ of $\mu = 0$, i.e. span a range of $2q\sqrt{h/12} = q\sqrt{h/3}$. If $h = 3$, this range is q: $\mathbf{a} \cdot \mathbf{s}$ spans only one period of the modulus. This explains why SALSA has difficulty recovering secrets with $h \geq 4$: the model must learn modulus wrapping. Let α be the reduction factor $\mathbf{std}(\mathbf{A})/\mathrm{std}(\mathbf{A}_{rand})$ achieved via preprocessing (see Section 6.1 and Table 7). Then $\mathbf{a} \cdot \mathbf{s}$ is a random variable with mean μ=0, and $\sigma = \alpha q \sqrt{h/12}$. If $h < 3/\alpha^2$, then $\mathbf{a} \cdot \mathbf{s}$ will span only one modulus, enabling easier learning. This suggests that larger h may be recovered by improving preprocessing.

Ethical considerations. Although PICANTE demonstrates significant progress towards attacking real-world LWE problems with sparse binary secrets, it cannot yet break problems with real-world-size parameters. In particular, the LWE schemes standardized by NIST use smaller modulus q and non-sparse secret distributions. Hence, we do not believe our paper raises any ethical concerns. Nonetheless, we shared a copy of the current paper with the NIST Cryptography group, to inform them of our approach.

Future directions. More work is needed to better understand the effect of the data preprocessing step, since we observe that we only need a 5% reduction of data entropy to succeed (Table 7). Additionally, there may be better ways to preprocess the data to improve transformer learning, which are less costly than using BKZ. In the future, the model training and secret recovery components of the attack could benefit from parallel runs across multiple GPUs, given our observation that different transformer initializations may result in different speeds of secret recovery (§ 6.4). Furthermore, improvements to transformer architecture and secret recovery methods may enable recovery of secrets with more complex parameter settings. In particular, future work could explore the use of simpler model architectures to reduce memory and time costs. For example, prior work shows RNNs and LSTMs can perform modular addition, but [34] suggests that this is difficult for FFNs. Finally, cross-attention secret recovery suggests that useful information can be gleaned from inspecting models' intermediate representations. Better understanding of these would be interesting future work.

ACKNOWLEDGEMENTS

We thank Mark Tygert and Matteo Pirotta for many helpful discussions, and the anonymous reviewers for suggestions and edits.

REFERENCES

[1] AKKAYA, I., ANDRYCHOWICZ, M., CHOCIEJ, M., ET AL. Solving rubik's cube with a robot hand, 2019. https://arxiv.org/abs/1910.07113.

[2] ALBRECHT, M., CHASE, M., CHEN, H., ET AL. Homomorphic encryption standard. In *Protecting Privacy through Homomorphic Encryption*. 2021, pp. 31–62. https://eprint.iacr.org/2019/939.

[3] ALBRECHT, M. R. On Dual Lattice Attacks Against Small-Secret LWE and Parameter Choices in HElib and SEAL. In *Proc. of EUROCRYPT* (2017).

[4] ALBRECHT, M. R., GÖPFERT, F., VIRDIA, F., AND WUNDERER, T. Revisiting the expected cost of solving usvp and applications to lwe. In *Proc. of ASIACRYPT* (2017).

[5] ALBRECHT, M. R., PLAYER, R., AND SCOTT, S. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology 9*, 3 (2015), 169–203.

[6] AVANZI, R., BOS, J., DUCAS, L., KILTZ, E., LEPOINT, T., LYUBASHEVSKY, V., SCHANCK, J. M., SCHWABE, P., SEILER, G., AND STEHLÉ, D. CRYSTALS-Kyber (version 3.02) – Submission to round 3 of the NIST post-quantum project. Available at https://pq-crystals.org/.

[7] BAHDANAU, D., CHO, K., AND BENGIO, Y. Neural machine translation by jointly learning to align and translate. In *Proc. of ICLR* (2014).

[8] BAI, S., MILLER, S., AND WEN, W. A refined analysis of the cost for solving LWE via uSVP. In *Proc. of ASIACRYPT* (2019).

[9] BRAKERSKI, Z., LANGLOIS, A., PEIKERT, C., REGEV, O., AND STEHLÉ, D. Classical Hardness of Learning with Errors. In *Proc. of the ACM Symposium on Theory of Computing* (2013).

[10] CARION, N., MASSA, F., SYNNAEVE, G., ET AL. End-to-end object detection with transformers, 2020. https://arxiv.org/abs/2005.12872.

[11] CHARTON, F. Linear algebra with transformers, 2021. https://arxiv.org/abs/2112.01898.

[12] CHEN, H., CHUA, L., LAUTER, K., AND SONG, Y. On the Concrete Security of LWE with Small Secret. Cryptology ePrint Archive, Paper 2020/539, 2020. https://eprint.iacr.org/2020/539.

[13] CHEN, L., MOODY, D., LIU, Y.-K., ET AL. PQC Standardization Process: Announcing Four Candidates to be Standardized, Plus Fourth Round Candidates. *US Department of Commerce, NIST* (2022). https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4.

[14] CHEN, Y., AND NGUYEN, P. Q. BKZ 2.0: Better Lattice Security Estimates. In *Proc. of ASIACRYPT* (2011).

[15] CHEON, J. H., HHAN, M., HONG, S., AND SON, Y. A Hybrid of Dual and Meet-in-the-Middle Attack on Sparse and Ternary Secret LWE. *IEEE Access* (2019).

[16] CHEON, J. H., KIM, A., KIM, M., AND SONG, Y. Homomorphic encryption for arithmetic of approximate numbers. In *Proc. of ASIACRYPT* (2017).

[17] CHEON, J. H., KIM, D., LEE, J., AND SONG, Y. Lizard: Cut Off the Tail! A Practical Post-quantum Public-Key Encryption from LWE and LWR. In *Security and Cryptography for Networks* (2018).

[18] CHO, K., VAN MERRIENBOER, B., GULCEHRE, C., ET AL. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *Proc. of EMNLP* (2014).

[19] CSORDÁS, R., IRIE, K., AND SCHMIDHUBER, J. The Neural Data Router: Adaptive Control Flow in Transformers Improves Systematic Generalization. In *Proc. of ICML* (2022).

[20] CURTIS, B. R., AND PLAYER, R. On the feasibility and impact of standardising sparse-secret LWE parameter sets for homomorphic encryption. In *Proc. of the ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (2019).

[21] DEHGHANI, M., GOUWS, S., VINYALS, O., USZKOREIT, J., AND KAISER, Ł. Universal transformers. In *Proc. of ICLR* (2019).

[22] DEVELOPMENT TEAM, T. F. fplll, a lattice reduction library, Version: 5.4.4. Available at https://github.com/fplll/fplll, 2023.

[23] DONG, L., XU, S., AND XU, B. Speech-transformer: A no-recurrence sequence-to-sequence model for speech recognition. In *Proc. of ICASSP* (2018).

[24] DOSOVITSKIY, A., BEYER, L., KOLESNIKOV, A., ET AL. An image is worth 16x16 words: Transformers for image recognition at scale. In *Proc. of ICLR* (2021).

[25] DUCAS, L., KILTZ, E., LEPOINT, T., LYUBASHEVSKY, V., SCHWABE, P., SEILER, G., AND STEHLÉ, D. CRYSTALS-Dilithium – Algorithm Specifications and Supporting Documentation (Version 3.1). Available at https://pq-crystals.org/.

[26] KANNAN, R. Minkowski's Convex Body Theorem and Integer Programming. *Mathematics of Operations Research 12* (1987), 415–440.

[27] KINGMA, D. P., AND BA, J. Adam: A method for stochastic optimization. In *Proc. of ICLR* (2015).

[28] LAINE, K., AND LAUTER, K. Key recovery for lwe in polynomial time. *Cryptology ePrint Archive* (2015). https://eprint.iacr.org/2015/176.pdf.

[29] LAMPLE, G., AND CHARTON, F. Deep learning for symbolic mathematics. In *Proc. of ICLR* (2020).

[30] LENSTRA, H. J., LENSTRA, A., AND LOVÁSZ, L. Factoring polynomials with rational coefficients. *Mathematische Annalen 261* (1982), 515–534.

[31] LI, C., SOTAKOVA, J., WENGER, E., ALLEN-ZHU, Z., CHARTON, F., AND LAUTER, K. Salsa verde: a machine learning attack on learning with errors with sparse small secrets, 2023. https://arxiv.org/abs/2306.11641.

[32] LYUBASHEVSKY, V., AND MICCIANCIO, D. On bounded distance decoding, unique shortest vectors, and the minimum distance problem. In *Proc. of CRYPTO* (2009), S. Halevi, Ed.

[33] MICCIANCIO, D., AND VOULGARIS, P. Faster exponential time algorithms for the shortest vector problem. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms* (2010).

[34] PALAMAS, T. Investigating the ability of neural networks to learn simple modular arithmetic.

[35] PEIKERT, C. Public-Key Cryptosystems from the Worst-Case Shortest Vector Problem: Extended Abstract. In *Proc. of the ACM Symposium on Theory of Computing* (2009).

[36] POLU, S., AND SUTSKEVER, I. Generative language modeling for automated theorem proving, 2020. https://arxiv.org/abs/2009.03393.

[37] RADFORD, A., NARASIMHAN, K., SALIMANS, T., AND SUTSKEVER, I. Improving language understanding by generative pre-training. *OpenAI blog* (2018). https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf.

[38] RADFORD, A., WU, J., CHILD, R., LUAN, D., AMODEI, D., SUTSKEVER, I., ET AL. Language models are unsupervised multitask learners. *OpenAI blog* (2019). https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

[39] RAMESH, A., PAVLOV, M., GOH, G., GRAY, S., VOSS, C., RADFORD, A., CHEN, M., AND SUTSKEVER, I. Zero-shot text-to-image generation, 2021. https://arxiv.org/abs/2102.12092.

[40] Regev, O. Quantum computation and lattice problems. *SIAM Journal on Computing 33*, 3 (2004), 738–760.
[41] Regev, O. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. In *Proc. of the ACM Symposium on Theory of Computing* (2005).
[42] Rivest, R. L., Shamir, A., and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM* (1978).
[43] Rogez, G., Weinzaepfel, P., and Schmid, C. Lcr-net: Localization-classification-regression for human pose. In *Proc. of CVPR* (2017).
[44] Rothe, R., Timofte, R., and Van Gool, L. Dex: Deep expectation of apparent age from a single image. In *Proc. of ICCV* (2015).
[45] Schnorr, C. A hierarchy of polynomial time lattice basis reduction algorithms. *Theoretical Computer Science 53*, 2 (1987), 201–224.
[46] Schnorr, C. P., and Euchner, M. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming 66*, 1-3 (Aug. 1994), 181–199.
[47] Schrittwieser, J., Antonoglou, I., Hubert, T., et al. Mastering ATARI, Go, Chess and Shogi by planning with a learned model. *Nature 588* (2020), 604–609.
[48] Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL, Jan. 2023. Microsoft Research, Redmond, WA.
[49] Vaswani, A., Shazeer, N., Parmar, N., et al. Attention is all you need. In *Proc. of NeurIPS* (2017).
[50] Wang, Y., Mohamed, A., Le, D., et al. Transformer-based acoustic modeling for hybrid speech recognition. *Proc. of ICASSP* (2020).
[51] Wenger, E., Chen, M., Charton, F., and Lauter, K. Salsa: Attacking lattice cryptography with transformers. In *Proc. of NeurIPS* (2022).

A APPENDIX

A.1 Comparison of TinyLWE and LWE

Table 16 compares secret recovery performance of models trained using samples generated via Picante's TinyLWE approach ($4n$ initial samples) vs. a baseline approach (2^{22} initial samples).

Setting	Hamming weight h						
$n = 80$	4	5	6	7	8	9	10
TinyLWE	3/5	3/5	2/5	2/5	1/20	1/20	0/20
LWE	5/5	4/5	3/5	3/5	0/20	1/20	0/20
$n = 150$	9	10	11	12	13	14	15
TinyLWE	4/5	2/5	3/5	1/5	1/20	0/20	0/20
LWE	5/5	2/5	2/5	1/20	0/20	0/20	0/20
$n = 200$	17	18	19	20	21	22	23
TinyLWE	3/5	2/5	3/5	1/5	2/5	2/20	0/20
LWE	3/5	2/5	1/5	1/5	2/20	0/20	0/20
$n = 256$	26	27	28	29	30	31	32
TinyLWE	4/5	1/5	1/5	3/5	3/5	4/20	0/20
LWE	3/5	2/5	3/5	3/5	1/5	3/20	2/20
$n = 300$	28	29	30	31	32	33	34
TinyLWE	2/5	2/5	1/5	1/5	2/5	1/5	0/20
LWE	1/5	3/5	2/5	1/5	1/5	0/5	0/20

Table 16: Secret recovery performance: TinyLWE vs. LWE. Reported values are successes/attempts across different n/h settings.

A.2 Statistical properties of secret verification

At the end of the secret recovery phase, we are provided a secret guess $\mathbf{s}'$, that we need to check. To do so, we use the original $m = 4n$ LWE samples $(\mathbf{a}_i, b_i)$, and compute the m residuals $r_i = b_i - \mathbf{a}_i \cdot \mathbf{s}'$. If the secret is recovered, we expect the r_i to have the same standard deviation as a LWE sample, i.e. σ. Otherwise, we expect the standard deviation to be that of the uniform distribution, i.e. $q/\sqrt{12}$. The standard deviation of residuals is estimated by the formula:

$$\sigma_{\text{emp}} = \sqrt{\frac{1}{m-1}\sum_{i=0}^{m-1}(r_i - \overline{r})^2}$$

Lower and upper confidence intervals, with level $100(1-\alpha)\%$ are:

$$\sigma_{emp}\sqrt{\frac{m-1}{\chi^2_{\alpha/2,m-1}}}, \sigma_{emp}\sqrt{\frac{m-1}{\chi^2_{(1-\alpha)/2,m-1}}}$$

Since $m > 100$, we approximate the chi-square distribution with $m-1$ degrees of freedom by the normal distribution $\mathcal{N}(m-1, 2m-2)$. Table 17 provides estimates of the confidence intervals at level 0.001% for different values of n, and around $\sigma = 3$ and $q/\sqrt{12}$.

n	m	Right ($\sigma = 3$)	Wrong ($q/\sqrt{12}$)
80	320	[2.58, 3.72]	[28.08, 40.45]
150	600	[2.68, 3.48]	$[1.65 \times 10^3, 2.15 \times 10^3]$
200	800	[2.71, 3.40]	$[3.42 \times 10^4, 4.28 \times 10^4]$
256	1024	[2.74, 3.34]	$[1.62 \times 10^6, 1.98 \times 10^6]$
300	1200	[2.76, 3.31]	$[2.50 \times 10^7, 3.00 \times 10^7]$
350	1400	[2.78, 3.29]	$[1.02 \times 10^9, 1.21 \times 10^9]$

Table 17: Confidence intervals (0.001%) for secret verification. Confidence level 0.001%. Right: secret is correctly predicted ($\sigma = 3$). Wrong: secret is incorrectly predicted ($\sigma = q/\sqrt{12}$). q from Table 2.

For instance, for $n = 80$, we have $m = 320$ and $q = 113$. The 0.001% level confidence interval for a correct secret prediction (i.e. measuring $\sigma = 3$) is $[2.58, 3.72]$. For an incorrect prediction (measuring $\sigma = q/\sqrt{12} = 32.62$), it is $[28.08, 40.45]$. Since the two intervals do not overlap, the sample size we use (m) is large enough to verify secret guesses (with quasi-certitude). As dimension increases, the confidence intervals grow. This proves our claim that the original $4n$ LWE samples are sufficient to verify model predictions.

A.3 Understanding secret recovery

Figure 6 shows Picante's secret recovery for a successful $n = 350$ experiment, in which the combined method recovers the secret in epoch 5. Figure 6 shows how the rankings of the 1-bits of the secret change throughout training. Our recovery methods guess that the h top-ranked bits are the 1-bits, so successful recovery occurs when 1-bits occupy the first h slots on the x-axis.

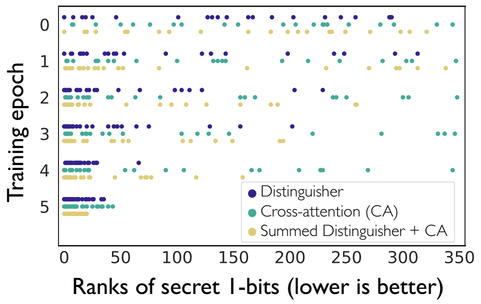

Figure 6: Change in secret bit ranks as training progresses ($n = 350$).

Over time, distinguisher and CA methods learn better ranks for true secret 1-bits. By epoch 5, the combined method, which sums the ranks of distinguisher and CA methods, correctly guesses the secret. We do not include direct secret recovery results because it performs poorly for large n. Plotting Figure 6 requires knowledge of the secret s, leveraged here *for illustrative purposes only*. Picante can validate secret guesses without knowledge of s, using verification as in § 4.3.