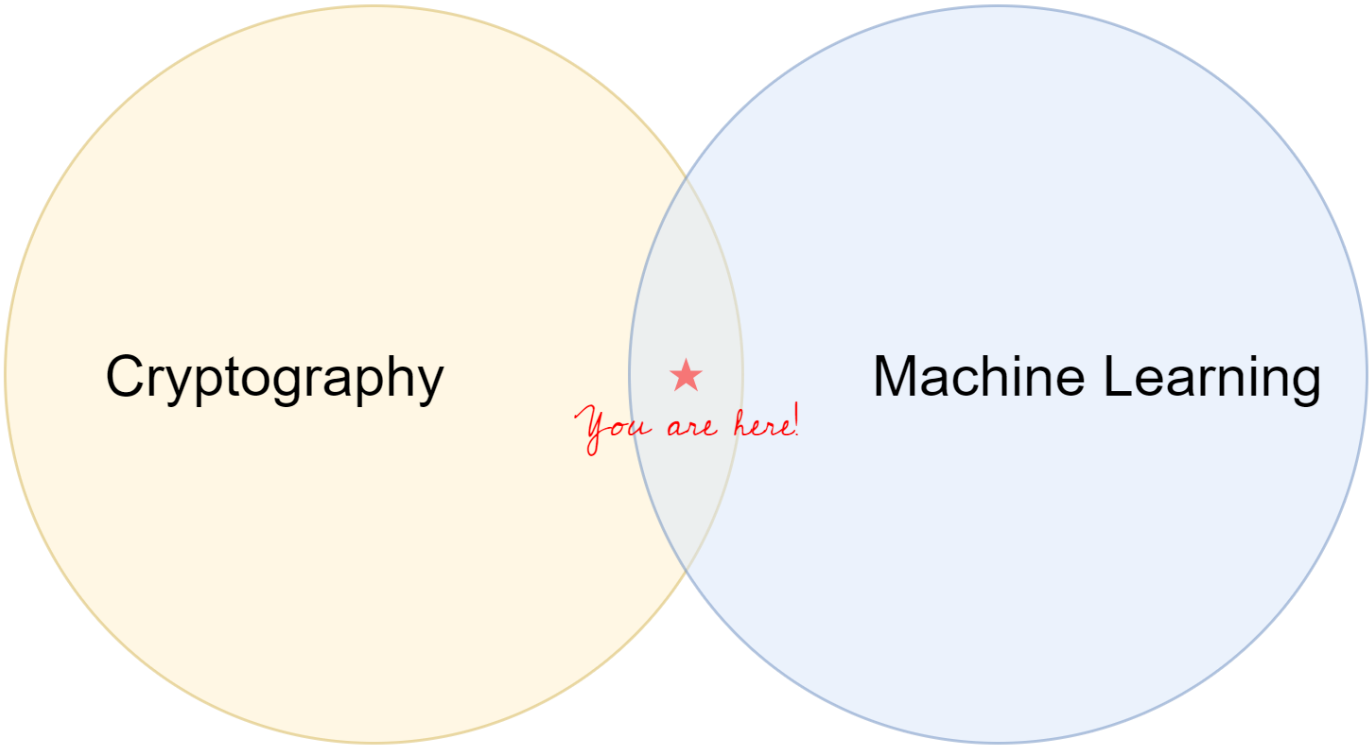
Where Machine Learning meets Cryptography

Solving the cryptographically-relevant Learning Parity with Noise Problem via machine learning

 Dr. Robert Kübler Jan 9, 2020 · 12 min read ★



When reading this, chances are that you know one or another thing about machine

end.

What you maybe have heard about (but did not dig deeper into) is the field of *cryptography*. It is this mysterious subject where it's all security, passwords, hiding things. Maybe you have even heard about *AES* or *RSA*, which are algorithms to *encrypt* data.

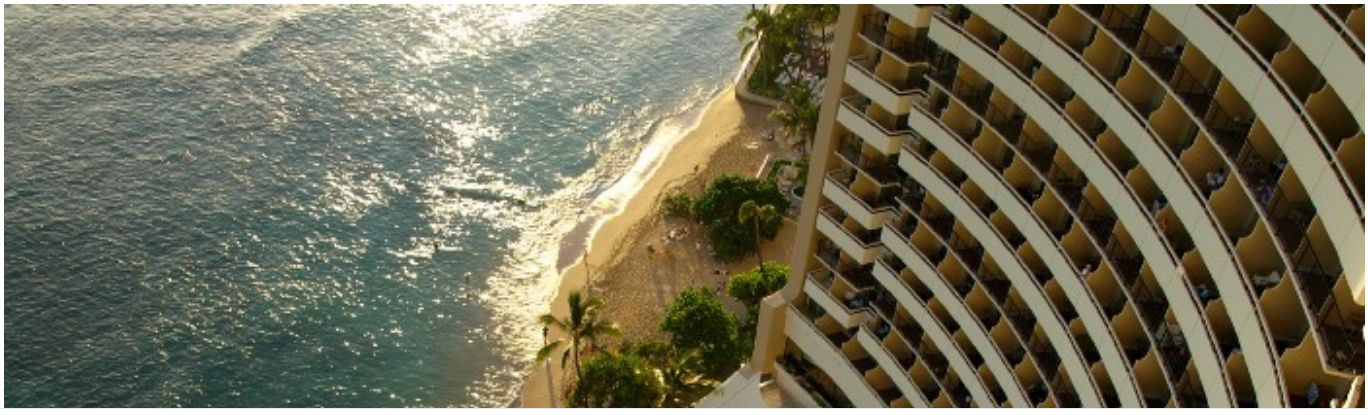
But don't worry, even if you have never dealt with cryptography before, you will be able to follow along since I will explain everything on an introductory level .

In this article, I want to bring both fields together. I will present to you an easy to understand, yet hard to solve problem used to build cryptographic algorithms — the so-called **Learning Parity with Noise** problem, **LPN** for short. The "L" in LPN should ring your machine learning alarm bells already because this problem can be seen as a routine machine learning problem!

But first, let us see where the LPN problem naturally arises in a cryptographic setting and how to define it. We will solve the LPN problem by using machine learning afterward.

Motivation

Imagine that you own a hotel and you want to manage access to the guests' rooms, i.e. each guest should only be able to enter their own room. Makes sense, right?



Your hotel. Photo by Eiji K on Unsplash.

Now, traditionally you could use normal, physical keys. The disadvantage is that people sometimes lose their keys, which means a lot of costs for your business since you have to replace the lock from the affected doors.

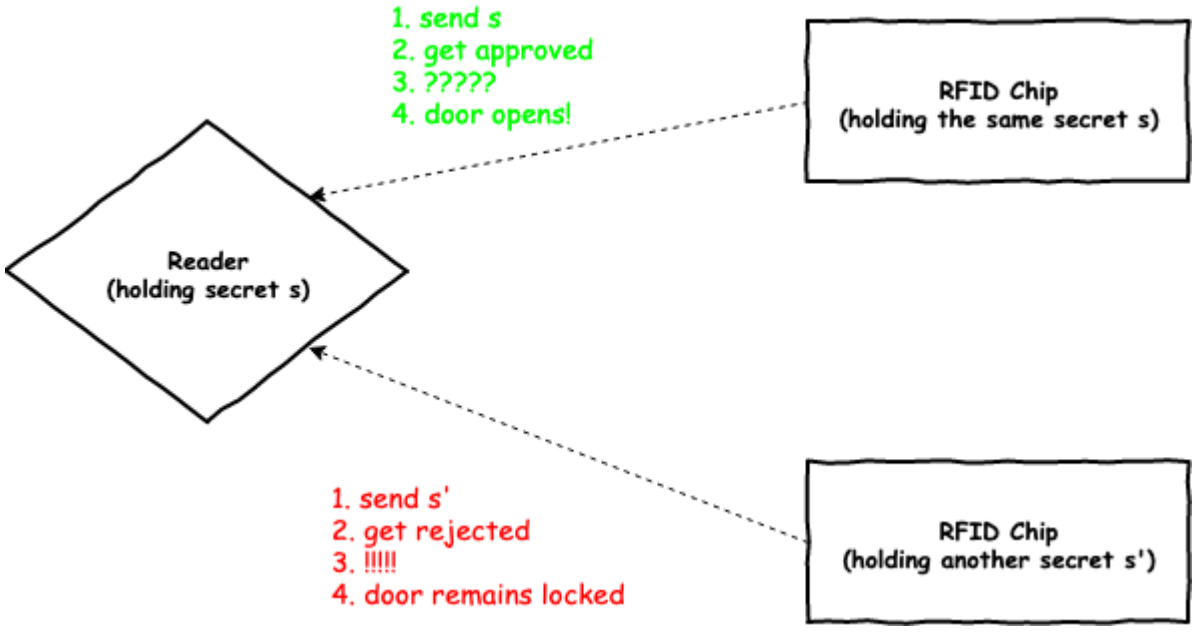
So you decide on deploying smart cards, in particular cards with RFID (radio-frequency identification) chips, and also the corresponding locks. Since you have to provide for a lot of doors and you want to save money, you choose **very weak RFID chips**, i.e. chips with diminishing computational power, maybe even without its own source of electricity.

Photo by Susanne Plank on Pixabay.

The way your system should work is the following: Every lock and every card has a **secret key** stored, a binary vector such as $s=(1,0,1,0)$, *just much longer in practice*. If you hold your card next to a lock, the lock works as a **reader**, scanning the card's secret key. The chip is called a **tag** in this context.

> The clue: If the secret keys of the card and the door match, the door opens.

Perfect! But how to do it? Well, an easy way is to hold your card next to the lock and the lock tells the chip on the card to send its secret key to the lock. Then the lock checks if both secret keys are equal and open the door, if yes.



This makes sense, because if you do not have the correct card, i.e. the secret key on your chip is different from the secret key in the door lock, the door will not open.

The Problem

typing innocently on their notebook. What the attacker actually does is *sniffing the RFID traffic,* i.e. reading the communication between the lock and the guest's chip. If the chip sends the secret key directly, the attacker will see it, store it, forge a card containing this key and then will be able to enter the room.

A prototypical hacker at work, this time without a ski mask. Photo by Nahel Abdul Hadi on Unsplash.

So, this is a bad idea. It only works if there are no bad people in the world (highly unlikely). Instead, we have to arm ourselves and improve security for our guests. The idea is the following:

possesses the correct secret key without revealing it.

I hear you scream: ***That's what encryption is for!*** And you are right. The attacker would only see garbage in the sniffing tool and wouldn't be able to reconstruct the key. But sadly, the RFID chip is much too weak for encrypting anything because you wanted to save money, remember? *Sadly, this is also true for bigger companies in the real world.* The chip has nearly no computational power and also only barely enough storage for its secret key. So we need another, more light-weight solution.

One way to do that is to use a cryptographic protocol like the **HB Protocol** by **Hopper** and **Blum** [1]. This protocol makes it difficult for this attacker to extract the key.

Photo by Goh Rhy Yan on Unsplash

security, more secure extensions of this protocol or other secure protocols should be used.

HB Protocol

So, you have a **reader *R*** (the lock) and a **tag *T*** (your chip). *T* now wants to prove to *R* that it possesses the same secret key **without revealing it.** This is done by *R* repeatedly challenging *T* with questions only a tag with the correct secret key can answer. So far, we have seen that the single question "What is your secret key?" is insecure since this reveals too much information already. Instead, in the HB Protocol *T* is asked to only reveal small portions of the secret one tiny bit at a time, until *R* can be sure that *T* has the correct secret key.

Imagine that the secret keys of *R* and *T* are in fact both the same $s=(1,0,1,0)$. Now *R* sends a random binary vector a (e.g. $a=(1,0,1,1)$) to *T* and expects *T* to respond back to it the scalar product $b=\langle a, s\rangle$, which is

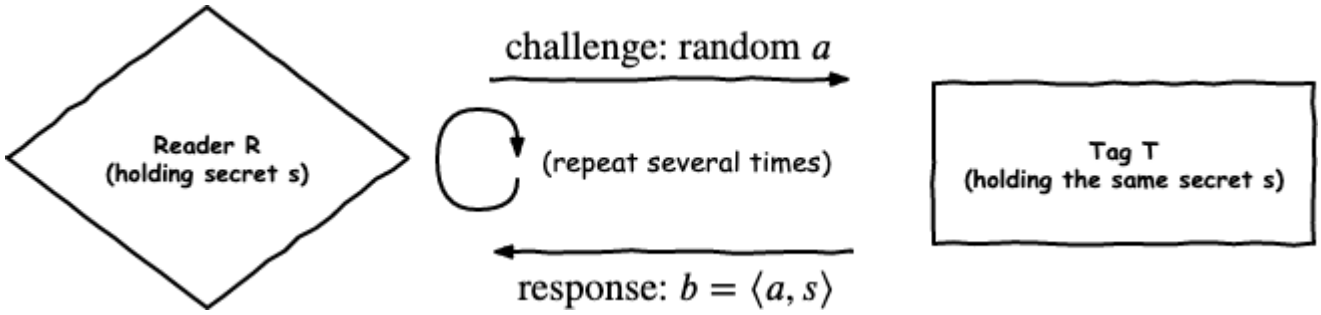
$$\langle (1,0,1,1),(1,0,1,0)\rangle = 1\cdot 1+0\cdot 0+1\cdot 1+1\cdot 0 = 1+1 = 0$$

in this example. We call this a a *challenge*. Remember, we deal with bit arithmetic here, so the "+" is, in fact, an XOR. The multiplication is the same as in the real numbers. Or for mathematicians: we calculate in the field GF(2) or $\mathbb{F}_2$, the field with 2 elements.

XOR	0	1
0	0	1
1	1	0

XOR is like the normal addition for integers, just that 1+1=0.

To increase confidence, this game is repeated several times.



For example, if T does not have the correct key, it would be very unlikely to succeed after a sufficient number of rounds, since a single response would be only correct with probability 0.5. Hence, after 10 rounds, for example, the chance of successful authentication is just 1/1024, **less than 0,1%**.

This sounds much better, right? T is not revealing its secret in one go now, instead, it gives some information to R by answering the challenges. **But sadly, this is also completely insecure.** An attacker could still write down the complete communication between R and T and then easily solve a system of linear equations to recover s. This is done in the following way: Imagine the attacker has written down the following for challenge/response pairs:

$$a_1 = (1, 0, 1, 1),\ b_1 = 0$$
$$a_2 = (0, 1, 1, 1),\ b_2 = 1$$
$$a_3 = (0, 0, 1, 0),\ b_3 = 1$$
$$a_4 = (0, 0, 0, 1),\ b_4 = 0$$

The attacker also knows that

where A is the matrix containing the a_i's as rows and b the b_i's. In our case:

The system of linear equations the attacker has to solve.

So solving this system for s yields the secret. This can also easily be done via Gaussian Elimination if s is much larger, i.e. 1024 bits long. The solution is $s=(1,0,1,0)$ by the way. :)

The Learning Parity with Noise Problem

There is one very small but **extremely important tweak** to make this secure against our attacker: T just adds some random Bernoulli noise to its responses. Instead of sending $<a,s>$ back to R, it flips a coin e which is 1 with probability p and 0 otherwise and sends back $<a,s>+e$ to the reader. In other words, with probability 1-p the tag sends $<a,s>$ back to R and with probability p it flips the response bit from 0 to 1 or from 1 to 0. We assume that $p<0.5$.

This does **not** prevent the attacker from sniffing the communication between R and T and taking notes, of course, but they have to solve the following problem now:

This notation indicates that each equation of the equation system is only correct with probability 1-p. More formally, you can write it as $As+e=b$, where e is the

> Thus, the attacker has to solve a noisy system of equations over GF(2). For a constant error rate p, this problem — the Learning Parity with Noise (LPN) Problem — is conjectured to be infeasible to solve for large enough length of the secret key. This is also true, if the attacker can get arbitrarily many equations.

Even with these errors added, R can do its job of determining whether T knows s or not. If T has the correct s, a fraction of about 1-p responses will be correct. That means if $p=0.25$ the HB Protocol runs for 1000 iterations, T should give **around 750 correct responses**.

If T does not have the correct s, it will give a fraction of around 0.5 correct answers, i.e. **500 out of 1000** rounds protocol run. This allows R to decide whether T has the correct secret or not and this protocol still makes sense for our use case.

Solving LPN via Machine Learning

Let's get to the fun part now. We have established that solving the LPN problem means, given a random binary matrix A and a binary vector $b=As+e$, recovering s.

The important observation: We can treat each row a_i of matrix A now as a sample and the corresponding value $b_i=<a_i, s>+e_i$ in the vector b as the label.

Using an example with a secret length of 4 and six captured communications, we can see that each row of matrix A consists of four features and the entry in b is the corresponding label. Our dataset has a size of six.

As found in normal datasets used in machine learning, a label b_i actually resembles the scalar product of the feature vector a_i and a fixed secret vector s *(some ground truth),* but with an error term added. But how can we get the secret s when we throw a machine learning algorithm for predicting the labels on it?

Well, if we could learn the problem *reasonably well,* we could generate good predictions for the labels (the scalar products; the ground truth) for each feature vector a_i we like. If we throw in the vector $a = (1,0,0,0)$, we would then receive a good guess for

the first bit of s! Do the same with the vectors (0, 1, 0, 0), (0, 0, 1, 0) and (0, 0, 0, 1) and we have all the bits of the secret key.

Thus, we can solve the LPN problem using machine learning.

Remarks

The LPN Problem is a very versatile problem that you can also use to build *encryption, identity-based encryption, user authentication, oblivious transfer, message authentication codes*, and probably more constructions. Also, unlike the factorization problem, the LPN problem cannot easily be solved using quantum computers. Paired together with its light-weightiness it is a good candidate for building post-quantum secure algorithms. So, no worries, if RSA, which is kind of based on factoring large numbers, dies in the presence of quantum computers. ;)

Experiments

Let us first define an *LPN oracle,* i.e. a class that we can feed with a secret key and an error level p upon instantiation, which gives us as many samples as we want.

This can easily be done using the following code:

```
import numpy as np

class LPNOracle:
    def __init__(self, secret, error_rate):
        self.secret = secret
        self.dimension = len(secret)
        self.error_rate = error_rate

    def sample(self, n_amount):
        # Create random matrix.
        A = np.random.randint(0, 2, size=(n_amount, self.dimension))

        # Add Bernoulli errors.
        e = np.random.binomial(1, self.error_rate, n_amount)

        # Compute the labels.
        b = np.mod(A @ self.secret + e, 2)

        return A, b
```

We can now instantiate this an oracle with a random secret of length 16 and $p=0.125$.

```
p = 0.125
dim = 16
s = np.random.randint(0, 2, dim)
lpn = LPNOracle(s, p)
```

We can now sample from the `lpn` :

```
lpn.sample(3)
```

```
array([1, 1, 1], dtype=int32))
```

Here we have sampled 3 data points. Now, let us try to find *s* using a Decision Tree. Why? Just because it's fast and Logistic Regression and Bernoulli Naive Bayes did not work for me.

```
from sklearn.tree import DecisionTreeClassifier

dt = DecisionTreeClassifier()

# Get 100000 samples.
A, b = lpn.sample(100000)

# Fit the tree.
dt.fit(A, b)

# Predict all canonical unit vectors (1, 0, 0, ..., 0), (0, 1, 0 ,0,
..., 0), ..., (0, 0, ..., 0, 1).
s_candidate = dt.predict(np.eye(dim))

# Check if the candidate solution is correct.
if np.mod(A @ s_candidate + b, 2).sum() < 14000:
    print(s_candidate, s)
else:
    print('Wrong candidate. Try again!')
```

The learning algorithm might fail to capture the ground truth and learn another function. In this case, the so-called *Hamming weight* of the vector is quite high (around 50000 for our vector of length 100000). This corresponds to the case where the tag T has the wrong key and can answer about half of the challenges correctly. If s_candidate $= s$, the Hamming weight will be around 0.125 * 100000 = 12500.

Having a threshold of 14000 in this example is a good tradeoff between recognizing the correct secret and not outputting a wrong candidate as the solution. You can find how to get this bound in [2, page 23].

Conclusion

Decision Tree.

But the journey just starts here: We can use other/better algorithms (deep learning, anyone?) or clever tricks to

- get higher success rates,
- using fewer samples and
- being able to solve problems with higher dimensions.

For a list and explanations of non-machine learning algorithms to solve LPN, check out my dissertation [2]. Also, if you want fame, try to solve an instance with a secret length of 512 and p=0.125, for example. This LPN instance is currently unbroken and used for some real-world cryptosystems. Good luck! ;)

References

[1] N. Hopper and M. Blum, Secure human identification protocols (2001), International conference on the theory and application of cryptology and information security, Springer

[2] R. Kübler, Time-Memory Trade-Offs for the Learning Parity with Noise Problem (2018), Dissertation (Ruhr University Bochum)

I hope that you learned something new, interesting, and useful today. Thanks for reading!

As the last point, if you

1. **want to support me in writing more about machine learning and**

Why not do it via this link? This would help me a lot!

To be transparent, the price for you does not change, but about half of the subscription fees go directly to me.

Thanks a lot, if you consider supporting me!

If you have any questions, write me on LinkedIn!