# Time-Lock Encryption: The Digital Time Capsule

This project is a Python implementation of a **Time-Lock Puzzle**, based on the original paper "Time-Lock Puzzles and Timed-Release Crypto" by Rivest, Shamir, and Wagner.

It allows you to encrypt a file in such a way that it can only be decrypted after a specific, real-world time delay has passed. The decryption process involves solving a computational puzzle that is inherently sequential and cannot be significantly sped up by using more powerful computers (parallel processing).

This makes it a "digital time capsule": you can lock a secret away, and be confident that no one, including yourself, can access it until the puzzle is solved.

## How It Works

The system relies on the mathematical properties of modular arithmetic.

1. **Encryption (The Shortcut):**
   * A large RSA modulus n is created from two secret primes, p and q. The secret phi(n) = (p-1)(q-1) is calculated.
   * A puzzle is created that requires calculating g^(2^t) mod n. Using the secret phi(n), the creator can solve this puzzle almost instantly.
   * A random AES key is generated to encrypt the target file.
   * This AES key is then "hidden" by XORing it with the puzzle's solution.
   * The secret primes p and q are discarded. The final output is the encrypted file and a puzzle.json file containing n, the time parameter t, and the hidden key.
2. **Decryption (The Hard Way):**
   * The decrypter does not have the secret phi(n).
   * Their only way to find the puzzle's solution is to perform the squaring operation t times in a row: g -> g^2 -> g^4 -> g^8 ... up to 2^t, all modulo n.
   * This sequential work is what creates the time delay.
   * Once the puzzle is solved, the result is XORed with the hidden key to reveal the AES key, which is then used to decrypt the file.

## Features

* **Adjustable Time Delay:** Specify the desired delay in seconds.
* **Machine Calibration:** A calibrate command benchmarks your machine's performance to make the time delay more accurate.
* **Secure Encryption:** Uses AES-GCM for authenticated encryption of the file data.
* **User-Friendly CLI:** A simple command-line interface to calibrate, encrypt, and decrypt files.
* **Progress Bar:** The tqdm library provides a progress bar during the long-running decryption process.

## Setup

1. **Clone the repository or save the timelock.py script.**
2. **Install the required Python libraries:**  
   pip install pycryptodome tqdm

## Usage

The tool has three main commands: calibrate, encrypt, and decrypt.

### Step 1: Calibrate (Important First Step!)

Before you encrypt anything, you should calibrate the tool to your machine's speed. This saves a calibration.json file that is used to estimate how many computations are needed for a given time delay.

python timelock.py calibrate

You only need to do this once, unless your machine's performance changes significantly.

### Step 2: Encrypt a File

Create a sample file to encrypt. For example, secret.txt.

To encrypt secret.txt with a target delay of **60 seconds**, run:

# Create a dummy file  
echo "This is a secret message for the future." > secret.txt  
  
# Encrypt it with a 60-second delay  
python timelock.py encrypt --input secret.txt --delay 60 --output my\_time\_capsule

This will produce two files:

* my\_time\_capsule.enc: The AES-encrypted data.
* my\_time\_capsule.json: The puzzle parameters needed to solve and decrypt.

### Step 3: Decrypt the File

To decrypt the file, you must solve the puzzle. This will take approximately the amount of time you specified in the delay.

python timelock.py decrypt --puzzle my\_time\_capsule.json --input my\_time\_capsule.enc --output decrypted\_secret.txt

You will see a progress bar while the puzzle is being solved.

Solving Puzzle: 100%|██████████| 12345678/12345678 [01:02<00:00, 198123.45it/s]  
Puzzle solved!  
Decryption successful!  
Decrypted file saved to: decrypted\_secret.txt

After it's finished, the decrypted\_secret.txt file will contain the original content.