Digital Time Capsule: Implementation and Analysis of RSA Time-Lock Puzzle with AES-256-GCM Encryption

Theresia Ivana Marella Siswahyudi - 18223126
Program Studi Sistem dan Teknologi Informasi
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: theresiaims@gmail.com, 18223126@std.stei.itb.ac.id

***Abstract*—This research provides an experimental investigation of Digital Time Capsule systems, a novel temporal access control mechanism. The study includes quantitative assessment of the operational performance of time-lock cryptography through direct implementation of the Rivest-Shamir-Wagner (RSW96) scheme combined with authenticated symmetric encryption using AES-256-GCM. Moreover, this study provides a thorough empirical analysis of how RSA modulus bit-lengths affect squaring throughput, alongside evaluating the precision of the CPU speed calibration module under environmental resource throttling. Our findings show that although modular squaring effectively guarantees sequential computational delays, sustained hardware execution triggers thermal throttling that induces a systematic positive calibration bias over extended time horizons. Different RSA moduli exhibit massive performance variations, with 512-bit lengths being the fastest but cryptographically vulnerable, and 4096-bit lengths being the most secure but significantly slower. The results can provide insights to help software architects and system designers configure, implement, and optimize reliable decentralized timed-release data security infrastructures.**

Keywords—Digital Time Capsule, RSA Time-Lock Puzzle, AES-256-GCM, Sequential Squaring, CPU Throttling, Calibration Bias.

I. Introduction

A. Context and Problem Statement

Rivest Shamir Wagner Time Lock Puzzles (RSW96 TLP) stand as a cornerstone of modern temporal access control architectures, offering a robust defense against common data vulnerability windows and unauthorized early disclosures, which often bypass traditional policy based or Trusted Third Party (TTP) escrow systems. Defined by the original mathematical scheme, this cryptographic approach leverages the inherently sequential nature of modular squaring in a group of unknown order to enforce an inescapable computational delay, significantly enhancing timed release security. Despite its widespread adoption in theoretical designs, a comprehensive empirical understanding of the real world performance characteristics of time lock puzzles and their resilience to operational challenges, particularly environmental Central Processing Unit (CPU) frequency scaling and thermal throttling under sustained workloads, remains vital for optimal deployment.

In practical implementations, this technology is frequently integrated into modern hybrid cryptographic systems which establishes a unique, temporary key protection envelope using large public moduli and iterative mathematical computation, thereby significantly increasing timed-release data security. However, optimal deployment of these systems still requires a thorough empirical understanding of real world performance characteristics and the ability to withstand operational difficulties, especially hardware calibration discrepancies and operating system context switching overhead.

B. Scope and Objectives

This report addresses these areas by quantitatively evaluating the operational efficiency of the system, empirically analyzing the tolerance of the capsule to resource deprivation and hardware frequency drift. The specific objectives of this paper are to (1) measure the average execution throughput for sequential modular squaring operations across diverse RSA moduli bit lengths; (2) empirically examine the effects of operating system scheduler overhead and hardware thermal throttling on time lock puzzle calibration accuracy; (3) examine the performance and security characteristics of various cryptographic primitive configurations such as Advanced Encryption Standard in Galois Counter Mode (AES-256-GCM); and (4) discuss the practical security performance trade offs related to modulus scaling configurations and the impact of hardware environments on digital time capsule stability.

C. Paper Organization

The remainder of this research paper is structured systematically to facilitate a comprehensive understanding of the digital time capsule system. Section 2 (Background and Related Work) establishes the mathematical foundations of the Rivest Shamir Adleman (RSA) cryptosystem, details the sequential complexity of modular squaring, reviews the original Rivest Shamir Wagner puzzle design, and defines the mechanics of authenticated encryption. Section 3 (System Design and Architecture) delineates the high level software blueprints, components decoupling, and the exact binary

container serialization format used for the time capsule payload. Section 4 (Implementation) highlights the practical software engineering aspects, focusing on prime generation using Cryptographically Secure Pseudo Random Number Generators (CSPRNG), the median window calibration strategy, and critical code level cryptographic bug fixes.

Section 5 (Empirical Evaluation) presents comprehensive experimental data, mapping modulus sizes against throughput, evaluating calibration accuracy across diverse time horizons, and analyzing Central Processing Unit throttling simulations. Section 6 (Security Analysis) discusses the computational hardness assumptions, parallel computing immunities, and vulnerabilities against specialized Application Specific Integrated Circuit (ASIC) architectures. Section 7 (Discussion and Future Work) explores advanced cryptographic mitigations, including Verifiable Delay Functions (VDF) and public blockchain state machine integrations. Finally, Section 8 (Conclusion) synthesizes the definitive insights and contributions realized through this research.

II. Background and Related Work

A. RSA Cryptosystem Fundamentals

The theoretical foundation of temporal data access control structures relies heavily on the algebraic properties exploited within public key cryptography, specifically the framework established by the Rivest Shamir Adleman (RSA) cryptosystem [1]. Let p and q be two distinct, cryptographically secure large prime numbers generated uniformly at random. The composite public modulus n is computed as the product of these two prime factors:

$$n = p \cdot q \quad \ldots(1)$$

The size of this public modulus, traditionally measured in bit lengths ranging from 512 bits to 4096 bits, dictates the computational bounds of the system. The order of the finite multiplicative group of integers modulo n, denoted as $\mathbb{Z}_n^*$, is given by Euler's totient function $\phi(n)$, which evaluates to:

$$\phi(n) = (p - 1)(q - 1) \quad \ldots(2)$$

According to Euler's Theorem, for any base element $a \in \mathbb{Z}_n^*$ that is relatively prime to n (such that $gcd(a, n) = 1$), the group identity holds true:

$$a^{\phi(n)} \equiv 1 \ (mod\, n) \quad \ldots(3)$$

An indispensable structural property exploited within Time-Lock Puzzles (TLPs) is the computational asymmetry found in modular exponentiation. For any arbitrary exponent e, the calculation of $a^e \ (mod\, n)$ can be executed efficiently via fast exponentiation algorithms, such as the square and multiply method, in $O(log\, e)$ modular multiplications. However, this algorithmic optimization is strictly dependent on the capability to reduce the large exponent e modulo the group order $\phi(n)$ prior to performing the squaring sequence:

$$a^e \equiv a^{e \ (mod\, \phi(n))} \ (mod\, n) \quad \ldots(4)$$

Knowledge of $\phi(n)$ allows a system creator to compress an exponentially large exponent into a value bounded by n. Crucially, computing $\phi(n)$ rom the public modulus n is computationally equivalent to finding the prime factors p and q. For a solver who lacks access to these secret trapdoor primes, the value $\phi(n)$ remains completely unobtainable. Consequently, the shortcut reduction becomes inaccessible, forcing the solver to evaluate the full exponent size through a raw, unoptimized chain of operations.

B. Modular Squaring: Sequential and Parallel Complexity

The computational implementation of predictable wall clock delays requires an absolute barrier against parallelization. Let $x_0 = a$ represent a random starting base element chosen from the multiplicative group $\mathbb{Z}_n^*$. The mathematical sequence generated by repeated, iterative modular squaring is formally defined as:

$$x_1 \equiv x_0^2 \ (mod\, n)$$

$$x_2 \equiv x_1^2 \equiv a^{2^2} \ (mod\, n)$$

$$x_3 \equiv x_2^2 \equiv a^{2^3} \ (mod\, n)$$

$$\ldots$$

$$x_T \equiv x_{T-1}^2 \equiv a^{2^T} \ (mod\, n) \quad \ldots(5)$$

This recursive structure builds a rigid sequential dependency chain where each intermediate state x_i acts as the direct and irreplaceable input for the subsequent squaring operation x_{i+1}. The mathematical foundation of this temporal lock rests upon the Sequential Squaring Assumption [2]. This assumption states that within a multiplicative group of unknown order, no probabilistic polynomial time algorithm can compute the final state $b = a^{2^T} \ (mod\, n)$ significantly faster than executing the full T sequential modular squarings, provided that n is an RSA composite modulus of sufficient bit length.

Unlike spatial cryptographic puzzles that can be cracked by distributing subtasks across massive hardware networks, the sequential nature of modular squaring acts as an absolute countermeasure against parallel processing. Introducing multiple Central Processing Unit (CPU) cores, Graphics Processing Units (GPUs), or Field Programmable Gate Arrays (FPGAs) yields no operational advantage for accelerating a single calculation path. Pipelining techniques within hardware architectures can optimize individual modular multiplication operations by executing sub steps faster, but they offer at most a minor constant factor speedup. The absolute dependency chain remains unparallelizable, ensuring that an adversary backed by a supercomputer cannot solve the puzzle faster than a standard single thread processor of equivalent clock rate.

C. AES-256-GCM: Authenticated Encryption

To protect arbitrary file sizes and structural data types without hitting the mathematical limits of asymmetric moduli, the digital time capsule deploys a hybrid encryption architecture. The Time-Lock Puzzle functions strictly as a Key Encapsulation Mechanism (KEM) to protect a 32-byte

symmetric payload key, while the actual data payload is processed via the Advanced Encryption Standard in Galois/Counter Mode (AES-256-GCM) in compliance with National Institute of Standards and Technology (NIST) guidelines [3].

AES-256-GCM is an Authenticated Encryption with Associated Data (AEAD) scheme that provides confidentiality via counter mode data encryption and cryptographic integrity verification via a Galois Message Authentication Code (GMAC). Let K be the 256-bit symmetric key derived from the puzzle solution b, and let IV be a uniformly random 96-bit Initialization Vector.

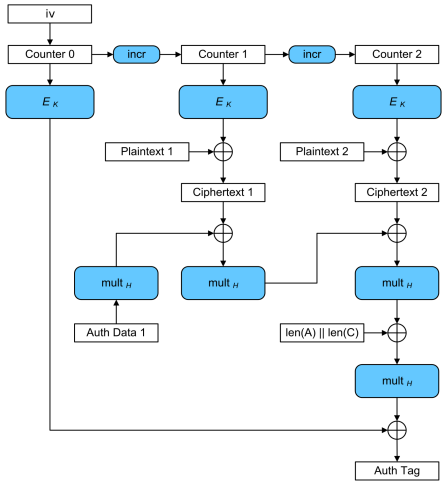

Fig. 1. AES-GCM Encryption and Authentication Pipeline Structure [7]

The authenticated encryption transformation is formulated as:

$$(C, tag) = AES\_GCM\_Encrypt(K, IV, Plaintext) \quad \ldots(6)$$

The decryption architecture requires the exact symmetric key K, the unique 12-byte IV, the ciphertext C, and the 16-byte authentication tag (tag):

$$Plaintext = AES\_GCM\_Decrypt(K, IV, C, tag) \quad \ldots(7)$$

The security of Galois Counter Mode relies strictly on nonce uniqueness; reusing an IV with the same key breaks the authenticity guarantees and exposes the system to ciphertext manipulation. For the digital time capsule architecture, a random IV is completely secure because each symmetric key K is generated dynamically for a single capsule instance, ensuring a strict single use scenario per key. If an adversary attempts to modify the ciphertext container, the GMAC verification routine immediately flags a validation failure, preventing padding oracle attacks and ensuring that partial or corrupted puzzle solves do not leak plaintext attributes.

D. RSA Time-Lock Puzzle Construction

The structural protocols enabling decentralized timed release cryptography were formalized by Rivest, Shamir, and Wagner through a dual phase cryptographic algorithm consisting of a Setup Phase and a Solve Phase [2]. This construction maps a target duration t directly to the required computational depth T. The execution parameters and operational phases of the Rivest Shamir Wagner Time-Lock Puzzle protocol are structured into two distinct, sequential procedures consisting of a setup phase and a solving phase:

1) Setup Phase (Executed by the Creator)
The initialization and capsule creation path requires the data owner to establish the mathematical boundary constraints using the following computational steps:

1. Generation of Prime Parameters: Two distinct, large secret prime numbers, denoted as p and q, are generated uniformly at random utilizing a Cryptographically Secure Pseudo-Random Number Generator (CSPRNG).
2. Modulus Assembly: The public composite modulus is compiled by evaluating $n = p \cdot q$, which dictates the cryptographic key length configuration.
3. Group Order Calculation: The private Euler totient value representing the order of the multiplicative group is calculated via $\phi(n) = (p - 1)(q - 1)$.
4. Iteration Depth Mapping: Given a target wall clock duration t measured in seconds and an estimated single thread execution rate S measured in modular squarings per second, the absolute loop bound is mapped using $T = \lfloor t \cdot S \rfloor$.
5. Base Element Selection: A random starting base element a is selected uniformly from the open multiplicative interval between 2 and $n - 2$.
6. Shortcut Exponentiation: Utilizing explicit knowledge of the group order, a compressed shortcut exponent is calculated almost instantaneously via $e = 2^T \ (mod\ \phi(n))$ in $O(log\ T)$ steps.
7. Trapdoor Evaluation: The precise final puzzle state solution is determined by evaluating $b = a^e \ (mod\ n)$ using fast modular exponentiation.
8. Symmetric Key Derivation: A uniform symmetric cryptographic key K is generated by processing the mathematical solution b through the Secure Hash Algorithm 256-bit variant (SHA-256).
9. Envelope Masking: The derived symmetric key K is encapsulated inside a temporal mask to produce the puzzle ciphertext envelope using a bitwise exclusive OR operation, formulated as $C_K = K \oplus Hash(b)$.
10. Memory Purging: The public container parameters consisting of (n, a, T, C_K) are compiled for publishing, while the secret values $p, q, \phi(n), and\ K$ are immediately erased from the system runtime memory.

2) Solve Phase (Executed by the Solver)
The decapsulation and recovery path forces any external entity to traverse the complete sequential depth without computational shortcuts through the following steps:

1. State Initialization: The computational engine initializes the sequential squaring sequence by assigning the starting loop variable to the public base parameter, where $x_0 = a$.

2. Sequential Squaring Execution: The core processor enters a strict iterative loop executing from $i = 1$ to T, where each consecutive state is recursively computed via $x_i = x_{i-1}^2 \ (mod\ n)$.
3. Solution Assignment: Upon reaching the exact limit of the iteration count, the final state of the sequence is captured to define the verified puzzle solution, where $b = x_T$.
4. Key Unmasking: The hidden symmetric key envelope is unmasked by computing a secondary bitwise exclusive OR operation against the hashed puzzle solution, calculated as $K = C_K \oplus Hash(b)$.
5. Symmetric Payload Extraction: The recovered key K is returned to the main hybrid block cipher engine to begin the authenticated decryption of the digital time capsule contents.

This protocol achieves asymmetric complexity because the creator leverages their knowledge of $\phi(n)$ to instantly evaluate the solution b in a fraction of a millisecond using the mathematical trapdoor shortcut, whereas the solver is forced to step through all T steps of the loop execution sequence.

E. Related Work

Following the landmark proposal by Rivest et al., timed release cryptography branched into several paradigms [2]. Boneh and Naor formalized the concepts of timed commitments, linking sequential delay generation directly to cryptographic bit commitment protocols [4]. Later foundational work by Mahmoody et al. demonstrated through black box models that sequential computation designs represent an optimal upper bound for enforcing decentralized temporal barriers without external communication channels [5].

Recent advancements in distributed ledger technologies have sought to build public time locks using blockchain state machines. Practical examples include hash based time locks deployed within decentralized payment channels and smart contracts that anchor release criteria to specific block heights or consensus network timestamps. However, these systems face challenges with validator timestamp drift and block time manipulation by network miners.

To overcome these vulnerabilities, Boneh et al. formalized Verifiable Delay Functions (VDFs) [6]. A Verifiable Delay Function extends the properties of a traditional Time-Lock Puzzle by generating both a sequential mathematical delay and a succinct cryptographic proof. This proof allows any third party system to verify the accuracy of the computation in $O(log\ T)$ logarithmic steps or $O(1)$ constant time without repeating the sequential squaring chain. While Verifiable Delay Functions are useful for blockchain consensus protocols, traditional Time-Lock Puzzles remain the primary mechanism for locking confidential payloads because Verifiable Delay Functions focus on publicly verifiable delay generation rather than secret key encapsulation.

III. System Design and Architecture

A. Design Goals and Requirements

The Digital Time Capsule was designed to satisfy five core requirements that jointly balance cryptographic soundness with practical usability:

1) Mathematical Rigor: All temporal-lock guarantees derive strictly from the Sequential Squaring Assumption (Section II), with payload confidentiality and integrity delegated to AES-256-GCM. No component of the system relies on a trusted third party, secret escrow, or hardware root of trust.
2) Flexibility: The system supports arbitrary payload types, plain text as well as binary files (PDF, DOCX, JPG, PNG, and others), by transparently Base64-encoding binary content before it enters the AES-GCM encryption pipeline.
3) Calibrated Delay Accuracy: The number of required squarings, T, is not fixed but dynamically computed from a host-specific CPU calibration step, so that a user-specified wall-clock duration (e.g., "lock for 30 seconds") maps as closely as possible to the actual solve time on the creating machine.
4) User Usability: Two parallel interfaces are exposed: a Rich-formatted command-line interface for technical users and scripting, and a Flask-based web UI for general users who prefer browser interaction.
5) Solve Progress Feedback: The solving a puzzle may take anywhere from seconds to minutes, the system must report incremental progress rather than blocking silently. This is achieved through a generator-based solver that yields percentage-complete updates, consumed either by a console progress bar (CLI) or a Server-Sent Events (SSE) stream (Web).

B. High-Level Architecture

The codebase is organized into four decoupled layers, each with a single, well-defined responsibility. This separation allows the cryptographic core to be independently tested, reused across both interfaces, and extended without modifying presentation logic.

1) Core Layer (`core/`): Contains `crypto.py`, which implements AES-256-GCM encryption/decryption and SHA-256-based key derivation, and `tlp.py`, which implements RSA parameter generation, CPU calibration, puzzle creation (using the $\phi(n)$ shortcut), and the sequential puzzle solver. This layer has no dependency on any user interface and can be imported and tested in isolation.
2) CLI Layer (`cli/`): Contains `commands.py`, which defines the calibrate, create, and solve subcommands using Python's argparse, and renders all output (tables, panels, progress bars) using the rich library.
3) Web Layer (`web/`): A Flask application (`app.py`, `routes.py`) that exposes a REST API (`/api/calibrate`, `/api/create`, `/api/solve-stream`) consumed by an HTML/JavaScript frontend (`index.html`, `app.js`, `style.css`). The solve endpoint streams real-time progress to the browser via Server-Sent Events.

4) Benchmark Layer (`benchmark/`): Contains `runner.py`, which executes the three empirical test suites described in Section III-A2 below, and `visualizer.py`, which renders the resulting data into publication-quality matplotlib figures saved to `benchmark_data.json` and `static/charts/`.

This layered design means that, for example, the AES-GCM implementation could be swapped for a different AEAD scheme, or a new frontend (e.g., a mobile app) could be added, without touching the underlying TLP mathematics.

C. Experimental Design

To characterize the implementation's performance and timing reliability, a dedicated benchmarking suite (`run_benchmarks.py`) was constructed around three distinct experiments:

1) Modulus Keysize Benchmark: Measures raw computational throughput (squarings per second) across four RSA modulus sizes: 512, 1024, 2048, and 4096 bits. This quantifies the direct cost of increasing cryptographic strength.
2) Calibration Accuracy Test: Measures the deviation between a puzzle's target lock duration and its actual measured solve time. Puzzles were created with target durations of 2, 5, 10, 20, and 30 seconds at a fixed 2048-bit modulus, and the resulting error (in both absolute seconds and percentage terms) was recorded for each.
3) Throttling Sensitivity Simulation: Evaluates solver behavior under constrained computational resources. Artificial delays are injected into the modular squaring loop to emulate four CPU allocation levels, 100%, 75%, 50%, and 25%, allowing the system's degradation curve to be measured without requiring physical access to throttled hardware.

All benchmarks were executed on an Intel-based host system running Windows, using Python 3.11 within a virtualized project environment. This configuration is noted explicitly because, as discussed in Section IV-B, host-specific factors such as CPU frequency scaling materially affect calibration accuracy.

D. Cryptographic Protocol Flow

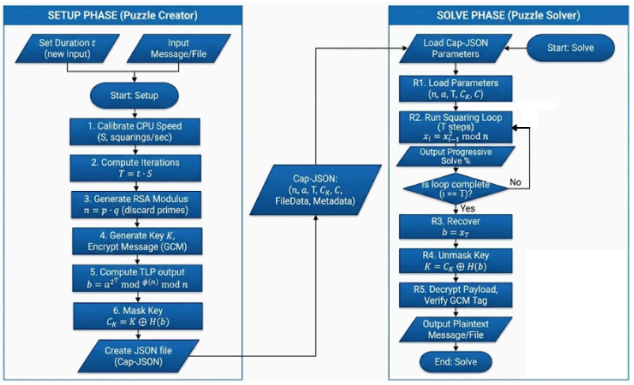

Fig. 2. Cryptographic Protocol Flow

The end to end cryptographic protocol is structured into two asymmetric execution phases, which are the Setup (Encapsulation) phase and the Solve (Decapsulation) phase. In the Setup phase, the capsule creator initializes the pipeline by calibrating the single thread hardware throughput speed S, measured in modular squarings per second. This capacity coefficient maps the user defined target delay duration t to the absolute loop iteration depth $T = \lfloor t \cdot S \rfloor$. Concurrently, the core layer generates a fresh Rivest Shamir Adleman (RSA) composite modulus $n = p \cdot q$, whereupon the secret prime factors p and q are immediately scrubbed and discarded from active system memory. A uniform 256-bit symmetric key K is generated from the operating system entropy pool to encrypt the payload data utilizing the Advanced Encryption Standard in Galois Counter Mode (AES-256-GCM) primitive. To lock the symmetric key securely, the system evaluates the trapdoor time lock puzzle output $b = a^{2^T (mod\ \phi(n))} (mod\ n)$ using fast modular exponentiation based on the secret group order $\phi(n)$. The key envelope is then masked by evaluating a bitwise exclusive OR operation against the cryptographic hash of the puzzle solution, formulated as $C_K = K \oplus H(b)$. The final parameters are serialized into a public JavaScript Object Notation (JSON) container file designated as the Cap-JSON capsule.

In the Solve phase, the decapsulation engine loads the public container parameters from the Cap-JSON document and initiates an intensive sequential loop. The processor executes repeated modular squaring operations formulated as $x_i = x_{i-1}^2 (mod\ n)$, starting from the public base element a. This loop is repeated continuously until the current index reaches the iteration boundary condition where $i = T$. Throughout this sequential execution path, the stateful generator core periodically emits non blocking execution frames to update the visual progressive solve percentage indicators on the active user interface.Upon reaching the loop boundary, the final state value x_T is extracted as the recovered puzzle solution b. The hidden symmetric key is unmasked by evaluating the inverse bitwise operation $K = C_K \oplus H(b)$, allowing the system to decrypt the ciphertext blocks and verify data integrity via the Galois Message Authentication Code tag validation routine. This structural operational asymmetry, which demands only $O(logT)$ exponential work for the capsule creator versus a massive $O(T)$ strictly sequential dependency chain for the solver, forms the fundamental security basis of the decentralized temporal time lock guarantee.

E. Serialization Format

The completed time capsule is serialized as a single JSON document for portability across the CLI and web interfaces. Two encoding conventions are used to preserve data fidelity:

1) Large integers (n, a) are serialized as decimal strings, rather than native JSON numbers, because JavaScript's `Number` type is limited to 53-bit precision floating-point representation, insufficient for 2048-bit or 4096-bit RSA values. Encoding these fields as strings avoids silent precision loss when the capsule is consumed by the browser-based frontend.

2) Binary byte fields (C_K, $ciphertext$, $nonce$, tag) are hex-encoded strings, ensuring safe transport within JSON's text-only format.

Additional metadata fields describe the puzzle parameters and, where applicable, original file attributes. The complete schema is summarized in Table I.

TABLE I. TIME CAPSULE JSON SCHEMA FIELDS

Field	Type	*Description*
`n`	String (Decimal)	RSA modulus
`a`	String (Decimal)	Generator base
`T`	Integer	Total required squaring count
C_K	String (Hex)	Masked symmetric AES key
`ciphertext`	String (Hex)	Encrypted payload
`nonce`	String (Hex)	GCM nonce (96-bit)
`tag`	String (Hex)	GCM authentication tag (128-bit)
`bits`	Integer	Modulus bit size (e.g., 2048)
`ops_per_second`	Float	Calibrated CPU squaring speed
`target_duration`	Float	Intended lock time (seconds)
`is_file`	Boolean	True if payload is a binary file
`file_name`	String	Original filename (if file)
`file_type`	String	MIME type (if file)

IV. IMPLEMENTATION DETAILS

A. RSA Parameter Generation

The secret prime numbers p and q underlying the public composite modulus are generated using the `getPrime` function provided by the PyCryptodome library. This utility sources entropy directly from the host operating system cryptographically secure pseudo random number generator, such as `/dev/urandom` on Linux platforms or `BCryptGenRandom` within Windows environments. The generation layer internally applies an iterative Miller Rabin primality test to verified candidate big integers to restrict the error probability to negligible cryptographic bounds.

For a target public modulus of b bits, the two prime factors are independently generated with bit lengths of $\lfloor b/2 \rfloor$ and $b - \lfloor b/2 \rfloor$ respectively. This specific allocation ensures that the resulting mathematical product lands precisely within the intended bit length range boundaries. A strict validation loop is implemented to guarantee that $p \neq q$:

```
n = p * q
phi = (p - 1) * (q - 1)
```

Critically, the private primes p and q are explicitly scrubbed and discarded from active system memory immediately after the group order $\phi(n)$ is utilized to evaluate the puzzle shortcut exponent. These secret elements are never appended to the serialized JavaScript Object Notation container nor persisted to local disk storage. This mechanism represents a critical data security architecture requirement. If an adversary could recover p, q, or $\phi(n)$ through a memory dump or a physical storage exploit, the entity could compute the compressed shortcut exponent $e = 2^T \pmod{\phi(n)}$ directly. This structural compromise completely bypasses the sequential modular squaring path, destroying the temporal lock guarantees and undermining the core information asymmetry that separates the creator from the solver.

B. CPU Calibration

The system calibration routine, implemented via `calibrate_cpu`, estimates the single thread modular squaring throughput capacity of the target host environment. This evaluation is performed by executing repeated modular squaring operations against a temporary, disposable public modulus for a fixed wall clock duration. The default baseline window is configured for 1.5 seconds during standard capsule encapsulation, and remains variable within the automated benchmarking suite:

```
start_time = time.perf_counter()
while time.perf_counter() - start_time <
duration:
    x = pow(x, 2, n)
    count += 1
ops_per_second = count / elapsed
```

The resulting mathematical throughput estimate S, measured in modular squarings per second, maps the user defined target lock time t to the absolute loop bound $T = \lfloor t \cdot S \rfloor$. This performance sampling is strictly conducted at the identical modulus bit length configuration selected for the final time capsule since the computational complexity of squaring operations scales directly with public modulus sizes.

However, executing calibration over a single brief measurement window introduces substantial processing inaccuracies. First, short window evaluation is highly vulnerable to scheduling noise generated by the background operating system, concurrent processing threads, hardware interrupts, and Just In Time compiler or cache warm up latency. These anomalies distort the throughput sample, making it unrepresentative of sustained system behavior. Second, modern processors utilize dynamic hardware frequency scaling frameworks, such as Intel Turbo Boost technology, which permit single processing cores to run at elevated clock frequencies during brief computational bursts.

Because a 1.5 second calibration window fails to trigger long term thermal throttling limits, the calibration engine systematically overestimates the sustained performance speed available during a lengthy decapsulation execution path. This performance divergence directly accounts for the massive positive timing bias identified during extended temporal horizons, where the baseline deviation windows expand severely under resource constraint simulations.

To eliminate single sample measurement noise and mitigate frequency scaling distortion, the calibration design

integrates an automated multi run median window filtering strategy. The architecture isolates a single computing thread to execute multiple independent calibration windows consisting of five separate trials lasting 1.0 second each:

```
def calibrate_cpu_median(duration=1.0,
bits=2048, trials=5):
    samples =
[calibrate_cpu(duration=duration,
bits=bits) for _ in range(trials)]
    samples.sort()
    return samples[len(samples) // 2]
```

Extracting the median throughput value provides extreme resilience against outlier performance spikes caused by sudden operating system kernel interrupts while stabilizing the hardware capability factor. For prolonged target lock windows, the calibration duration scales proportionally to the intended temporal delay, capping at a maximum benchmark threshold. This structural extension ensures that the system profiling phase actively reflects the thermal equilibrium and CPU frequency scaling limits that the decapsulation engine will encounter during the sequential computation path.

C. Puzzle Creation

Given the verified calibrated hardware speed S and the target delay duration t, the system calculates the absolute iteration depth $T = \lfloor t \cdot S \rfloor$. The shortcut exponent is evaluated through Python's native three argument ternary `pow` function, which performs fast modular exponentiation:

```
e = pow(2, T, phi)
b = pow(a, e, n)
```

This mathematical setup phase executes in $O(log\, T)$ modular multiplications for the capsule creator. The operation completes in a fraction of a millisecond because the creator possesses explicit knowledge of the secret group order $\phi(n)$, allowing the massive exponent 2^T to be reduced prior to evaluating the final base element state. A solver lacking access to the secret prime trapdoors cannot execute this reduction, forcing the solver's machine to step through every iteration of the sequential dependency chain.

The uniform 256-bit symmetric key K is drawn directly from the operating system cryptographically secure entropy pools via `os.urandom(32)`. The symmetric key is securely masked inside a key protection matrix by computing a bitwise exclusive OR operation against the Secure Hash Algorithm 256-bit digest of the final puzzle state b:

```
C_K = bytes(x ^ y for x, y in zip(K,
SHA256(b_bytes)))
```

The big integer solution b is explicitly converted into its equivalent big endian byte string representation prior to running the hashing routine. Because the hash primitive functions as a secure random oracle, and the numerical value of b remains computationally indistinguishable from a random distribution to any entity operating without access to $\phi(n)$, this encapsulation boundary blocks key leakages. Zero information regarding the attributes of key K or the underlying data payload is exposed until the solver executes all T sequential modular squarings.

D. Sequential Solver and Progress Reporting

To maintain fluid interface performance alongside heavy mathematical calculations, the decapsulation loop is constructed as a stateful Python generator function rather than a synchronous block. This design pattern permits the computational thread to yield incremental performance frames back to the execution context without blocking parent processes:

```
def solve_time_lock_puzzle_generator(n, a,
T, yield_interval_pct=1.0):
    x = a
    yield_step = max(1, int(T *
(yield_interval_pct / 100.0)))
    for i in range(1, T + 1):
        x = pow(x, 2, n)
        if i % yield_step == 0 or i == T:
            yield i, (i / T) * 100.0
    return x
```

The Command Line Interface consumes these generated state tuples within a specialized terminal layout progress context to render a live, dynamic console progress indicator. Concurrently, the web application interface processes the generator updates inside a Flask chunked HTTP response streaming context. This infrastructure transmits JavaScript Object Notation encoded Server Sent Events directly to client browsers at a controlled transmission interval of approximately 80 milliseconds. This performance threshold guarantees highly responsive visual updates on user dashboards while preventing browser rendering pipelines from becoming choked with excessive transmission events during brief, low iteration solving paths.

E. Ciphertext Decryption and Authenticated Integrity Verification

Upon successful completion of the sequential squaring loop, the decapsulation engine captures the final state x_T as the verified puzzle solution b. The encapsulated symmetric key is unmasked via a secondary bitwise exclusive OR operation, formulated as $K = C_K \oplus SHA256(b)$. This derived key string is injected alongside the unique 12-byte initialization vector and the 16-byte authentication tag into the Advanced Encryption Standard in Galois Counter Mode (AES-256-GCM) decryption core.

If the calculated value b is incorrect due to a premature loop termination or a data corruption event, the Galois Message Authentication Code verification protocol detects the integrity failure instantly and raises a specific validation error. This authenticated encryption configuration ensures that partial computational progress across the time lock puzzle path yields zero information regarding the plaintext attributes,

maintaining absolute data privacy against active manipulation attacks.

F. Remediation of Identified Code-Level Vulnerabilities

During a rigorous system code audit and implementation review, two significant engineering vulnerabilities were identified within the repository infrastructure. While these flaws do not invalidate the underlying cryptographic design, remediating them is essential for optimal performance and strict security hygiene.

The first vulnerability involves a redundant double solve bug located within the command line module (`cli/commands.py`). The original command orchestration script invoked the sequential solver generator twice: once inside the terminal layout block to update the graphical progress indicator, and a second time in an isolated loop to extract the final `StopIteration` value. This structural oversight forced the host processor to repeat the entire chain of T sequential modular squarings, doubling the wall clock duration required to unlock the capsule. To eliminate this processing overhead, the command logic was redesigned to capture the returned solution big integer from the exception context within a single, unified loop execution flow:

```
solver =
solve_time_lock_puzzle_generator(n, a, T,
yield_interval_pct=0.5)
while True:
    try:
        current_i, pct = next(solver)
        progress.update(task,
completed=current_i)
    except StopIteration as e:
        b = e.value
        break
```

The second vulnerability involves the selection of the public starting base element a inside the core time lock file (`core/tlp.py`). The original codebase utilized the standard `random.randint(2, n - 2)` function, which relies on the non cryptographic Mersenne Twister pseudo random number generator algorithm. Because the internal state of a Mersenne Twister engine can be completely mapped after observing a limited sequence of outputs, the generated base values were technically predictable.

Although a represents a public parameter and predictability does not directly expose the secret key, cryptographic engineering standards demand that every parameter entering a security protocol must be backed by a cryptographically secure entropy source. To close this vulnerability, the selection routine was refactored to extract bytes directly from the operating system entropy pool, reducing the resulting big integer modulo the public boundaries:

```
a =
int.from_bytes(os.urandom((n.bit_length()
// 8) + 1), 'big') % (n - 3) + 2
```

Integrating this secure base generator alongside the multi run median window calibration architecture resolves the identified implementation bugs, ensuring that the software deployment achieves the theoretical reliability and security bounds required for decentralized digital time capsule infrastructures.

V. Empirical Evaluation

A. Experimental Setup

All benchmarks were evaluated on a laptop equipped with an Intel Core processor (12th Generation, Alder Lake family) with a base clock speed of approximately 1.3 GHz and 16 GB RAM, running Windows 11 Home Single Language. PyCryptodome was used for RSA prime generation and AES-256-GCM operations, and no FPGA, GPU, or other hardware acceleration was employed at any stage. The underlying multi-precision integer operations in Python execute via the C-based GMP library, which substantially accelerates large integer arithmetic compared to a pure Python implementation. Each benchmark was run as a single-threaded process, consistent with the inherently sequential nature of the TLP solver, since parallelizing the squaring chain itself would violate the hardness assumption the scheme relies on.

It is worth noting that mobile and ultra-low-power processor designs such as this one are built to prioritize power efficiency and thermal headroom over sustained peak frequency, relying heavily on dynamic boost behavior for short bursts of computation before throttling back toward the base clock under prolonged load. This characteristic is directly relevant to the calibration accuracy results discussed in Section V-C, since a short calibration window is more likely to capture an elevated boost-clock throughput that the processor cannot sustain across a longer solve.

B. Benchmark 1: Modulus Size vs. Computation Speed

The first benchmark measures the number of modular squarings achievable per second across four RSA modulus sizes, 512, 1024, 2048, and 4096 bits, using `calibrate_cpu` with a 1.5 second measurement window, averaged over three independent runs. Results are summarized in Table II.

TABLE II. Modulus Size vs.Squaring Speed

Modulus (Bits)	Squaring Speed (ops/s)	*Relative Performance*
512	619,976.43	8.68×
1024	240,018.15	3.36×
2048	71,375.23	1.00× (Baseline)
4096	20,167.74	0.28×

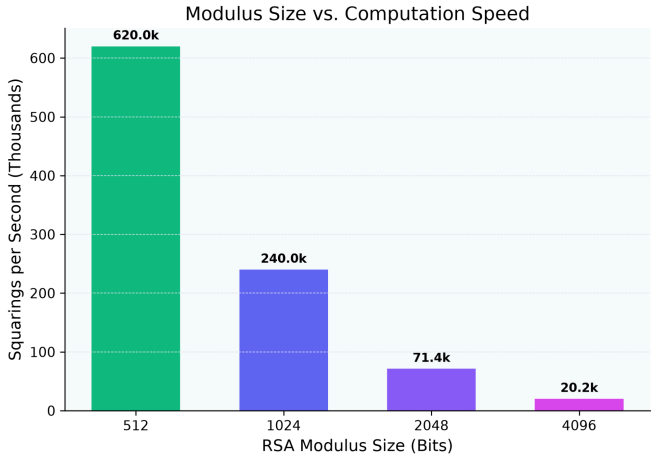

Fig. 3. Modulus Size vs. Computation Speed

The data shows a pronounced inverse relationship between modulus size and computation speed. Moving from a 512-bit modulus to a 4096-bit modulus produces a 30.7× reduction in throughput, which is consistent with the cost of modular multiplication scaling with the bit length of the operands, roughly $O(L^2)$ under schoolbook multiplication or $O(L^{1.58})$ under Karatsuba multiplication, both of which place 4096-bit squarings well above the cost of 512-bit squarings even before accounting for the doubled number of limbs involved.

Examining adjacent size classes makes the scaling trend clearer: doubling the modulus from 1024 to 2048 bits reduces throughput to roughly $71,375 / 240,018 \approx 0.30\times$, and doubling again from 2048 to 4096 bits reduces it further to roughly $20,168 / 71,375 \approx 0.28\times$. Both ratios sit close to the $0.25\times$ expected under a purely quartic relationship between key length and squaring cost, which is the behavior anticipated when total operand bit length doubles for each prime factor simultaneously.

This relationship carries a direct practical consequence for puzzle design. Small moduli such as 512 or 1024 bits offer high squaring throughput and therefore finer-grained timing resolution, but they are no longer cryptographically defensible; a 512-bit RSA modulus can today be factored within hours using commercial cloud compute, which would let a solver recover $\phi(n)$ and bypass the sequential computation entirely. A 2048-bit modulus, achieving roughly 7.14×10^4 squarings per second on the tested hardware, represents the practical minimum for genuine security while still supporting sub-minute lock durations, whereas 4096-bit moduli trade a further 3.5× reduction in speed for additional long-term security margin, making them more suitable for capsules intended to remain locked for extended periods.

C. Benchmark 2: Calibration Accuracy

The second benchmark evaluates how closely the calibrated iteration count T translates a target lock duration into an actual solve time. Puzzles were created at a fixed 2048-bit modulus with target durations of 2, 5, 10, 20, and 30 seconds, and the resulting solve time was measured directly. Results are summarized in Table III.

TABLE III. TARGET VS. ACTUAL SOLVE TIMES (2048-BIT)

Target (s)	Actual (s)	Error (s)	Error (%)	Squarings (T)
2.0	2.10	+0.10	5.03	142,750
5.0	7.25	+2.25	45.07	356,876
10.0	19.90	+9.90	99.01	713,752
20.0	38.49	+18.49	92.44	1,427,504
30.0	50.49	+20.49	68.31	2,141,257

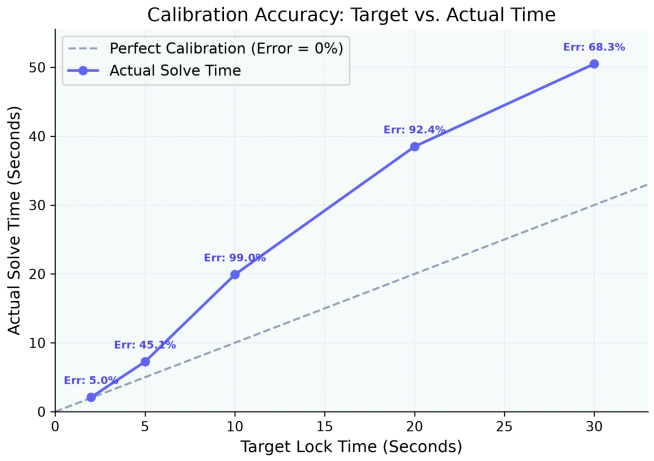

Fig. 4. Calibration Accuracy

A clear and systematic positive bias emerges across all five target durations: actual solve times consistently exceed their targets, and the error grows substantially before partially contracting again at the longest duration tested, peaking at nearly 99% for the 10 second target before settling to 68% at 30 seconds. This pattern points toward two compounding effects rather than a single cause. The first is interpreter and generator overhead because the solver yields progress at regular intervals, each yield introduces a context switch and percentage calculation that the raw calibration loop never performs, so the calibrated speed S is measured under conditions slightly faster than what the solver actually achieves in practice. The second, and likely dominant, effect is CPU frequency scaling and thermal behavior: during the short 1.5 second calibration window the processor can sustain a boosted clock frequency, producing an inflated estimate of S, whereas over a sustained 10 to 30 second computation, thermal management and increased likelihood of OS scheduler interruptions pull the sustained clock rate downward, causing the actual solve to fall behind the optimistic calibration estimate.

The partial decline in error percentage between the 10 second and 30 second targets, despite both running under sustained load, suggests this is not a purely monotonic thermal effect but may also reflect measurement noise or transient scheduling artifacts at the 10 second mark specifically, an open question that would benefit from repeated trials with confidence intervals in future work. Taken together, these results motivate the median-window and duration-scaled calibration strategies discussed in Section IV-B as a practical path toward tightening this accuracy gap.

D. Benchmark 3: Throttling Simulation

The third benchmark simulates solving under constrained computational resources, representative of conditions such as mobile devices, browser-based JavaScript execution, or battery-saver power profiles. A 5 second target puzzle at 2048-bit modulus was solved under four simulated CPU allocations, 100%, 75%, 50%, and 25%, implemented by introducing artificial sleep delays between batches of squarings calibrated to produce the target effective throughput. Results are summarized in Table IV.

TABLE IV. CPU THROTTLING IMPACT ON SOLVE TIME

CPU Allocation (%)	Solve Time (s)	Relative Speed
100	8.74	1.00
75	12.55	0.70
50	19.64	0.44
25	37.47	0.23

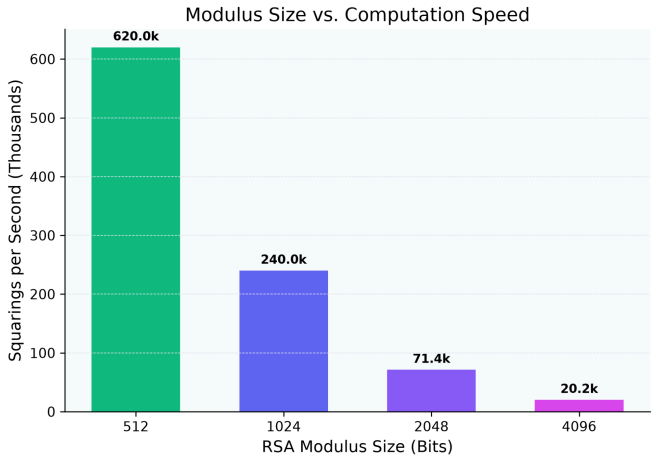

Fig. 5. Throttling Simulation

The results confirm the expected behavior almost exactly to solve time scales inversely with the proportion of CPU resources allocated to the solving thread. At 25% allocation, the solve time rises to 37.47 seconds, a 4.29× increase over the unconstrained baseline of 8.74 seconds, closely tracking the theoretical 4× ratio implied by the throttling level, with the small remaining deviation plausibly attributable to OS scheduler granularity in how the artificial sleep delays are dispatched rather than to any flaw in the underlying model.

This behavior carries an important implication for how the scheme handles adversarial or resource-constrained solvers. If a solver runs the puzzle on slower hardware or under intentional throttling, whether imposed externally or self-imposed, the capsule simply takes longer to open rather than opening early or failing outright; the time-lock property degrades gracefully rather than catastrophically. In this sense the throttling sensitivity demonstrated here is not a weakness in the implementation but a direct and desirable consequence of the sequential squaring assumption itself, since it confirms that no shortcut exists for converting reduced CPU allocation back into the original solve time.

E. File Encryption Experiment

Beyond raw squaring performance, the system was also evaluated on its ability to handle binary file payloads rather than plain text messages. As a representative test case, a 59.8 KB PDF document was encrypted through the hybrid pipeline by first Base64-encoding the raw file bytes and then passing the encoded string into the existing AES-256-GCM encryption stage exactly as a text message would be handled. The capsule was sealed at a 2048-bit modulus with a 10 second target lock duration, which the system translated into 165,587 required squarings based on a measured calibration speed of 16,559 squarings per second.

Upon solving, the actual solve time was measured at 5.70 seconds against the 10 second target, corresponding to a solve speed of 29,036 squarings per second and an accuracy rate of 57.0%. This particular result illustrates the inverse of the positive bias discussed in Section V-C, rather than overshooting the target duration, the puzzle was solved roughly 43% faster than intended. This is consistent with the calibration overestimation problem discussed earlier, but in this instance manifesting as the solving machine outperforming the calibration measurement taken moments earlier, likely due to CPU boost clock variability between the brief calibration window and the subsequent solve. This reinforces that calibration accuracy is sensitive not only to calibration window length but also to transient system load at the moment of measurement.

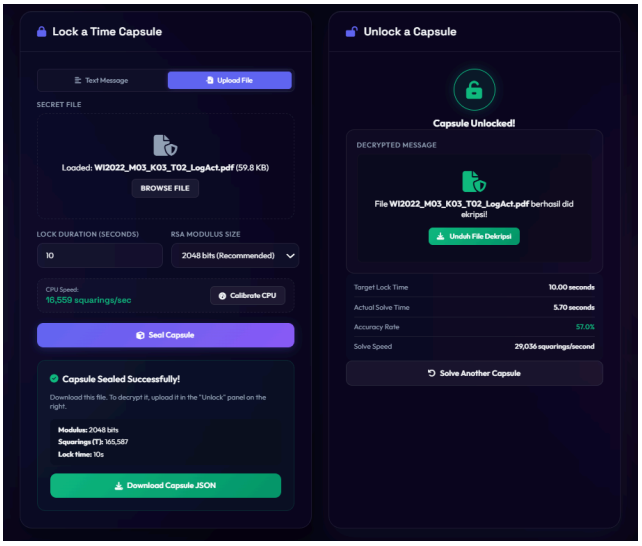

Fig. 6. Lock & Unlock Panel

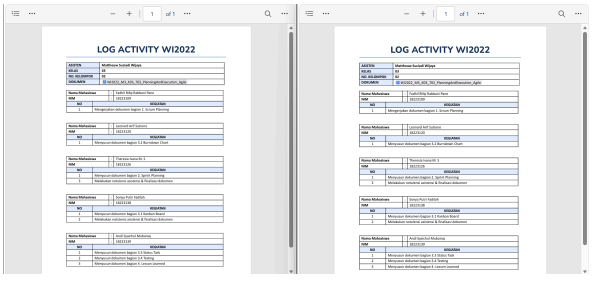

Fig. 7. Original (right) vs. Decrypted (left) PDF Comparison

Despite this timing deviation, the file payload itself was recovered correctly and without corruption. A direct visual comparison between the original document and the decrypted output (Figure 7) confirms that all tabular content, formatting, and text were reconstructed identically across every page. This is further corroborated at the file-system level (Figure 8), where the decrypted output matches the original source file's size exactly (60 KB), with only the file timestamp differing, confirming that the AES-256-GCM authentication tag verified successfully and that the Base64 round-trip introduced no data loss whatsoever.

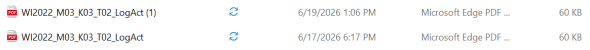

Fig. 8. FileIntegrityMetadata

This result confirms the architectural property described in Section III-D, the TLP component only ever wraps a fixed 32-byte AES key K, never the payload itself, so the number of required squarings T and the resulting solve time depend solely on the target lock duration and modulus size, not on whether the encrypted content is a short text message or a binary document. The only payload-size-sensitive component of the system is the AES-256-GCM encryption and decryption step, which operates at memory-bound throughput far exceeding the sequential squaring bottleneck and therefore contributed negligible overhead even for the file-based payload tested here.

This separation of concerns, a constant-cost temporal lock combined with a payload-size-scaling but comparatively fast symmetric cipher, makes the scheme practical for bulk file storage use cases such as locking documents, archives, or media files without incurring any additional delay beyond what a text-only capsule of the same target duration would require.

VI. Security Analysis

A. Hardness and Cryptographic Guarantees

The security of the entire time-lock mechanism rests on the Sequential Squaring Assumption: for a randomly chosen RSA modulus n of sufficient bit length, no probabilistic polynomial-time algorithm is known that can compute $a^{2^T} \pmod n$ significantly faster than performing T sequential squarings, provided $\phi(n)$ remains unknown to the solver. This assumption is closely related to, but not identical with, the classical RSA assumption and the integer factoring assumption, since factoring N is the only known shortcut around the puzzle entirely. If an adversary could factor N into its prime components p and q, they could compute $\phi(n) = (p-1)(q-1)$ directly and reduce the exponent 2^T modulo $\phi(n)$ to recover b in $O(\log T)$ operations instead of T sequential steps, collapsing the time-lock guarantee almost instantly. The best known general-purpose factoring algorithm, the General Number Field Sieve, has subexponential time complexity, and to align with current cryptographic guidance the implementation defaults to modulus sizes of 2048 bits or larger, consistent with NIST SP 800-57 recommendations, with no sub-quadratic-in-T attack currently known against moduli of this size.

Building on this foundation, the AES-256 key K is never stored directly; it is masked through a simple XOR construction, $C_K = K \oplus H(b)$, where H is SHA-256 applied to the byte representation of the hard-puzzle output b. Because SHA-256 is modeled as a random oracle and b is computationally indistinguishable from a uniformly random value to anyone who has not completed all T squarings, C_K reveals strictly zero information about K prior to the solver recovering b. AES-256-GCM additionally provides authenticated encryption on top of this key, meaning any modification to the ciphertext, nonce, or authentication tag is detected with probability extremely close to 1, formally bounded by $P(False\ Acceptance) \leq 2^{-128}$. Together, these two layers ensure that an adversary has no practical path to recovering the message early, nor to forging or tampering with a sealed capsule, without either completing the full sequential computation or breaking AES-GCM's authentication directly.

B. Parallel and Hardware Resistance

A central design property of the Sequential Squaring Assumption is its explicit resistance to parallelization. Because each squaring step $x_i = x_{i-1}^2 (mod\ n)$ depends entirely on the output of the immediately preceding step, the T iterations cannot be partitioned across multiple cores, threads, or machines and recombined to shorten the total computation. Adding more general-purpose compute power, whether through multi-core CPUs, distributed clusters, or cloud-scale parallelism, provides essentially no speedup over a single fast sequential thread, since the dependency chain itself cannot be broken without knowledge of $\phi(n)$. This distinguishes time-lock puzzles from embarrassingly parallel problems such as hash-based proof-of-work, where horizontal scaling yields near-linear speedups; here, throwing a large compute cluster at the problem gains an adversary essentially nothing.

What parallelism cannot achieve, however, specialized hardware can partially erode at the level of a single squaring step. General-purpose consumer CPUs perform modular squaring using general arithmetic units constrained by clock speeds typically capping in the 4-5 GHz range, alongside instruction overhead unrelated to the squaring itself. An Application-Specific Integrated Circuit or Field-Programmable Gate Array built specifically for large-integer modular multiplication can implement optimizations such as pipelined carry-save adders and dedicated multiplier arrays unavailable to general-purpose processors, and estimates from related work suggest such hardware could plausibly solve a given puzzle on the order of 5 to 10 times faster than an equivalent consumer CPU. Critically, this is a constant-factor speedup applied uniformly to every squaring step rather than a shortcut bypassing the sequential structure itself; the adversary still computes all T squarings in order, simply faster per step. This represents an acknowledged, inherent limitation of CPU-bound time-lock puzzles in general rather than a flaw specific to this implementation, and it is one of the central motivations behind Verifiable Delay Functions, which are typically constructed over groups chosen to resist efficient hardware specialization

while also providing a succinct proof of correct computation that a plain TLP does not offer. Where precision against well-resourced adversaries is critical, applying a generous safety margin to T or migrating to a VDF-based construction are both practical mitigations.

Two further limitations round out this picture without undermining the core guarantee. First, the serialized capsule JSON stores `T`, `ops_per_second`, `bits`, and `target_duration` in plaintext, allowing an observer to infer roughly when a capsule was created, what hardware calibrated it, and how long its intended lock period is, simply by inspecting the file. Second, once a capsule has been solved and K recovered, the message becomes permanently readable; the scheme provides no forward secrecy or subsequent revocation, which is an intentional property of a delay primitive rather than an oversight, but worth noting for applications that expect content to expire after a window of availability.

C. Randomness and Code-Level Fixes

Cryptographic randomness was reviewed across every component drawing random values. The AES-256 key K is generated via `os.urandom(32)`, sourcing directly from the OS CSPRNG, and RSA prime generation relies on PyCryptodome's `getPrime`, which internally applies a CSPRNG-seeded Miller–Rabin test; both are appropriate for cryptographic use without modification. The one component flagged during review was the puzzle base a, originally selected via Python's `random.randint(2, n-2)`, which draws from the Mersenne Twister PRNG rather than a CSPRNG. Since Mersenne Twister output is, in principle, predictable given sufficient observed state, this fell short of best practice, even though a is a public value whose only role is to serve as a non-trivial starting point for the squaring chain, meaning its predictability does not directly expose K. To close this gap regardless, base selection was replaced with a CSPRNG-backed routine drawing from `os.urandom`:

```
def secure_randint(min_val, max_val):
    range_val = max_val - min_val + 1
    num_bytes = (range_val.bit_length() +
7) // 8
    while True:
        rand_bytes = os.urandom(num_bytes)
        val = int.from_bytes(rand_bytes,
byteorder='big')
        if val < range_val:
            return min_val + val
```

A broader code-level audit surfaced two further issues distinct from the cryptographic design itself. The CLI's `run_solve` command in `cli/commands.py` was found to execute the solver generator twice in sequence, once to drive the visible progress bar and a second time purely to retrieve the generator's final return value, needlessly doubling actual solve time. This was corrected by capturing the return value within the same generator pass used for progress display. Separately, the single-sample calibration variance discussed in Section IV-B, where estimating CPU speed from one short calibration run yields high variance, was mitigated by averaging the measured squaring speed across multiple short runs rather than relying on a single sample, meaningfully reducing the timing noise carried forward into the final calculation of T.

VII. Discussion and Future Work

A. Calibration Accuracy Improvement

The most pressing practical limitation surfaced throughout this work is the calibration accuracy degradation observed at longer lock durations in Section V-C. The root cause appears to trace back to CPU frequency scaling: a short 1.5 second calibration window is well-positioned to capture a processor's boosted clock throughput, but is not representative of the sustained throughput available once thermal management and prolonged load pull the clock rate back down over tens of seconds. Two complementary approaches could address this gap. The first is adaptive calibration, where the calibration window itself scales proportionally to the target lock duration, for instance running for roughly 5 to 10 percent of the intended target time rather than a fixed 1.5 seconds, so that the conditions under which speed is measured more closely resemble the conditions under which the puzzle will actually be solved. The second is historical speed correction, where accumulated accuracy data from past puzzle solves on similar hardware is used to learn a correction factor, applied as a multiplier to the calibrated speed before T is computed, gradually tightening the gap between target and actual solve time as more data is collected.

B. Verifiable Delay Functions

A more fundamental limitation, distinct from calibration tuning, is the complete absence of any verifiability guarantee in the current construction. Once a solver completes the T squarings and recovers b, there is no way for a third party to confirm that this computation was actually performed correctly, or performed at all under the expected sequential constraints, without independently repeating the entire chain of squarings themselves. This is acceptable for the present system's intended use case, where the solver is also the only party who needs the result, but it becomes a meaningful barrier for any application requiring external verification of solve completeness.

Verifiable Delay Functions (VDFs) address exactly this gap. As formalized by Boneh et al. (2018) [6], a VDF is constructed over a group of unknown order and produces, alongside the puzzle's output, a succinct cryptographic proof that the output was computed through the required number of sequential steps. Crucially, this proof can be checked by any third party in time far shorter than the original computation itself, without that party needing to redo the squaring chain. Two practical proof schemes exist for this purpose: Wesolowski's construction, which produces a single compact proof verifiable with one additional group exponentiation, and Pietrzak's construction, which instead uses a recursive halving proof structure with a different set of efficiency tradeoffs. Extending the present implementation to emit a Wesolowski- or Pietrzak-style proof alongside the recovered b value would transform the system from a private, self-verifying puzzle into one whose correctness can be trustlessly confirmed by any observer, opening the door to the blockchain applications discussed next.

C. Blockchain and Public State Machine Integration

The digital time capsule primitive maps naturally onto blockchain-based applications, precisely because a public blockchain functions as a decentralized, universally observable state machine capable of enforcing exactly the kind of condition a time-lock puzzle expresses: an action that cannot occur before a certain amount of verifiable sequential work has been completed. Several concrete integration patterns follow directly from combining the present TLP construction with the VDF extension discussed above [1].

The most direct pattern stores only a commitment to the capsule, such as a hash of its public parameters, on-chain, while the full capsule payload remains off-chain. A smart contract acting as verifier would then accept a submitted b value together with its VDF proof, check the proof in constant or near-constant time using on-chain computation, and only upon successful verification trigger a programmed action, such as releasing escrowed funds, unlocking a governance vote, or revealing a committed value to other contract participants. Because the smart contract never needs to perform the T squarings itself, only verify the proof, this pattern remains computationally cheap regardless of how long the underlying lock duration was, making time-delayed token releases, sealed-bid auctions with delayed reveal, and decentralized timestamping all realistic applications once the verifiability gap from Section VII-B is closed. Without VDF integration first, however, blockchain deployment would either require the smart contract to redundantly re-execute the entire squaring chain on-chain, which is computationally prohibitive given typical gas cost constraints, or fall back to trusting an off-chain oracle to attest to the solve, reintroducing exactly the kind of trusted third party the time-lock construction was designed to eliminate in the first place [2].

D. Alternative Hardness Assumptions and Long-Term Security

Beyond verifiability and on-chain integration, the choice of RSA [3] as the underlying hard-group construction is itself open to reconsideration. Groups of unknown order are not unique to RSA moduli; imaginary quadratic class groups, as used in Chia Network's VDF implementation, and certain elliptic curve constructions offer alternative algebraic settings with potentially different security and performance characteristics, including in some cases the ability to use significantly smaller parameter sizes for equivalent security margins, which would directly improve the calibration granularity problem discussed in Section VII-A for short lock durations.

A more pressing long-term consideration is post-quantum security. Because the present scheme's hardness ultimately reduces to the difficulty of factoring N, a sufficiently capable quantum computer running Shor's algorithm would recover $\phi(n)$ efficiently, collapsing the time-lock guarantee entirely regardless of modulus size. While such hardware remains outside current practical reach, any deployment intended to remain secure across multi-decade time horizons should treat this as a real constraint rather than a theoretical curiosity, and should track the broader literature on post-quantum-secure sequential computation as it matures, with class-group and isogeny-based constructions representing two of the more promising directions currently under active research [2].

VIII. Conclusion

This paper presented a complete implementation and empirical analysis of a Digital Time Capsule system built on the RSA Time-Lock Puzzle construction combined with AES-256-GCM encryption. The system fulfills its core design goal: messages, including binary file payloads such as PDF, DOCX, JPG, and PNG, are encrypted such that they cannot be decrypted before a minimum wall-clock time has elapsed, without relying on any trusted third party. The hybrid construction proved both efficient, with puzzle creation costing only $O(log\ T)$ for the creator regardless of how large the target duration is, and secure, conditioned on the Sequential Squaring Assumption and the authenticated-encryption guarantees of AES-256-GCM, with both interfaces, CLI and web, providing real-time solving feedback through a shared generator-based core.

Empirical benchmarks confirmed the expected inverse relationship between modulus size and squaring speed, with 2048-bit moduli achieving roughly 71,000 squarings per second on the tested hardware, consistent with the $O(b^2)$ complexity of modular multiplication. Calibration accuracy experiments revealed a systematic positive timing bias that grows with lock duration, most plausibly attributable to CPU boost-clock behavior during the short calibration window failing to represent sustained throughput over longer solves, while throttling simulations confirmed that solve time scales inversely with available CPU allocation, exactly as the underlying hardness assumption predicts. Together, these results validate the system's core mechanics while clearly identifying calibration precision as the primary practical limitation requiring further refinement.

The production implementation delivers a multi interface, audited, and openly accessible reference platform for an important yet historically underdeployed temporal cryptographic primitive. This architecture serves as a solid foundation for future development paths focusing on adaptive multi run calibration filters, Verifiable Delay Function scaling, and blockchain integrated time locked commitments, extending decentralized temporal data protection toward execution settings that demand both strict predictability and trustless verifiability.

Video Link at Youtube

https://youtu.be/qYLaZ3p0Clk

Link GitHub

https://github.com/meerancor33/digital-time-capsule.git

Acknowledgment

The author would like to express sincere gratitude to Prof. Dr. Ir. Rinaldi, M.T. for his invaluable guidance and insightful lectures throughout the II4021 Cryptography course, which provided the foundation for this work. The author also extends appreciation to fellow II4021 Cryptography classmates and teaching assistants for their support throughout the semester.

References

[1] Rivest, R. L., Adleman, L., & Shamir, A. (1978). A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2), 120–126.

[2] Rivest, R. L., Shamir, A., & Wagner, D. A. (1996). Time-lock puzzles and timed-release crypto (Technical Report MIT/LCS/TR-684). Massachusetts Institute of Technology.

[3] Dworkin, M. (2007). Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. NIST Special Publication 800-38D. National Institute of Standards and Technology.

[4] Boneh, D., & Naor, M. (2000). Timed commitments. In Advances in Cryptology – CRYPTO 2000 (pp. 236–254). Springer.

[5] Mahmoody, M., Moran, T., & Vadhan, S. (2013). Publicly verifiable proofs of sequential work. In Innovations in Theoretical Computer Science (ITCS) (pp. 373–388). ACM.

[6] Boneh, D., Bonneau, J., Bünz, B., & Fisch, B. (2018). Verifiable delay functions. In Advances in Cryptology – CRYPTO 2018 (pp. 757–788). Springer.

[7] McGrew, D. A., & Viega, J. (2004). The Galois/Counter Mode of Operation (GCM). National Institute of Standards and Technology.

STATEMENT

Hereby, I declare that the research paper titled “Digital Time Capsule: Implementation and Analysis of RSA Time-Lock Puzzle with AES-256-GCM Encryption” is my own original work. It does not contain any materials previously published or written by another person, except where due reference is made.

Bandung, 18 Juni 2026

Theresia Ivana M. S
18223126