A Time-Locked Message Capsule Platform Based on Threshold Networks

Antoine Karam
Department of Computer Science
American University of Beirut
Beirut, Lebanon

Ghady Youssef
Department of Computer Science
American University of Beirut
Beirut, Lebanon

***Abstract*—Timelock encryption allows a user to encrypt messages to a future specified unlock date. First, we explore previous attempts of timelock encryption that rely on agents and puzzles to achieve the desired functionality. Then, we present a time capsule platform based on the timelock encryption scheme which allows users to encrypt messages to a future specified date. The scheme leverages a threshold network that produces private keys at specified intervals. We use an identity based encryption scheme with round numbers as identities in order to encrypt messages to a future date. We discuss the implementation details of the scheme and platform, the challenges faced, and the practices followed in order to guarantee security. Finally, we explore possible research directions that benefit from the proposed primitive to build innovative applications such as secure online voting, sealed bid auctions, and the possibility of transitioning this scheme to use post quantum secure algorithms.**

***Index Terms*—Time-lock encryption, BLS signatures, Identity-based encryption, Distributed verifiable randomness**

I. Introduction

Timelock encryption refers to encrypting a message or plaintext to the *future*. This means that the encryptor of the message would specify an *unlock date* after which the ciphertext can be than automatically be decrypted. Before the unlock date, no one and even the encryptor should not be able to recover the plaintext before the specified date has passed. This would be theoretically implemented by releasing the key on the specified unlock date. Concretely, problems arise when we try to implement such a scheme. We might ask the following questions:

- Who should be responsible for releasing the key at the unlock date?
- How can we guarantee that the ciphertext will not be decrypted before the unlock date?
- How can we enforce the concept of time on a traditional encryption scheme?

Primary incentives drive the research towards the development of both *practical*, *reliable* and *secure* timelock encryption scheme. We note some of them:

- **Secure Electronic Voting:** Digital voting has been a challenge for researchers and governments. Solving this problem would allow a greater number of citizens to participate in the voting procedure. However, solving this problem must not trade-off nationwide security. It should be trivial as to why we wouldn't want a corrupt or malicious voting platform, which could compromise votes, rig elections; which could ultimately directly affect the future of a country and its citizens. Timelock encryption could contribute to this field by enabling the secure transmission of the electors' votes and guarantee that those votes are not released before the end of the voting process. We can think of it similar to when the votes are stored in the ballots and no one is allowed to check them before the end of the elections process. More challenges remain but are not addressed by this primitive.
- **Sealed-bid auctions:** Similar to secure online voting, sealed-bid auctions could benefit from timelock encryption by storing the encrypted bids until a specified unlock time which would ensure fairness across all bidders (by ensuring that bids are submitted anonymously and that nobody could decrypt them before the end of the auction).
- **Transfer of assets:** We could use timelock encryption in order to freeze some assets to a future specified time, preventing their release before some transactions or funds have been transferred.
- **Preservation of legal documents:** Legal or government documents can be encrypted to remain hidden from the public until a specific date.

II. Background

This section covers some mathematical and cryptographic primitives that were used throughout this project.

A. Elliptic Curves

An elliptic curve is a curve on a plane with a pair of (x, y) coordinates. A curve's equation defines all the points that belong to that curve. In cryptography, curves typically follow the Weierstrass form:

$$y^2 = x^3 + ax + b \tag{1}$$

We typically use curves that range over discrete values rather than continuous values, since we would use them to generate keys. Elliptic curves are typically defined over finite fields of prime order q. For example $\mathbb{Z}_q$, the set of integers modulo q. The security of such curves in the context of cryptography relies on the order or number of points in the

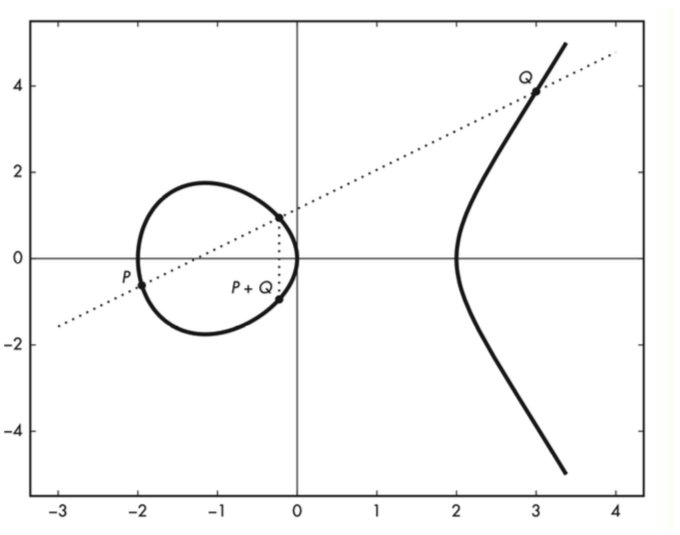

Fig. 1. Adding two points on an elliptic curve. (Source: Serious Cryptography, Applied Cryptography course slides)

field, the greater its security since it makes it harder to guess the number (or key) that was used.

Operations on the curve are defined on the points that belong to the curve. For example, we could add two points of the curve, and the result would be a new point on the curve. Consider two points $P, Q \in E$ where E is the set of points that satisfy the curve's equation. $P + Q$ is a point that is also in E, see Figure 1.

B. Bilinear Maps and Pairings

Pairing are maps that take a point $P \in \mathbb{G}_1$ and another $Q \in \mathbb{G}_2$ and return a point $X \in \mathbb{G}_T$. Bilinear maps have the following structure, by defining the pairing e:

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T \tag{2}$$

These bilinear maps have the following special properties:

- $e(P, Q + R) = e(P, Q) \cdot e(P, R)$
- $e(P + S, Q) = e(P, Q) \cdot e(S, Q)$

and from those we have the following equalities that hold:

$$\begin{aligned} e(aG_1, bG_2) &= e(G_1, bG_2)^a \\ &= e(G_1, G_2)^{ab} \\ &= e(G_1, aG_2)^b \\ &= e(bG_1, aG_2) \end{aligned}$$

It is not necessary to dive deep into the proof of how these work, but it is necessary to note, in order to understand the constructs that will follow.

C. Hash to Curve

In order to compute the signature of a message or plaintext, we must first transform this message to a point on the elliptic curve we are using. We will not go into the details of this transformation, but it is important to note that this transformation is needed in order to use the message as a point on the curve and compute the signature through curve operations.

D. BLS signatures

BLS signatures typically use bilinear pairings in order to sign and verify signatures. Private and public keys are defined over the two groups $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively (up to swap). We can compute the private key s by randomly sampling a point from $\mathbb{G}_1$. Consider $G_1 \in \mathbb{G}_1$ and $G_2 \in \mathbb{G}_2$ which are both generators of their respective groups. The public key would be obtained by multiplying the private key by the generator $G_1 \in \mathbb{G}_1$ where $p = s \cdot G_1$. Computing the signature of a plaintext pt would first consist of mapping its value to a valid point on the curve of the respective group (using a hash to curve function) and then compute $\sigma = p \cdot H(pt)$ which is a standard elliptic curve operation.

In order to verify the signature, we should verify that $e(G_1, \sigma) = e(p, H(m))$, we can validate this claim by the properties of the bilinear pairings.

$$\begin{aligned} e(p, H(m)) &= e(s \cdot G_1, H(pt)) \\ &= e(G_1, H(pt))^s \\ &= e(G_1, s \cdot H(pt)) \\ &= e(G_1, \sigma) \end{aligned}$$

E. Finite, Prime and Extension Fields

Fields are sets on which addition, subtraction, multiplication and division are defined. Finite fields are fields that have a finite number of elements. For example, the set of integers $\mathbb{Z}_{\shortparallel}$ modulo q, where q is a prime number $F_q = \{0, 1, \ldots, q - 1\}$. Extensions of a field typically allow some operations not possible in field F_1 possible in field F_2 which extends the latter. A trivial example would be the set of real numbers $\mathbb{R}$ and complex numbers $\mathbb{C}$. For example, $x^2 - 1 = 0$ has no solution in $\mathbb{R}$ because $x^2 \geq 0$ and this $\forall x \in \mathbb{R}$. The introduction of the imaginary number $i^2 = -1$ allows us to solve the aforementioned equation in $\mathbb{C}$. A field extension $\mathbb{F}_{q^2}$ of $\mathbb{F}_q$ would essentially mean that we will need two elements of $\mathbb{F}_q$ or we can refer to it as a quadratic extension of $\mathbb{F}_q$ [7]. It allows us to represent the field elements as polynomials, where adding two elements consists of adding the coefficients of the polynomials. Some operations like multiplication cannot still hold (like in the example of complex numbers), so we introduce some rules that are irreducible in the field we are extending. This will allow us to perform twisting operations by "implementing" large extension fields by a combination of smaller extensions.

We apply this concept in the context of curves over finite fields modulo q, F_q.

F. BLS Curves

BLS12-381 is a standard elliptic curve that has the equation $y^2 = x^3 + 4$. It has some additional properties which make it "pairing friendly" [7]. We will not dive into the exact reasons for the choice of parameters and values for this curve as it is beyond the scope of this project.

A *twist* allows us to transform the points of one curve to another. This is typically done when the fields have high degrees making the computation within the field very complex and expensive. We can reduce operations on $\mathbb{F}_{q^{12}}$ to be performed on $\mathbb{F}_{q^2}$. The BLS12-381 curve's second group $\mathbb{G}_2$ can be reduced from $\mathbb{F}_{q^{12}}$ to $\mathbb{F}_{q^2}$ which will make the computations much more efficient.

The two groups of the pairing will be [7]:

- $\mathbb{G}_1 \subset E(\mathbb{F}_q)$ where $E : y^2 = x^3 + 4$
- $\mathbb{G}_2 \subset E'(\mathbb{F}_{q^{12}})$ where $E : y^2 = x^3 + 4(1 + i)$

Where $(1 + i)$ is the term that was added in order to make the extension in $\mathbb{F}_{q^{12}}$ work properly.

The BLS12-381 has two curves which makes implementation much simpler, instead of sending coordinates of size 48 bytes, we could send only the x coordinate and then we would know which curve to use based on the remaining 3 bits which acts as flags. One bit represent whether it uses the first or second curve, the second represents the point at infinity and the last would represent whether the point is compressed or not. Note that we have 3 bits of flags and 381 bits of coordinates (hence the 381 in the name), for a total of 48 bytes.

G. Distributed Key Generation

Distributed key generation is a secret sharing mechanism in which $\mathcal{P}$ parties contribute to generate a key K. Each party computes $\mathcal{P}_i$ their share of the secret. With any k parties we can recover K. It is important to note that it is not required for the $\mathcal{P}$ parties to be present, only k is enough to recover the key.

H. Identity-Based Encryption

Traditional symmetric and asymmetric encryption primitives would require some sort of key in order to encrypt messages. Symmetric keys would use the same key to both encrypt and decrypt messages. Asymmetric primitives would involve a private key which could be used to decrypt and a public key to encrypt (or the inverse if this is a signature scheme). These keys are ideally indistinguishable from random by an adversary. However, managing these keys could be a tedious task and could make the design of some systems cumbersome. Identity-based encryption (IBE) consists of using well known values that represent a person or organization through their identity. Some trivial values that could represent identities in a system would be: ID, email, unique username or phone number. The chosen identity does not affect the functionality of the scheme. We would say that you would "encrypt to an identity" meaning that for example, Alice could encrypt something to Bob just by knowing Bob's email address. Then, Bob with his private key could decrypt Alice's message.

I. The League of Entropy

The League of Entropy (LoE) [5] is a publicly verifiable decentralized Randomness-as-a-Service which provides its users with fresh randomness at periodic intervals. The network is hosted by over 19 active nodes with 100% up-time since 2020 deployed over five continents by various educational institutions and companies. This distributed nature of the network allows for decentralized trust where its users can confidently depend on its services without worrying that it is fully controlled by some malicious actor, while verfiying its correctness through BLS signatures. The LoE has a current threshold as of the time of writing support up to 11 compromised nodes without affecting its services. For example, this means that even if one node is compromised, this does not affect the availability, reliability or break trust with its users.

III. Related Work

There are previous attempts at implementing timelock encryption and we distinguish two main approaches to tackling this problem:

A. Agent-based Timelock Encryption

As previously mentioned, to achieve the desired functionality of timelock encryption, we would need a manual or automatic way of releasing the private keys of a designated ciphertext exactly at the unlock time specified by the user. In the case of agent-based timelock encryption, this is solved trivially by offloading this task to a trusted third-party or agent which will release the private keys at the specified time. One could naively implement this by sharing the private keys to a message to their friend whom they trust. This poses some critical downsides. (1) we have to trust that our friend does not use the private key to peek at the message by decrypting the ciphertext, (2) we have to hope that our friend truely releases the keys are the designated date (not before and not at a later date) and (3) that they actually release the key.

B. Puzzle-based Timelock Encryption

Another approach is inspired by proof-of-work systems in the sense that we use a puzzle or task in order to enforce the notion of time by requiring decryptors to perform some computations which require effort and time in order to access the plaintext. Puzzle-based timelock encryption uses this concept in order to "guarantee" (to some extent) that accessing the plaintext would not be possible before a certain amount of time due to the computational cost of solving the puzzle. This is similar to the proof-of-work systems imposed in blockchain systems and cryptocurrencies where a user has to compute thousands of hashes in order find the secret value and write the block. In our case, we would use it to enforce a minimum amount of time that the user has to spend in order to access the plaintext. However, this approach has several issues. (1) There are no guarantees that the puzzle would be as hard as it is today in the future. This is due to the continuous hardware and software advancements which could make some puzzles either weak or easier to solve and thus, losing any guarantees on the unlock date. (2) Enforcing time using computationally hard problems implies that a lot of energy, money and other computational resources would be wasted in order to implement such functionality. This is not an efficient nor reliable option that would stay relevant in the future and thus, making it obsolete for such an application where we need to “encrypt to the future”.

C. Agent-based Timelock Encryption

Most recently, a variation of agent-based timelock encryption has been proposed [8] which keeps the simplicity of “trusting someone to release the keys in the future” but addresses its known issues. They used the fact that drand [2] is a network which maps round numbers to verifiable randomness which are released at specific periodic intervals of time. We can they trust this network to release the randomness exactly when the network reaches a designated round number. If we consider the round number as the public key and the signature or randmoness as the private key, we can build a system where we can “encrypt to a round number” and then only be able to decrypt the ciphertext when that round number is actually reached and the randomness is released by the network. Recall that we can easily precompute round numbers ahead of time because the network increments the round number for each period, and that each round number corresponds to a specific timestamp (according to genesis time which indicates the mapping of round number 1 and its relevant timestamp). Technically, this approach uses identity-based encryption with round numbers as “identities” with private keys being the randomness released at each round. This approach addresses the issue of trust since drand is a decentralized network and is not controlled by a single party, neither drand or the encryptor or anyone else is able release the private keys before the designated time (up to a certain threshold of compromised node).

IV. System Design & Architecture

Our timelock library does not encrypt the data itself. It is used to encrypt the symmetric key handled by another library, which in our case is the age [3] framework. This scheme is known as hybrid encryption, which allows us to encrypt arbitrarily sized data while only time-locking a symmetric key.

Ciphertexts are encrypted to both a round number p and the public key pk, such that decryption requires the round private key. The round private key is derived from the master secret key and the round number. This is identity-based encryption where we encrypt and decrypt to an identity, which in our case is the round number.

We define two generators for the elliptic-curve groups G_1 and G_2, and we make use of the bilinear target group G_T, along with four cryptographically secure hash functions:

- $H_1 : \{0,1\}^* \to \mathbb{G}_2$
- $H_2 : \mathbb{G}_T \to \{0,1\}^\ell$
- $H_3 : \{0,1\}^\ell \times \{0,1\}^\ell \to \mathbb{Z}_p^*$
- $H_4 : \{0,1\}^\ell \to \{0,1\}^\ell$

We use separate hash functions H_1, H_2, H_3, and H_4 to explicitly indicate that each one serves a different purpose, even though they all use the same underlying SHA-256 primitive.

For our encryption scheme, we chose to work with an unchained beacon, which means each round is independent of the other. Therefore, the round private key π is defined in G_1, while the master public key P is defined in G_2. As a tradeoff, the signature size is smaller, but the public key is larger because G_2 is defined over an extension field of degree 2 compared to G_1. In other words, the coordinates in G_2 are twice the size of those in G_1.

A. Encryption

Algorithm 1 Encryption algorithm. Source: The original timelock paper [8].

```
procedure ENC($pp, \rho, M$)
    Parse $pp \to (bg, P, H = (H_1, H_2, H_3, H_4))$
    $PK_\rho \leftarrow e(P, H_1(\rho))$                ▷ round public key
    $\sigma \leftarrow \{0,1\}^\ell$                          ▷ nonce
    $r \leftarrow H_3(\sigma, M)$
    $U \leftarrow rG_1$                               ▷ ephemeral public key
    $V \leftarrow \sigma \oplus H_2((PK_\rho)^r)$   ▷ hiding commitment to nonce
    $W \leftarrow M \oplus H_4(\sigma)$                     ▷ one-time-pad
    return $(ct = (U, V, W), \tau = r)$
end procedure
```

In our architecture, the encryption of a message $M \epsilon \{0,1\}^l$ for round number p is performed as follows:

- Compute the round public key using identity-based encryption.
- Generate a nonce σ.
- Hash the nonce together with the message to derive the secret exponent r.
- Compute the ephemeral public key $U = rG_2$.
- Hide the nonce σ in V.
- Encrypt the message M using a one-time pad with the key derived from σ.

We now explain why this works in detail.

a) Round public key: In identity-based encryption, the idea is to take a point in G_1 and a point in G_2 and compute a bilinear pairing to obtain a point in G_T. The master public key P is already in G_2. To convert the round number p into a point in G_1, we use a hash-to-curve function, yielding $H_1(p) \in G_1$. We can then compute the pairing $e(P, H_1(p))$ which forms the round public key. This relation will be reversed during decryption to recover secrets.

b) Nonce generation: A secure random number generator is used to produce a nonce σ.

c) Secret exponent derivation: We derive a value r from the nonce σ and the message M. The value r serves as a secret exponent in this encryption.

d) Ephemeral public key: We compute $U = rG_2$, which serves two purposes. It is used to reconstruct σ during decryption and to verify that the ciphertext has not been tampered with.

e) Hiding the nonce: We calculate $V = \sigma \oplus H_2((PK_p)^r)$. This hides the nonce σ, which can only be recovered once the round private key is known.

f) Message encryption: Finally, we encrypt the message M using a one-time pad with the key derived from the nonce σ, which is itself hidden in V.

B. Decryption

Algorithm 2 Decryption algorithm. Source: The original timelock paper [8].

```
procedure DEC($pp, \rho, \pi_\rho, ct_\rho$)
    Parse $ct_\rho \to (U, V, W)$
    $\sigma' \leftarrow V \oplus H_2(e(U, \pi_\rho))$
    $M' \leftarrow W \oplus H_4(\sigma')$
    $r \leftarrow H_3(\sigma', M')$
    if $U = rG_1$ then
        return $M'$
    else
        return $\perp$
    end if
end procedure
```

Given a ciphertext ct_p and the round key π_p, decryption happens as follows:

- Recompute σ'
- Recover the message M'
- Recompute r from σ' and M'.
- Verify that $U' = rG_2$ equals U to ensure the ciphertext has not been tampered with.

a) Recovering σ': First, we recompute $\sigma' = V \oplus H_2(e(U, \pi_p))$. This formula holds because during encryption we hid σ through $V = \sigma \oplus H_2((PK_p)^r)$. We already have V, so we only need to prove that $H_2((PK_p)^r) = H_2(e(U, \pi_p))$:

$$\begin{aligned}\sigma &= V \oplus H_2((e(U, \pi_p)) \\ &= \sigma \oplus H_2((PK_p)^r) \oplus H_2(e(U, \pi_p)) \\ &= \sigma \oplus H_2(e(P, H_1(p))^r) \oplus H_2(e(U, \pi_p)) \\ &= \sigma \oplus H_2(e(P, H_1(p))^r) \oplus H_2(e(rG_2, sH_1(p))) \\ &= \sigma \oplus H_2(e(sG_2, H_1(p))^r) \oplus H_2(e(rG_2, sH_1(p))) \\ &= \sigma \oplus H_2(e(G_2, H_1(p))^{rs}) \oplus H_2(e(G_2, H_1(p))^{rs}) \\ &= \sigma\end{aligned}$$

where $U = rG_2$, $P = sG_2$ is the master public key in G_2 with secret s, and $\pi_p = sH_1(p)$ is the round private key.

b) Message recovery and verification: Once σ' is recovered, the message is decrypted as $M' = W \oplus H_4(\sigma')$. We recompute $r = H_3(\sigma', M')$ and verify that $U' = rG_2$ equals U. This final check ensures that both σ' and M' have not been modified, providing CCA security.

V. Implementation

The encryption happens client-side to prevent sending the message in plaintext to the server and having the server perform the encryption.

For our encryption algorithm, we use the `noble` [4] library for hashing, elliptic curve operations, and secure random number generation. To handle the secret exponent correctly, we implemented a function which iteratively hashes the nonce and message until a valid scalar is produced. This guarantees that the secret exponent r is always suitable for elliptic curve operations. For the output in G_T, we serialize the $\mathbb{F}_{p^{12}}$ elements by recursively breaking them down into $\mathbb{F}_{p^6}$, $\mathbb{F}_{p^2}$, and $\mathbb{F}_p$ components to ensure a consistent representation, allowing it to be used across implementations or libraries. We also use utility functions to convert big integers into consistent byte representations.

Our encryption scheme is hybrid, meaning that file encryption is delegated to the `age-encryption` [3] library. Our responsibility is to encrypt the symmetric file key used by `age` with our timelock encryption. To achieve this, we implemented the portion of `age` responsible for wrapping the file key in a `Stanza`, allowing us to intercept the key, encrypt it with our timelock scheme, and store it as encrypted metadata alongside the file.

The backend service is written in Rust. The API layer is built using Axum, a lightweight and simple framework that allows us to quickly define routes and handle requests.

We maintain repository wrappers around both S3 and SQLite. Generic data, such as capsule ID, round number, filename, status, and subscription information is stored in SQLite. Encrypted and decrypted file contents are stored in S3, which is highly efficient for file storage. We keep both the relational database and S3 synchronized by updating the capsule state depending on the operation performed.

We also implement a dedicated drand client responsible for communicating with the League of Entropy's drand beacon API. This component fetches round numbers and beacon values, verifies signatures, and maintains an internal cache for performance.

We implement the decrypt job, which runs every second. Its main responsibility is to fetch capsules ready for processing, decrypt them, update states, save the decrypted files, and notify subscribers of the capsule.

The decryption algorithm closely mirrors the encryption process. We use the `ark` [1] library for elliptic curve operations and the `sha2` [6] library for hashing. To handle the secret exponent correctly, we implemented the same function used on the client side. For the output in G_T, we use the serialize compressed function provided by the `ark` library which removes the need for manual serialization like on the client side.

During decryption, we rely on the `age` library to recover the file, but the file key itself is encrypted using our timelock scheme. Therefore, we implemented the portion of `age` responsible for unwrapping the `Stanza` so that we can intercept the encrypted file key, decrypt it, and supply it back to `age` for file decryption.

Our timelock-age wrapper takes an input ciphertext, invokes the `age` decryptor with our custom implementation, and returns the decrypted output, split into the original filename and its corresponding content.

For error handling, we ensured that our service never panics. We implemented custom exceptions, wrapping every third-party exception with our own. All exceptions are propagated to the last layer of the service, where our exception handler intercepts them and, based on the exception type, returns the appropriate response code and message to the user.

VI. Threat Model & Security

We provide end-to-end encryption to ensure that files remain confidential. At no point does the server have access to the plaintext data since all encryption occurs client-side. This guarantees that even a compromised server cannot bypass the encryption.

A. Security Assumptions

Our system assumes the following security properties:

- **Cryptographic primitives:** The underlying elliptic curve, pairing operations, and hash functions used are secure. All primitives used in the codebase are imported from battle-tested libraries. So we rely on both the security of the implementation and the design of those primitives.
- **Threshold beacon security:** We rely on the League of Entropy's drand network for public randomness. The drand network is a threshold BLS system, meaning no single node can independently release round keys. A threshold of nodes must collaborate in order to take control over the release of keys and signatures, if the number of nodes compromised is below the threshold, we can still benefit from a security guarantee. Note that with the current deployment of the network, the feasibility of such an attack is highly impractical, especially if we recall that the nodes are not controlled by a single entity. All signatures fetched from the network are verified on the server to ensure that they are valid points on the elliptic curve.

B. Potential Threats

The primary threats considered are:

- **Compromised server:** Since encryption occurs client-side and file keys are protected with timelock encryption, a compromised server cannot access plaintext data directly. Also, any corrupted ciphertext is detected through consistency checks in our decryption scheme. This prevents decryption of incorrect data but it does not provide a full integrity guarantee. Since ciphertexts are not signed, a compromised server could still delete or produce valid ciphertexts without detection.
- **Compromised drand network:** If an attacker manages to compromise the threshold of the network and manages to take control, they could potentially take control over and release malicious signatures, meaning that they can decrypt messages submitted to the platform.

VII. Conclusion

This paper presented a time capsule platform that allows users to encrypt messages to the future, in a reliable, practical and secure way. We leveraged the drand threshold network which allows us to generate private keys in a future timestamp, which are mapped to a round number. The round numbers act as identities in the identity-based encryption scheme used. We described the design of the system, the implementation choices, and the challenges faced while ensuring security and correctness. This line of work is promising especially for more innovative applications such as secure online voting and sealed-bid auctions.

VIII. Future Work

A. Post-Quantum Migration

Our current implementation relies on elliptic curves and is therefore not post-quantum secure. Both the threshold network and the BLS-based IBE scheme should be replaced with post-quantum alternatives. Research in this direction could help ensure that this idea remains relevant in a post-quantum world.

B. Commitment Schemes

Integrating commitment schemes into our timelock encryption system could provide additional functionality, enabling users to commit to values securely and reveal them later. This could serve as a foundation for more advanced cryptographic protocols.

C. Secure Electronic Voting

Our existing timelock encryption primitive could be used as a building block for a proof-of-concept cryptographically secure electronic voting platform. Electronic voting has been a topic of research for decades, but practical implementation at the national level is challenging due to security issues such as vote forgery, DDoS attacks, and other manipulations. Building a secure voting platform using timelock encryption and commitment schemes could be a step toward a future where citizens can vote online safely, without the risk of interference.

References

[1] arkworks. https://github.com/arkworks-rs. [Accessed 18-11-2025].

[2] Drand. https://www.drand.love. [Accessed 18-11-2025].

[3] GitHub - FiloSottile/typage: A TypeScript implementation of the age file encryption format, available as an npm package or as a bundled .js file., howpublished = https://github.com/filosottile/typage, year = , note = [Accessed 18-11-2025],.

[4] GitHub - paulmillr/noble-curves: Audited & minimal JS implementation of elliptic curve cryptography. https://github.com/paulmillr/noble-curves. [Accessed 18-11-2025].

[5] League of Entropy. https://www.drand.love/loe. [Accessed 18-11-2025].

[6] sha2 - Rust. https://docs.rs/sha2/latest/sha2/. [Accessed 18-11-2025].

[7] Ben Edington. BLS12-381 For The Rest Of Us - HackMD. https://hackmd.io/@benjaminion/bls12-381About-curve-BLS12-381, 2025. [Accessed 18-11-2025].

[8] Nicolas Gailly, Kelsey Melissaris, and Yolan Romailler. tlock: Practical timelock encryption from threshold BLS. Cryptology ePrint Archive, Paper 2023/189, 2023.