Walrus ve Seal Protokolleri ile Merkeziyetsiz Veri ve Gizlilik Mimarisi: Kapsamlı Teknik Referans ve Uygulama Kılavuzu

1. Yönetici Teknik Özeti ve Mimari Vizyon

Web3 ekosistemi, basit varlık transferlerinden karmaşık, veri yoğunluklu uygulamalara doğru evrilirken, altyapı katmanında iki temel darboğaz ile karşılaşmaktadır: büyük ölçekli verilerin zincir üzerinde saklanmasının maliyetsizliği ve şeffaf bir defter üzerinde veri gizliliğinin sağlanması paradoksu. Sui blok zinciri üzerinde inşa edilen **Walrus** (Merkeziyetsiz Depolama) ve **Seal** (Merkeziyetsiz Sır Yönetimi) protokolleri, bu sorunları ele alan birbirini tamamlayıcı iki kritik teknoloji olarak öne çıkmaktadır. Bu rapor, katılımcıları, sistem mimarları ve blok zinciri geliştiricileri için, bu teknolojilerin teorik temellerinden üretim ortamına (production) geçiş süreçlerine kadar uzanan eksiksiz bir teknik uygulama kılavuzu sunmaktadır.

Walrus, yapılandırılmamış büyük verilerin ("blob"lar) depolanması için silme kodlaması (erasure coding) ve Fountain kodlarına dayalı **Red Stuff** algoritmasını kullanarak, veriyi çoğaltmak yerine parçalara ayırıp dağıtarak yüksek erişilebilirlik ve düşük maliyet sağlar. Bu mimari, depolama maliyetlerini geleneksel replikasyon yöntemlerine kıyasla dramatik ölçüde düşürürken, Sui ağını bir koordinasyon katmanı olarak kullanarak verinin varlığını ve bütünlüğünü kriptografik olarak kanıtlar.

Seal ise, bu depolama katmanının üzerine programlanabilir bir gizlilik katmanı inşa eder. Kimlik Tabanlı Şifreleme (Identity-Based Encryption - IBE) ve Eşik Kriptografisi (Threshold Cryptography) kullanarak, verilerin şifresini çözme yetkisini statik anahtarlardan ayırıp dinamik zincir içi (on-chain) politikalara bağlar. Bu, geliştiricilerin "bu veriyi sadece şu NFT'ye sahip olan ve son 24 saatte aktif olan kullanıcı çözebilir" gibi karmaşık erişim kontrollerini doğrudan akıllı sözleşmeler (Move) üzerinden tanımlamasına olanak tanır.

Bu dokümantasyon, kurulumdan dağıtıma, akıllı sözleşme mühendisliğinden istemci tarafı entegrasyonuna kadar, güvenli ve ölçeklenebilir dApp'ler (merkeziyetsiz uygulamalar) geliştirmek için gereken tüm teknik detayları, kod örneklerini ve mimari kararları derinlemesine analiz etmektedir.

---

2. Walrus Protokolü: Merkeziyetsiz Depolamanın Yeni Paradigması

Geleneksel blok zinciri depolama yöntemleri, verinin her düğüm (node) tarafından kopyalanmasını gerektirdiğinden, ölçeklenebilirlik açısından ciddi sınırlamalara sahiptir. Walrus, bu yaklaşımı reddederek, depolama ve mutabakat (consensus) katmanlarını birbirinden ayırır ve Sui'yi sadece metadata yönetimi için kullanır.

2.1 Red Stuff Kodlaması ve Veri Parçalanması

Walrus'un kalbinde, verinin "blob" adı verilen ikili büyük nesneler halinde işlenmesi yatar. Bir yayıncı (publisher) sisteme bir dosya yüklemek istediğinde, bu dosya **Red Stuff** adı verilen, RaptorQ kodlarına benzer gelişmiş bir hata düzeltme algoritması ile işlenir. Veri, "shard" veya "sliver" adı verilen küçük parçalara bölünür. Bu algoritmanın en kritik özelliği, verinin geri kazanılması için tüm parçalara ihtiyaç duyulmamasıdır; toplam parçaların sadece belirli bir alt kümesinin (örneğin $3f+1$ düğümden $2f+1$'inin) erişilebilir olması, orijinal verinin bit-bit yeniden oluşturulması için yeterlidir.

Bu mekanizma, ağdaki depolama düğümlerinin (storage nodes) önemli bir kısmı çevrimdışı olsa veya kötü niyetli davransa bile (Bizans hatası), verinin bütünlüğünün korunmasını sağlar. Geliştiriciler için bunun anlamı, gigabaytlarca veriyi (videolar, AI modelleri, oyun varlıkları) zincir güvenliğinde ancak bulut depolama maliyetlerine yakın bir maliyetle saklayabilmeleridir.

2.2 Sui ile Koordinasyon ve Kanıtlanabilirlik

Walrus, depolama düğümlerinin yönetimini ve ödeme süreçlerini Sui blok zinciri üzerinden koordine eder. Bir blob Walrus'a yüklendiğinde, bu bloğun metadata'sı (boyutu, kodlama parametreleri, ödeme bilgisi ve blob ID'si) bir Sui nesnesi (object) olarak zincire kaydedilir. Bu nesne, bloğun "varlık kanıtı" (Proof of Availability) işlevini görür. Akıllı sözleşmeler, verinin içeriğini okumak zorunda kalmadan, verinin Walrus ağında güvenli bir şekilde saklandığını ve erişilebilir olduğunu doğrulayabilir. Bu, "programlanabilir depolama" kavramının temelini oluşturur; depolama süresi uzatılabilir, sahiplik devredilebilir veya depolama alanı kiralanabilir.

2.3 Yayıncılar (Publishers) ve Toplayıcılar (Aggregators)

Walrus ekosistemi iki ana aktör üzerinden kullanıcı etkileşimini sağlar:

- **Yayıncılar (Publishers):** Veriyi kodlayan, parçalara ayıran ve depolama düğümlerine dağıtan istemci veya hizmetlerdir. Kullanıcılar veriyi yazmak (write) için yayıncılarla etkileşime girer.
- **Toplayıcılar (Aggregators):** Depolama düğümlerinden parçaları toplayan, veriyi yeniden inşa eden ve son kullanıcıya HTTP üzerinden sunan hizmetlerdir. Bir web tarayıcısı veya dApp, veriyi okumak (read) için bir toplayıcıya istek gönderir. Toplayıcılar genellikle CDN

(İçerik Dağıtım Ağı) gibi davranarak sık erişilen verileri önbelleğe alır ve performansı artırır.

Aşağıdaki tablo, Walrus mimarisindeki bileşenlerin rollerini ve etkileşimlerini özetlemektedir:

Bileşen	Görev ve Sorumluluk	Etkileşim Türü
Depolama Düğümü	Kodlanmış veri parçalarını (shards) fiziksel olarak saklar.	Arka Uç (Back-end)
Sui Akıllı Sözleşmesi	Metadata, ödeme, epoch yönetimi ve düğüm komitesi seçimini yönetir.	Koordinasyon
Yayıncı (Publisher)	Veriyi kodlar (Red Stuff), ödemeyi yapar ve düğümlere yazar.	Yazma (Write)
Toplayıcı (Aggregator)	Veriyi düğümlerden okur, kodu çözer ve HTTP ile sunar.	Okuma (Read)
Son Kullanıcı/İstemci	Veriyi görüntüler veya yükler (SDK aracılığıyla).	Uç Nokta (Endpoint)

---

3. Seal Protokolü: Programlanabilir Erişim Kontrolü

Merkeziyetsiz depolama tek başına verilerin herkese açık olduğu anlamına gelir. Seal, bu verilerin üzerine bir gizlilik katmanı ekler. Seal, veriyi şifrelemek için kullanılan anahtarları, zincir üzerindeki koşullara bağlayarak "erişimi kodla yönetilebilir" hale getirir.

3.1 Kimlik Tabanlı Şifreleme (IBE)

Geleneksel Açık Anahtar Altyapısı (PKI), önceden üretilmiş genel ve özel anahtar çiftlerine dayanır. Seal ise Kimlik Tabanlı Şifreleme (IBE) kullanır. IBE'de, kullanıcının veya verinin

"kimliği" (örneğin, bir akıllı sözleşme adresi ve o sözleşmedeki belirli bir ID), doğrudan şifreleme için genel anahtar olarak kullanılır. Bu, bir veri parçasının, henüz o veriyi çözme yetkisine sahip bir kullanıcı var olmasa bile (örneğin, gelecekte bir NFT'yi satın alacak kişi), o "kimlik" (NFT sahipliği) için şifrelenebilmesini sağlar. Şifreleme işlemi istemci tarafında, çevrimdışı olarak gerçekleştirilebilir ve herhangi bir merkezi sunucuyla etkileşim gerektirmez.

3.2 Eşik Kriptografisi ve Anahtar Sunucuları

Seal'de tek bir "özel anahtar" (private key) yoktur. Bunun yerine, bir "Ana Sır" (Master Secret), Shamir'in Sır Paylaşımı (Shamir's Secret Sharing) yöntemiyle birden fazla Anahtar Sunucusu (Key Server) arasında paylaştırılmıştır. Bir verinin şifresini çözmek isteyen kullanıcı, bu sunucuların bir alt kümesinden (örneğin 3 sunucudan 2'si) "şifre çözme payı" (decryption share) talep etmelidir. Kullanıcı yeterli sayıda payı topladığında, istemci tarafında nihai şifre çözme anahtarını oluşturabilir. Bu mimari, tek bir sunucunun hacklenmesi veya çevrimdışı olması durumunda bile sistemin güvenliğini ve erişilebilirliğini korur.

3.3 Zincir İçi Politika Doğrulama

Seal'in en yenilikçi yönü, anahtar sunucularının bir şifre çözme payı vermeden önce neyi kontrol ettiğidir. Geleneksel sistemlerde bu kontrol bir veritabanı veya API anahtarı ile yapılırken, Seal'de bu kontrol **Move akıllı sözleşmeleri** üzerinde gerçekleşir. Bir kullanıcı şifre çözme talebinde bulunduğunda, anahtar sunucuları Sui blok zincirindeki belirtilen akıllı sözleşmenin seal_approve fonksiyonunu simüle eder. Eğer bu fonksiyon başarılı bir şekilde çalışırsa (yani hata vermezse), sunucu kullanıcının yetkili olduğunu kabul eder ve payı imzalar. Bu mekanizma, erişim kontrolünün tamamen şeffaf, denetlenebilir ve programlanabilir olmasını sağlar.

---

4. Geliştirme Ortamının Hazırlanması ve Kurulum

Hackathon katılımcılarının projelerini başarıyla geliştirebilmeleri için stabil bir geliştirme ortamına ihtiyaçları vardır. Bu bölüm, Rust toolchain'inden Walrus CLI ve Seal SDK kurulumuna kadar gerekli adımları detaylandırır.

4.1 Temel Gereksinimler ve Bağımlılıklar

Geliştirme süreci, Sui ve Walrus'un temelini oluşturan Rust programlama dili ve ilgili araç setlerini gerektirir.

1. Rust ve Cargo Kurulumu:
   Sui ve Walrus binary'lerini derlemek veya çalıştırmak için Rust gereklidir.
   Bash
   curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

```
source "$HOME/.cargo/env"
```

Analiz: Rust'ın seçilmesi, bellek güvenliği ve performans avantajları sağlar; bu da yüksek verimli veri işleme (Walrus) ve güvenli kriptografik işlemler (Seal) için kritiktir.

2. Sui CLI Kurulumu (Testnet):
   Hackathon projeleri genellikle Testnet üzerinde geliştirilir.
   ```
   Bash
   cargo install --locked --git https://github.com/MystenLabs/sui.git --branch testnet sui
   ```

   Kurulum sonrası sui --version komutu ile doğrulama yapılmalıdır.
3. Node.js ve Paket Yöneticisi:
   İstemci tarafı (frontend) ve SDK entegrasyonu için Node.js (LTS sürümü, v18+) ve pnpm (performans ve disk alanı verimliliği için önerilir) kurulmalıdır.

4.2 Ağ Konfigürasyonu (Sui Testnet)

Sui Testnet, geliştiricilerin gerçek ekonomik değer riski olmadan projelerini test edebilecekleri bir ortamdır.

1. **Ortam Ayarlama:**
   ```
   Bash
   sui client new-env --alias testnet --rpc https://fullnode.testnet.sui.io:443
   sui client switch --env testnet
   ```

2. Cüzdan Oluşturma ve Fonlama:
   İşlem ücretleri (gas) için SUI token'larına ihtiyaç vardır.
   ```
   Bash
   sui client new-address ed25519
   ```

   Discord faucet veya cüzdan eklentileri kullanılarak bu adrese Testnet SUI talep edilmelidir.

4.3 Walrus CLI Kurulumu ve Yapılandırma

Walrus CLI, depolama düğümleriyle doğrudan etkileşim kurmak ve sistem durumunu sorgulamak için kullanılır.

1. Binary Kurulumu:
   Resmi kurulum betiği kullanılarak en güncel Testnet sürümü indirilir.
   ```
   Bash
   curl -sSf https://docs.wal.app/setup/walrus-install.sh | sh -s -- -n testnet
   export PATH="$HOME/.local/bin:$PATH"
   ```

2. İstemci Yapılandırması (client_config.yaml):

Walrus istemcisinin hangi sistem nesnelerini (system object) ve staking havuzlarını kullanacağını bilmesi gerekir. Bu yapılandırma dosyası, Testnet için önceden tanımlanmış parametreleri içerir.

Bash

```
mkdir -p ~/.config/walrus/
curl https://docs.wal.app/setup/client_config.yaml -o ~/.config/walrus/client_config.yaml
```

Önemli Detay: Bu dosya, system_object ve staking_object gibi kritik on-chain adresleri içerir. Yanlış yapılandırma, istemcinin ağ ile senkronize olamamasına neden olur.

3. WAL Token Edinimi:
   Depolama alanı satın almak için WAL token gereklidir. Testnet üzerinde SUI token'ları WAL ile takas edilebilir.

   Bash

   ```
   walrus get-wal --amount 500000000
   ```

   Bu komut, belirtilen miktarda (MIST cinsinden) SUI harcayarak karşılığında WAL (FROST) alır. Depolama satın alma işlemi epoch bazlıdır ve WAL ile ödenir.

4.4 SDK Kurulumu

Uygulama katmanı entegrasyonu için gerekli TypeScript paketleri projeye eklenmelidir.

Bash

```
npm install @mysten/sui @mysten/walrus @mysten/seal @mysten/bcs @mysten/dapp-kit @tanstack/react-query
```

- @mysten/walrus: Blob yükleme, indirme ve encoding işlemleri için.
- @mysten/seal: Şifreleme, şifre çözme ve anahtar sunucusu iletişimi için.
- @mysten/bcs: (Binary Canonical Serialization) Sui üzerindeki veri yapılarını serileştirmek için gereklidir, özellikle Move fonksiyonlarına argüman hazırlarken kullanılır.

---

5. Walrus Entegrasyonu: Veri Depolama ve Erişim

Bu bölümde, Walrus protokolünün bir uygulama içerisine nasıl entegre edileceği, hem CLI hem de SDK kullanılarak detaylandırılacaktır.

5.1 Blob ve Quilt Kavramları

Walrus'ta veri yönetimi iki ana yapı üzerinden yürütülür:

- **Blob:** Tekil bir dosya veya veri bütünüdür. Her blob'un benzersiz bir ID'si vardır.
- **Quilt:** Birden fazla blob'un bir araya getirilerek tek bir mantıksal birim olarak saklanmasını sağlar. Bu, örneğin bir web sitesinin HTML, CSS ve JS dosyalarının bütünsel olarak yönetilmesi ve versiyonlanması için kullanılır. SDK, dosyaları otomatik olarak Quilt formatında paketleyebilir.

5.2 HTTP API ve Daemon Modu

Geliştirme aşamasında veya SDK'nın kullanılamadığı durumlarda (örneğin curl ile test yaparken), Walrus istemcisi bir "daemon" modunda çalıştırılarak yerel bir HTTP sunucusu gibi davranabilir.

1. **Daemon'ı Başlatma:**
   Bash
   `walrus daemon --bind-address 127.0.0.1:31415`

   Bu komut, yerel makinede 31415 portunu dinleyen bir sunucu başlatır. Bu sunucu hem Yayıncı (Publisher) hem de Toplayıcı (Aggregator) işlevi görür.
2. **Veri Yükleme (Publisher):**
   Bash
   `curl -X PUT "http://127.0.0.1:31415/v1/blobs?epochs=5" -d "Hackathon İçin Önemli Veri"`

   - epochs=5: Verinin 5 epoch boyunca (Testnet'te 1 epoch yaklaşık 24 saattir ancak değişebilir) saklanacağını belirtir. Süre dolduğunda veri silinebilir.
   - *Çıktı:* İşlem başarılı olduğunda, blob ID'sini ve Sui üzerindeki olay (event) digest'ini içeren bir JSON döner.
3. **Veri Okuma (Aggregator):**
   Bash
   `curl "http://127.0.0.1:31415/v1/blobs/<BLOB_ID>"`

   Bu istek, parçaları depolama düğümlerinden toplar, birleştirir ve orijinal veriyi döndürür.

5.3 TypeScript SDK ile Programatik Yükleme

Modern dApp'ler için SDK kullanımı, tip güvenliği ve entegrasyon kolaylığı açısından zorunludur.

İstemci Başlatma:

TypeScript

```
import { WalrusClient } from '@mysten/walrus';
import { getFullnodeUrl, SuiClient } from '@mysten/sui/client';

// Sui ağına bağlantı
const suiClient = new SuiClient({ url: getFullnodeUrl('testnet') });

// Walrus istemcisi yapılandırması
const walrusClient = new WalrusClient({
    network: 'testnet',
    suiClient,
});
```

Dosya Yükleme (Doğrudan Yazma):
Bu yöntem, istemcinin (tarayıcı veya sunucu) depolama ödemesini yapabilecek bir özel anahtara sahip olmasını gerektirir.

TypeScript

```
import { Ed25519Keypair } from '@mysten/sui/keypairs/ed25519';

// Yayıncı hesabı (WAL bakiyesi olmalı)
const keypair = Ed25519Keypair.fromSecretKey(SECRET_KEY);
const file = new File(, "hello.txt", { type: "text/plain" });

const { blobId, info } = await walrusClient.writeBlob({
    data: file,
    epochs: 1,
    deletable: true, // Verinin daha sonra silinebilmesine izin ver
    signer: keypair,
});

console.log(`Blob ID: ${blobId}`);
```

Bu işlem sırasında SDK, dosyayı yerel olarak parçalara ayırır (WASM kullanarak), Sui üzerinde bir depolama kaynağı kaydeder ve ardından parçaları depolama düğümlerine gönderir. Tüm bu karmaşık süreç writeBlob fonksiyonu tarafından soyutlanmıştır.

Upload Relay Kullanımı:
Kullanıcıların kendi özel anahtarlarıyla işlem yapmasını istemediğiniz durumlarda (örneğin gas ücretini uygulamanın karşılaması gerekiyorsa), bir Upload Relay kullanılır. Relay, veriyi alır ve depolama ücretini ödeyerek Walrus'a yazar.

TypeScript

```
const uploadInfo = await walrusClient.writeBlobToUploadRelay({
  data: file,
  relayUrl: 'https://publisher.walrus-testnet.walrus.space', // Örnek genel relay
  epochs: 1,
});
```

Hackathon projelerinde kullanıcı deneyimini iyileştirmek için Relay kullanımı önerilir, çünkü kullanıcıdan WAL token talep etme zorunluluğunu ortadan kaldırır.

---

6. Seal Entegrasyonu: Şifreleme ve Politika Yönetimi

Seal entegrasyonu, verinin güvenliğini sağlarken erişim kontrolünü akıllı sözleşmelere devreder.

6.1 Testnet Konfigürasyonu

Seal istemcisi, hangi Anahtar Sunucuları ile iletişim kuracağını bilmelidir. Testnet için sabit adresler kullanılır.

TypeScript

```
import { SealClient } from '@mysten/seal';

const TESTNET_KEY_SERVERS = [
  {
    objectId: "0x73d05d62c18d9374e3ea529e8e0ed6161da1a141a94d3f76ae3fe4e99356db75",
    weight: 1
  },
```

```
    {
        objectId: "0xf5d14a81a982144ae441cd7d64b09027f116a468bd36e7eca494f750591623c8",
        weight: 1
    }
];

const sealClient = new SealClient({
    suiClient, // Mevcut Sui istemcisi
    serverConfigs: TESTNET_KEY_SERVERS,
    verifyKeyServers: true, // Sunucu imzalarını doğrula
});
```

.

6.2 Şifreleme Akışı (Client-Side Encryption)

Şifreleme işlemi tamamen istemci tarafında gerçekleşir. Veri ağa gönderilmeden önce şifrelenir.

1. **Politika ve Kapsam Belirleme:** Şifrelemenin hangi Move paketi (packageId) ve hangi nesne/ID (scope) tarafından yönetileceği belirlenir.
2. **Şifreleme İşlemi:**

TypeScript

```
const DATA_TO_ENCRYPT = new TextEncoder().encode("Gizli Proje Verisi");
const POLICY_PACKAGE_ID = "0x..."; // Deploy ettiğiniz Move sözleşmesi
const SCOPE_ID = "0x..."; // Erişim kontrolü için referans alınan ID (örn. NFT ID)

const { encryptedObject } = await sealClient.encrypt({
    data: DATA_TO_ENCRYPT,
    packageId: POLICY_PACKAGE_ID,
    id: SCOPE_ID,
    threshold: 2, // Şifre çözmek için en az 2 sunucudan onay gerekir
    demType: 1, // Veri Kapsülleme Mekanizması (AES-GCM)
    kemType: 0, // Anahtar Kapsülleme Mekanizması
});
// encryptedObject, Walrus'a yüklenmeye hazır şifreli bayt dizisidir.
```

Tasarım Kararı: threshold değeri güvenlik ve erişilebilirlik dengesini belirler. 2-of-N yapısı, bir sunucunun çökmesine karşı tolerans sağlarken güvenliği artırır.

6.3 Şifre Çözme ve Yetki Doğrulama

Kullanıcı veriyi indirdiğinde (Walrus'tan), şifresini çözmek için Seal Anahtar Sunucularına başvurmalıdır.

TypeScript

```
import { Transaction } from '@mysten/sui/transactions';
import { EncryptedObject } from '@mysten/seal';

// 1. Walrus'tan inen veriyi parse et
const encryptionMetadata = EncryptedObject.parse(downloadedBytes);

// 2. Erişim İsteği İşlemini (Transaction) Hazırla
// Bu işlem zincirde çalıştırılmaz, sadece sunuculara simülasyon için gönderilir.
const tx = new Transaction();
tx.moveCall({
    target: `${POLICY_PACKAGE_ID}::policy_module::seal_approve`,
    arguments:,
});

// 3. İşlem Baytlarını Oluştur
const txBytes = await tx.build({ client: suiClient, onlyTransactionKind: true });

// 4. Şifre Çözme İsteği
try {
    const decryptedData = await sealClient.decrypt({
        data: downloadedBytes,
        sessionKey: userSessionKey, // Kullanıcı cüzdanı tarafından imzalanmış oturum anahtarı
        txBytes: txBytes,
        checkShareConsistency: true,
    });
    console.log("Veri:", new TextDecoder().decode(decryptedData));
} catch (e) {
    console.error("Erişim reddedildi veya sunucu hatası:", e);
}
```

Mekanizma: Anahtar sunucuları txBytes'ı alır ve kendi yerel Sui düğümlerinde seal_approve fonksiyonunu çalıştırır (dry-run). Eğer fonksiyon başarılı olursa (abort etmezse), sunucu kullanıcının yetkili olduğuna kanaat getirir ve bir şifre çözme payı (key share) imzalar. İstemci bu payları toplar ve orijinal anahtarı oluşturur.

---

7. Akıllı Sözleşme Mühendisliği (Move)

Seal'in çalışabilmesi için zincir üzerinde bir "karar verici" mekanizmaya ihtiyaç vardır. Bu, Move dilinde yazılmış bir akıllı sözleşmedir.

7.1 seal_approve Arayüzü

Sözleşme, kesinlikle seal_approve adında, public entry görünürlüğüne sahip ve ilk argümanı vector<u8> olan bir fonksiyon içermelidir.

Örnek Senaryo: NFT Sahipliğine Dayalı Erişim
Aşağıdaki sözleşme, sadece belirli bir "MemberCard" NFT'sine sahip olan ve bu kartın seviyesi (tier) 1 veya üzeri olan kullanıcıların şifreyi çözmesine izin verir.

Kod snippet'i

```
module hackathon::access_control {
    use sui::object::{Self, UID};
    use sui::tx_context::{Self, TxContext};

    /// Hata kodları
    const E_NOT_AUTHORIZED: u64 = 1;

    /// Erişim için gerekli varlık
    public struct MemberCard has key, store {
        id: UID,
        tier: u8,
    }

    /// Seal tarafından çağrılan zorunlu fonksiyon
    /// @param scope_id: Şifreleme sırasında kullanılan ID (serileştirilmiş)
    /// @param proof: Kullanıcının sunduğu kanıt nesnesi (NFT)
```

```
    public entry fun seal_approve(
        scope_id: vector<u8>,
        proof: &MemberCard,
        ctx: &TxContext
    ) {
        // Kontrol 1: Move VM, 'proof' nesnesinin sahipliğini otomatik kontrol eder.
        // Fonksiyonu çağıran kişi nesnenin sahibi değilse işlem başarısız olur.

        // Kontrol 2: Üyelik seviyesi kontrolü
        assert!(proof.tier >= 1, E_NOT_AUTHORIZED);

        // Kontrol 3: Kapsam kontrolü (Opsiyonel)
        // Şifrelenen verinin ID'si ile NFT'nin ID'si eşleşiyor mu?
        let card_id_bytes = object::id_to_bytes(&object::id(proof));
        assert!(scope_id == card_id_bytes, E_NOT_AUTHORIZED);
    }
}
```

Dağıtım (Deployment):

1. Kodu derleyin: sui move build
2. Yayınlayın: sui client publish --gas-budget 100000000
3. Çıktıdan PackageID'yi not edin; bu ID, frontend entegrasyonunda POLICY_PACKAGE_ID olarak kullanılacaktır.

---

8. Uçtan Uca Uygulama Mimarisi ve React Entegrasyonu

Bir hackathon projesinde parçaları birleştirmek (Frontend + Storage + Privacy) en kritik adımdır.

8.1 Frontend Kurulumu (Sui dApp Kit)

React tabanlı bir arayüz için dApp Kit standarttır. Cüzdan bağlantısını ve RPC çağrılarını yönetir.

```
Bash
```

```
npm install @mysten/dapp-kit @tanstack/react-query
```

Provider Konfigürasyonu (App.tsx):

```
TypeScript
```

```
import { SuiClientProvider, WalletProvider } from '@mysten/dapp-kit';
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';
import { getFullnodeUrl } from '@mysten/sui/client';

const queryClient = new QueryClient();
const networks = { testnet: { url: getFullnodeUrl('testnet') } };

export default function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <SuiClientProvider networks={networks} defaultNetwork="testnet">
        <WalletProvider>
          <AnaBilesen />
        </WalletProvider>
      </SuiClientProvider>
    </QueryClientProvider>
  );
}
```

.

8.2 Mobil Geliştirme (React Native) ve Polyfill Sorunsalı

Seal ve Walrus SDK'ları, kriptografik işlemler için Node.js'in crypto modülüne veya tarayıcıların Web Crypto API'sine ihtiyaç duyar. React Native ortamında bu modüller varsayılan olarak bulunmaz. Bu nedenle, mobil uygulama geliştirirken **Polyfill** (yamama) işlemi zorunludur.

Gerekli Paketler:

1. react-native-quick-crypto: Node crypto modülünün yüksek performanslı C++ uygulaması.
2. text-encoding: TextEncoder/TextDecoder desteği için.

3. react-native-url-polyfill: URL işlemleri için.
4. readable-stream: Veri akışları için.

Uygulama (shim.js):
Projenin kök dizininde bir shim.js dosyası oluşturun ve index.js dosyasının en üst satırında import edin.

JavaScript

```
// shim.js
import 'react-native-url-polyfill/auto';
import { install } from 'react-native-quick-crypto';

// Global crypto nesnesini yamala
install();

if (typeof BigInt === 'undefined') global.BigInt = require('big-integer');
if (typeof TextEncoder === 'undefined') global.TextEncoder = require('text-encoding').TextEncoder;
if (typeof TextDecoder === 'undefined') global.TextDecoder = require('text-encoding').TextDecoder;
```

Bu adım atlanırsa, Seal SDK şifreleme sırasında "crypto not found" hatası verecektir.[19]

8.3 CORS ve Ağ Geçidi Sorunları

Web tarayıcılarından Walrus Aggregator'larına doğrudan veri çekerken (fetch), Cross-Origin Resource Sharing (CORS) hataları ile karşılaşılabilir.

- **Çözüm 1 (Geliştirme):** Yerel daemon'ı CORS başlıklarına izin verecek şekilde yapılandırın veya tarayıcı eklentileri kullanın.
- **Çözüm 2 (Production):** Kendi Aggregator'ınızı çalıştırıyorsanız, önünde Nginx veya Caddy gibi bir ters vekil sunucu (reverse proxy) kullanarak Access-Control-Allow-Origin: * başlığını ekleyin.
- **Testnet Durumu:** Mysten Labs tarafından işletilen genel toplayıcılar genellikle CORS'a izin verir, ancak yoğunluk durumunda 429 (Too Many Requests) hatası dönebilir.

---

9. Hackathon Stratejileri ve Proje Şablonları

Walrus ve Seal'in yeteneklerini en iyi sergileyen proje türleri, verinin sahipliğinin ve gizliliğinin merkeziyetsiz olarak yönetildiği senaryolardır.

9.1 Proje Fikri: "Merkeziyetsiz Patreon" (İçerik Aboneliği)

- **Konsept:** İçerik üreticileri yüksek çözünürlüklü videoları veya özel dosyaları Walrus'a yükler.
- **Seal Mantığı:** Dosyalar şifrelenir. seal_approve fonksiyonu, kullanıcının cüzdanında geçerli bir "Abonelik NFT'si" olup olmadığını ve bu aboneliğin süresinin dolup dolmadığını (zincir içi zaman damgası ile) kontrol eder.
- **Walrus Kullanımı:** Videolar blob olarak saklanır. Önizleme görselleri (thumbnail) şifresiz olarak saklanabilir.
- **Kazanma Faktörü:** Gerçek veri sahipliği gösterir; platform kapansa bile içerik ve erişim hakları (Sui üzerinde) yaşamaya devam eder.

9.2 Proje Fikri: "Güvenilir Yapay Zeka Veri Pazarı"

- **Konsept:** Araştırmacılar, AI eğitim setlerini satar.
- **Seal Mantığı:** Ödeme yapıldığında bir "Makbuz" nesnesi oluşturulur. Erişim politikası bu makbuzu kontrol eder.
- **Walrus Kullanımı:** Devasa veri setleri (GB'larca) Quilt yapısı kullanılarak verimli bir şekilde saklanır.
- **Kazanma Faktörü:** AI ve Blok zinciri kesişimindeki en büyük sorunlardan biri olan "Veri Erişilebilirliği" ve "Monetizasyon" sorununu çözer.

9.3 Proje Fikri: "Dijital Miras / Ölü Adamın Anahtarı"

- **Konsept:** Şifreli sırlar (cüzdan anahtarları, vasiyet), sahibi belirli bir süre "check-in" yapmazsa mirasçıya açılır.
- **Seal Mantığı:** Politika, kullanıcı nesnesindeki son_aktif_olma zaman damgasını kontrol eder. Eğer şu_anki_zaman > son_aktif_olma + 30_gün ise, önceden tanımlanmış mirasçı adresi için şifre çözme izni verilir.
- **Kazanma Faktörü:** Tamamen otonom, aracısız ve programlanabilir bir kriptografi örneğidir.

---

10. Sorun Giderme ve En İyi Uygulamalar

10.1 Sık Karşılaşılan Hatalar

- **Epoch Süresi:** Walrus depolama alanı "epoch" (dönem) bazlı satın alınır. Testnet'te 1 epoch süresi kısadır. Verinizin hackathon süresince silinmemesi için en az 30 epoch'luk depolama satın alın.

- **Manuel Parça Birleştirme:** Parçaları (slivers) manuel olarak birleştirmeye çalışmayın. SDK'nın readBlob fonksiyonu, eksik parçaları tolere ederek yeniden inşayı otomatik yapar.
- **Seal Erişim Reddi (NoAccessError):** Şifre çözme başarısız olursa şunları kontrol edin:
  1. Move fonksiyonu (seal_approve) public entry mi?
  2. Argüman tipleri (özellikle vector<u8>) doğru mu?
  3. Kullanıcının cüzdanı, kontratın beklediği nesneye (NFT vb.) gerçekten sahip mi?.

10.2 Tanılama Araçları

- **WalrusScan:** Blob ID'nizin ağda yayılıp yayılmadığını ve sertifikasyon durumunu kontrol etmek için(https://walruscan.com) kullanın.
- **Sui Explorer:** Seal entegrasyonunda hata alıyorsanız, işlemin simülasyonunu Sui Explorer üzerinde inceleyin. debug::print kullanarak Move sözleşmesindeki mantık hatalarını izleyebilirsiniz.

---

11. Sonuç

Walrus ve Seal, Web3 uygulamaları için eksik olan iki kritik parçayı—ucuz depolama ve programlanabilir gizlilik—tamamlamaktadır. Bu teknolojilerin kombinasyonu, geliştiricilere sadece finansal varlıkları değil, her türlü dijital içeriği merkeziyetsiz, sansüre dirençli ve özel bir şekilde yönetme gücü verir. Bu raporda sunulan mimari ve kod şablonları, hackathon katılımcılarının teknik engelleri hızla aşarak inovatif senaryolara odaklanmalarını sağlamak amacıyla tasarlanmıştır.

Önerilen Sonraki Adım: "Walrus Upload Relay Example" ve "Seal Example" depolarını klonlayın ve temel bir "Şifrele -> Yükle -> İndir -> Çöz" döngüsünü kendi yerel ortamınızda başarıyla gerçekleştirmeden karmaşık özellik geliştirmeye başlamayın. Başarı, altyapının sağlam temeller üzerine kurulmasına bağlıdır.

Walrus için dokümantasyonlar

https://sdk.mystenlabs.com/walrus
https://github.com/MystenLabs/awesome-walrus?tab=readme-ov-file#mainnet-publisher
https://blobboard.wal.app

https://docs.wal.app/docs/usage/sdks

https://github.com/MystenLabs/ts-sdks/tree/main/packages/walrus/examples

https://github.com/MystenLabs/ts-sdks/blob/main/packages/walrus/examples/publisher/index.ts
https://github.com/MystenLabs/ts-sdks/tree/main/packages/walrus/examples/basics

https://docs.wal.app/docs/usage/networks#testnet-wal-faucet

Seal Örnek

https://github.com/AliErcanOzgokce/sample_seal