Project Zoraqle Whitepaper

A proof of concept platform for publishing, managing, and monetizing authentic data streams originating in the physical world

Phil Strong[1]
CEO, Zymbit

Shivaansh Kapoor[2]
Software Engineer, Zymbit

Scott Miller[3]
CTO, Zymbit

25 August, 2022
V1.0

[1]phil@zymbit.com
[2]shiv@zymbit.com
[3]scott@zymbit.com

Contents

1 Abstract

The Internet of Things (IoT) is generating vast amounts of data that is often unique and has value to multiple parties over time. Such data is a form of non-fungible digital asset. To effectively transact and monetize such data assets requires that their origin, integrity, and ownership be established in a provable way. Additionally, a process is required to connect subscribers, establish terms of sale, and manage access rights to the data. These requirements predicate the foundation of project Zoraqle.

2 Motivating Factors

Zoraqle aims to produce "authentic"[1] data streams with immutable ownership rights. Owners should have the ability to monetize data streams to prospective subscribers at prices that they set. Here are Zoraqle's objectives:

1. Verifiable data "authenticity"
2. Immutable ownership rights and managed granular access rights
3. Facilitate transactions of data and monetary value between owners and subscribers

3 High Level Architecture

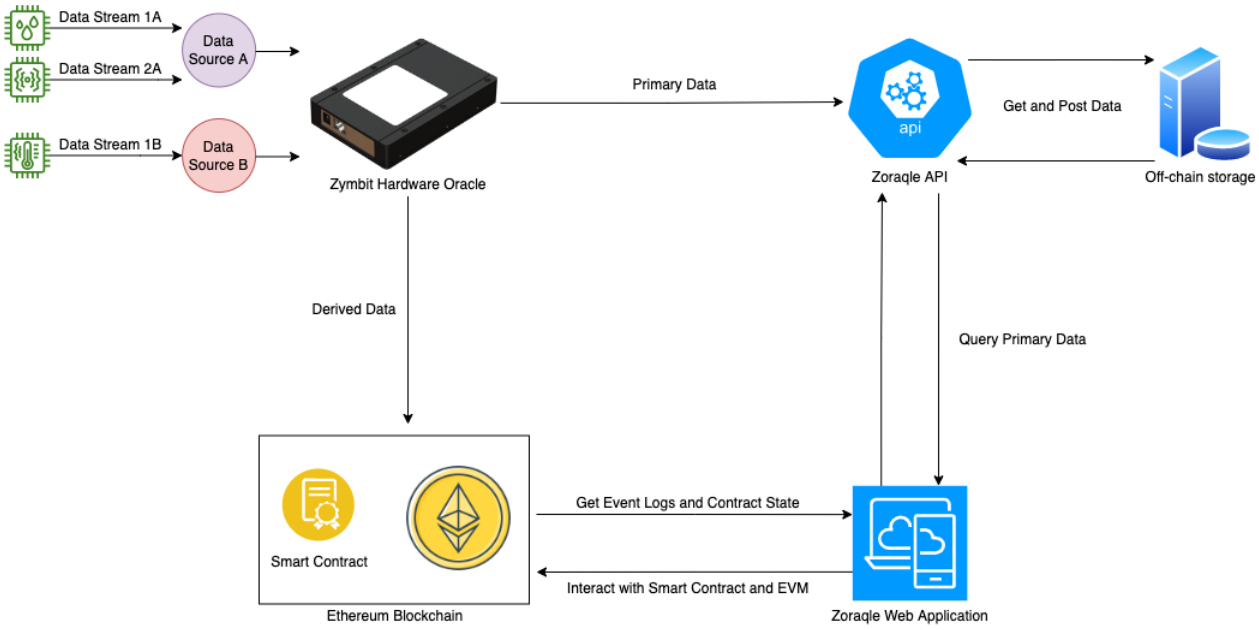
Zoraqle comprises of four key elements which work in synchrony to achieve the objectives described in the previous section.

1. **Hardware Oracle** - Embodied by Zymbit's Secure Compute Node[2], the oracle is the edge device used to collect, package, and post data packets
2. **Smart Contracts** - Formalize contractual agreements between owners and subscribers
3. **Zoraqle API** - Backend service used for interfacing with an off-chain data store
4. **Zoraqle Web Application** - Dashboard style user interface responsible for connecting owners with prospective subscribers and enabling users to seamlessly interact with the smart contracts and Zoraqle API

[1] Verifiable source and integrity

[2] https://www.zymbit.com/secure-compute-node/

3.1 Zoraqle Flow Diagram



3.2 Scalability

In an ideal world, all data packets that the hardware oracle collects would live in an encrypted tamper-proof setting. However, due to the current limitations posed by data sensitivity and on-chain storage costs, we must utilize on-chain and off-chain storage in parallel to scale the system. Off-chain storage is magnitudes cheaper than on-chain storage, incentivizing us to store the bulk of the data off-chain while only storing what is necessary to prove data "authenticity" on-chain. In the following sections, we discuss this in further detail.

3.3 Data Packets

In the context of Zoraqle, we define two distinct types of data packets:

1. **Primary Data Packet**
    - UNIX timestamp: λ (32 bytes)
    - Reference/Identifier: ζ (32 bytes)
        - $\zeta = BYTESRANDOM_{32}$

- ζ remains constant for every data stream's packet in the given sampling period[3]

- A set of raw sensor data from data stream n: ψ_n (arbitrary size)

2. **Derived Data Packet**

   - UNIX timestamp: λ (32 bytes)
   - Reference/Identifier: ζ (32 bytes)
   - $||\psi_n||$ (2 bytes)
   - Digital signature: γ (65 bytes)
     - $\gamma = ECDSASIGN^4{}_{secp256k1}(\Omega^5, {p_r}^6)$

Each data packet is associated with a specific **source**[7] and **stream**[7] from the oracle. Primary data packets are meant to be stored in an off-chain data store and derived data packets are posted on-chain through a smart contract. Every primary packet which lives off-chain has a corresponding derived packet on-chain. The cryptographic relationship between primary and derived packets is what enables Zoraqle to verify data "authenticity".

3.4 Verifying Data "Authenticity"

Zoraqle offers two schemes to prove the "authenticity" of primary data packets. On the surface, they differ in the degree of verification granularity and on-chain transaction costs.

3.4.1 Scheme 1

$$\Omega = \text{SHA256}^8(\text{BYTES}^9(\psi_n))$$

At the end of each sampling period, the oracle takes the primary data packet produced by each stream, and produces a γ with p_r set as the corresponding stream's private key[10]. Then it posts the primary and derived packets for each data stream. Scheme 1 is more capable of proving "authenticity" of granular data packets but has higher on-chain costs compared to scheme 2.

[3] An interval of time (set by the owner) at which data packets are posted by the oracle
[4] https://ethereum.github.io/yellowpaper/paper.pdf - Appendix F
[5] See section Verifying Data "Authenticity"
[6] The private key used to sign Ω
[7] See section Zoraqle Flow Diagram
[8] https://en.wikipedia.org/wiki/SHA-2
[9] A function which outputs the UTF-8 endcoded byte array of its input
[10] See section HD Wallet Structure

3.4.2 Scheme 2

$$\theta = \text{SORT}^{11}(\{\text{SHA256}(\text{BYTES}(\psi_1)),...,\text{SHA256}(\text{BYTES}(\psi_n))\})$$
$$\Upsilon = \theta_1 \parallel \theta_2 \parallel ... \parallel \theta_n$$
$$\Omega = \text{SHA256}(\text{BYTES}(\Upsilon))$$

At the end of each sampling period, the oracle takes the primary data packets produced by the data streams, hashes them, sorts their hashes (from lowest to highest), and produces a single concatenated string from the sorted hashes. Then, it hashes the resulting string and generates a γ with p_r set as the device key[10]. Finally, the oracle posts the resulting primary packets and singular derived packet. Evidently, this scheme results in a single on-chain transaction per sampling interval, but it has limited verification capability.

3.4.3 Verification Process

On the client side (Zoraqle Web Application or personal integration), owners and subscribers have the ability to verify the integrity of primary data packets. Here are the steps for scheme 1:

1. Retrieve a primary data packet from the Zoraqle API and stringify the raw JSON data
2. Convert the stringified packet into a UTF-8 encoded byte array
3. Generate a SHA256 hash from the byte array
4. Call Ξ_{ECREC}[12] on the hash and the γ retrieved from the derived packet
5. If the resulting address matches the corresponding stream's address, the primary data packet is "authentic"

The verification process for scheme 2 is more nuanced than that of scheme 1. One contingency, from a subscriber's perspective, is that they must have purchased every primary packet with the same ζ in order "authenticate" packets generated by scheme 2. If that's the case, this is how they would verify their integrity:

1. Retrieve all primary data packets with the same ζ from the Zoraqle API and stringify the raw JSON data
2. Convert all the stringified packets into UTF-8 encoded byte arrays

[11] Function which sorts its input from lowest to higest value and returns the ordered set
[12] https://ethereum.github.io/yellowpaper/paper.pdf - Appendix E

3. Generate SHA256 hashes from the byte arrays
4. Concatenate the hashes from lowest to highest hash value and convert the resulting string into a byte array
5. Generate a SHA256 hash of the byte array
6. Call Ξ_{ECREC} on the resulting hash and the γ retrieved from the derived packet with identifier ζ
7. If the resulting address matches the corresponding device's address, all the primary packets identified by ζ are "authentic"

Both these processes are intricate and susceptible to human error. For these reasons, the Zoraqle Web Application provides an easy to use service which verifies a packet's integrity and allows an owner or subscriber to download an "authenticity" report - detailing each step of the process.

3.5 Cost Analysis

In this section, we model the on-chain costs of each verification scheme. Here are the variables involved:

n : the number of data streams connected to the oracle
t : the total time of operation for the oracle (in seconds)
s : the sampling period for the oracle (in seconds)
$g()$: the gas cost of posting a derived packet on-chain

Verification Scheme 1:
$C_1(n, t, s) = (\frac{t}{s}) * n * g()$

Verification Scheme 2:
$C_2(n, t, s) = (\frac{t}{s}) * g()$

Clearly, scheme 1 is linearly more expensive than scheme 2 by a factor of n. Consequently, as n increases, it becomes increasingly important for the owner to weigh the trade-off between verification granularity and cost.

For instance, consider an oracle with 2 connected data streams and a sampling period of 15 minutes (4 packets posted per hour). Let's assume, for the purpose of simplicity, $g()$ eternally evaluates to \$0.50. The cost of running the oracle for a day with scheme 1 is \$96 and \$48 for scheme 2. If we

connect 3 additional streams to the oracle (total of 5), the cost rises to $240 for scheme 1 while remaining at $48 for scheme 2.

4 Monetizing Data Streams

4.1 Marketplace

The Zoraqle Web Application connects owners and subscribers. It allows prospective subscribers to view all the oracles in the Zoraqle ecosystem and request access to any of their permissioned marketplaces. If granted access by the owner, they are now authorized subscribers in the context of that oracle. Authorized subscribers can buy access to the oracle's available data streams for a specified period of time. In the initial project scope, this period of time is limited to historical data in order to eliminate the counterparty risk of the oracle halting operation in the future.

There are many approaches to customer discovery and connecting owners with prospective subscribers - our approach is subject to change over time.

4.2 Establishing a Price

We considered three common models for establishing the price of access to a data stream. The merits of each are summarized below, and Zoraqle currently uses method three, **Limit Sell**:

1. **Market Price** - Authorized subscribers can bid in an open marketplace for access to a data stream for a specified period (set by the owner) and the highest bidder is granted access

2. **Limit Buy** - Authorized subscribers can request to buy access to a data stream for a specified period and price (set by the subscriber) and they reach an agreement if the owner approves of their terms

3. **Limit Sell** - The owner publicly sets a price for access to a data stream for a specified period (i.e. $/hour) and authorized subscribers can buy access for however long (contingent on the amount paid)

Each one of these models has its own benefits and drawbacks. Option 1 allows the owner to discover the highest rate that a prospective subscriber is willing to pay, but it relies on the owner to be present in starting and ending the auction and only allows for one subscriber at a time to have access. On the other hand, option 2 works conversely to traditional markets,

as the market taker sets the price, and the market maker chooses to accept their terms, or not. Option 3 allows for the owner to set a price and an arbitrary number of authorized subscribers to buy access simultaneously in an autonomous manner. All three of these models have their unique use cases, but as stated previously, Zoraqle applies option 3.

It is up to the owner to decide upon a reasonable price which prospective subscribers would be willing to pay. Some factors they may consider are the market's demand for the data streams their oracle produces and the verification scheme that their oracle employs.

5 Smart Contracts

5.1 Why Smart Contracts?

Smart contracts are contractual agreements which are autonomously executed if certain pre-set conditions are met. They provide a powerful tool to formalize agreements with crucial invariants: immutability, transparency, and determinism.

For Zoraqle, we utilize the Ethereum[13] Blockchain as a medium to codify and enforce agreements between owners and subscribers, and post derived packets on-chain to ensure that the signatures generated by the oracle are eternal.

5.2 Important Considerations

5.2.1 Cost

Since smart contracts are executed in a decentralized environment, nodes need a cryptoeconomic incentive to honestly validate transactions. Ethereum manages this through transaction fees known as gas fees and a block reward. Every operation that the Ethereum Virtual Machine (EVM) can execute has a pre-determined gas cost[14]. A simple arithmetic operation like addition costs 3 gas, while a more computation and storage intensive operation like creating a contract costs 32000 gas.

[13] https://ethereum.org/en/
[14] https://ethereum.github.io/yellowpaper/paper.pdf - Appendix G

With this in mind, we must be careful and resourceful when designing our contract. The code should be efficient and minimal yet fully functional.

5.2.2 Security

Due to the *quasi*-Turing-complete[15] nature of the EVM, developers can write complex contracts which manage a great deal of data and monetary value. However, with great power comes great responsibility.

Smart contracts have been known to have potential security vulnerabilities which attack vectors can identify and exploit. Some examples of major exploitations are the DAO hack[16] and Parity multisignature wallet hacks[17]. The DAO hack resulted in a loss of 3.6 million ether ($50 million dollars at the time), and eventually lead to a controversial hard fork to the Ethereum network in an effort to restore the funds.

For these reasons, contract security is a fundamental part of our development process. We believe in carefully and iteratively auditing, testing, and documenting our contracts to ensure our contracts aren't vulnerable.

5.3 Example Zoraqle Smart Contract

To obtain a clear understanding of this section, you should have prior knowledge of the Solidity language[18] and the EVM. Here is an example of the state and behaviour of a Zoraqle contract (subject to change based on use case):

1. State variables for the oracle's address, metadata, sampling period, and owner's address

```
1  pragma solidity ^0.8.0;
2
3  import "./SafeMath.sol";
4
5  contract Zoraqle {
6      using SafeMath for uint256;
7
8      //address of device owner and oracle
9      address payable public deviceOwner;
```

[15] https://ethereum.github.io/yellowpaper/paper.pdf - Page 12, Section 9
[16] https://en.wikipedia.org/wiki/The_DAO_(organization)
[17] https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/
[18] https://github.com/ethereum/solidity

```
    address public device;

    //metadata associated with device
    bytes32 public metadata;

    //seconds between posting data
    uint32 public samplingPeriod;
```

2. Constructor to initialize the state of the contract

```
    //define constructor
    constructor(bytes32 data, address owner) {
        require(msg.sender == tx.origin && owner != msg.
            sender);
        metadata = data;
        deviceOwner = payable(owner);
        device = msg.sender;
        //set intial sampling period to 10 minutes
        samplingPeriod = 600;
    }
```

3. Function modifiers to restrict access to certain functions

```
    modifier onlyOwner() {
        require(
            msg.sender == deviceOwner,
            "Only the device owner can call this function.
                "
        );
        _;
    }
    modifier onlyDevice() {
        require(
            msg.sender == device,
            "Only the device can call this function."
        );
        _;
    }
    modifier onlySubscriber() {
        require(
            subscribers[msg.sender].authorized,
            "Only a subscriber can call this function."
        );
        _;
    }
```

4. Function to set the sampling period for the oracle

```
    //Set the sampling period of the device
    function setSamplingPeriod(uint32 period) public
        onlyOwner {
        samplingPeriod = period;
    }
```

5. Events for the client side to listen to

```
    //Define Events
    event SourceAndStreamSet(string sourceName, string
        streamName);
    event AccessRequested(address subscriber);
    event AccessGranted(address subscriber);
    event AccessRevoked(address subscriber);
    event Agreement(
        uint256 start,
        uint256 finish,
        uint256 amount,
        string indexed sourceName,
        string indexed streamName,
        address indexed subscriber
    );
    event DataRecord(
        uint256 timestamp,
        string indexed sourceName,
        string indexed streamName,
        uint16 numReadings,
        bytes32 ref,
        bytes signature
    );
```

6. Mechanism to define sources/streams and keep track of their prices

```
    //define sources, streams, and their respective prices
    struct Stream {
        bool exists;
        uint256 price;
    }
    mapping(string => mapping(string => Stream)) public
        sources;

    //function to set a source and stream in the context
        of this contract
    function setSourceAndStream(
        string memory sourceName,
        string memory streamName
    ) public onlyDevice {
        require(
            !sources[sourceName][streamName].exists,
```

```
            "Stream already exists!"
        );
        sources[sourceName][streamName].exists = true;
        emit SourceAndStreamSet(sourceName, streamName);
    }

    //function to set the price (in wei) of a specific
        stream
    function setPrice(
        string memory sourceName,
        string memory streamName,
        uint256 price
    ) public onlyOwner {
        require(
            sources[sourceName][streamName].exists,
            "Stream does not exist"
        );
        sources[sourceName][streamName].price = price;
    }
```

7. Mechanism to keep track of subscribers and their identity

```
    //define subscribers
    struct Subscriber {
        string name;
        bool authorized;
    }
    mapping(address => Subscriber) public subscribers;

    //function for a prospective subscriber to request
        access
    function requestAccess(string memory name) public {
        require(
            bytes(subscribers[msg.sender].name).length ==
                0 &&
                bytes(name).length != 0 &&
                msg.sender == tx.origin &&
                msg.sender != device &&
                msg.sender != deviceOwner,
            "Only new subscribers can call this function!"
        );
        subscribers[msg.sender].name = name;
        emit AccessRequested(msg.sender);
    }

    //function to authorize a prospective subscriber's
        request
    function authorize(address subscriber) public
        onlyOwner {
```

```
        require(
            !subscribers[subscriber].authorized &&
                bytes(subscribers[subscriber].name).length
                    != 0,
            "Only new subscribers can be granted access!"
        );
        subscribers[subscriber].authorized = true;
        emit AccessGranted(subscriber);
    }

    //function to revoke an authorized subscriber's access
    function revokeAccess(address subscriber) public
        onlyOwner {
        require(
            subscribers[subscriber].authorized,
            "Owners can only revoke access from authorized
                subscribers!"
        );
        subscribers[subscriber].authorized = false;
        emit AccessRevoked(subscriber);
    }
```

8. Function for subscribers to buy access for a specified interval of time

```
    //function for authorized subscribers to buy access to
        data
    function buyAccess(
        uint256 timestamp1,
        uint256 timestamp2,
        string memory sourceName,
        string memory streamName
    ) public payable onlySubscriber {
        require(
            sources[sourceName][streamName].price != 0 &&
                timestamp1 < timestamp2 &&
                timestamp1 < block.timestamp.mul(1000) &&
                timestamp2 <= block.timestamp.mul(1000) &&
                msg.value ==
                sources[sourceName][streamName].price.mul(
                    (timestamp2.sub(timestamp1)).div(1
                        hours * 1000)
                )
        );
        deviceOwner.transfer(msg.value);
        emit Agreement(
            timestamp1,
            timestamp2,
            msg.value,
            sourceName,
```

```
            streamName,
            msg.sender
        );
    }
```

9. Function for the oracle to post derived data packets to

```
    //function to post derived data packets
    function postData(
        string memory sourceName,
        string memory streamName,
        uint256 timestamp,
        uint16 numReadings,
        bytes32 ref,
        bytes memory signature
    ) public onlyDevice {
        require(sources[sourceName][streamName].exists, "
            Stream doesn't exist!");
        require(signature.length == 65, "Invalid Signature
            !");
        emit DataRecord(
            timestamp,
            sourceName,
            streamName,
            numReadings,
            ref,
            signature
        );
    }
```

6 Hardware Oracle

6.1 Functionality

The hardware oracle acts as an edge node in an IoT system which processes and relays sensor data to off-chain and on-chain storage. The oracle does the following:

1. Utilize the Secure Compute Node's built in supervised boot[19] and perimeter detect[20] features to prevent the oracle itself from being physically tampered with

2. Connect to a secure Ethereum node

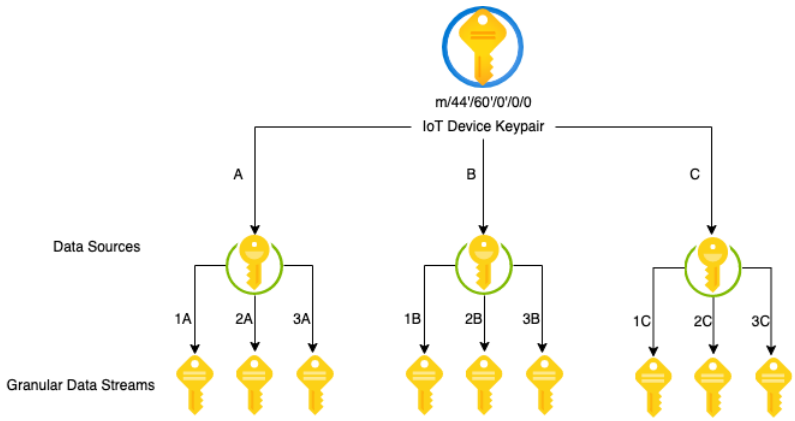
[19] https://zymbit-docs.github.io/docs-staging/branch/alpha/tutorials/supervised-boot/

[20] https://docs.zymbit.com/tutorials/perimeter-detect/hsm6/

- The node should be hosted by the oracle's owner or Zymbit

3. Create and maintain a BIP32[21] compliant wallet
   - Used to sign and fund Ethereum transactions
   - Use the BIP44[22] specification for key derivation
   - The Secure Compute Node does not allow for exporting private keys - guarantees origin of data packets
4. Publish a smart contract to the Ethereum Blockchain
   - This is only be done when the oracle is first configured
   - Each contract serves as an individual marketplace for the data produced by the oracle
5. Group sensor data for a specified interval (sampling period) and package and post the primary and derived data packets

6.2 HD Wallet Structure



7 Zoraqle API

7.1 Overview

The Zoraqle API is designed to be a RESTful service which interfaces with the off-chain storage space for primary data packets (an arbitrary cloud

[21] https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki
[22] https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki

storage solution). Owners and subscribers can make requests to the API to get data packets that they own or have purchased. Additionally, the oracles interact with the API to post primary data packets for specific data streams.

7.2 API Authentication

The Zoraqle API acts as the middleman between clients and valuable data packets, rendering API authentication a fundamental aspect of the service. The API must be able to verify the source of incoming requests and their relationship to the oracle that they are attempting to request data from.

Producing a digital signature cryptographically ties your identity to the message that you signed making it a convenient and secure way to prove one's identity. Thus, Zoraqle API requires you to attach a signature and the signed message to every API request's body. In order to ensure that the message and signature weren't compromised and reused, the client has to produce a new signature from a new message for every request they make - duplicates are rejected.

8 Zoraqle Web Application

The Zoraqle Web Application is the client side app which abstracts away the inherent complexity associated with Zoraqle's components. The goal of this application is that users can smoothly use what Zoraqle has to offer without needing a deep understanding of smart contracts, APIs, or oracles.