Decentralized Digital Currency

Chainlink, the Decentralized Oracle

by
Ariel Atar, Oliver Balfour and Agustín Golmar[1]

November 10, 2020

Department of Informatic Engineering
Instituto Tecnológico de Buenos Aires

1. Abstract

During the beginning of September 2017, Steve Ellis, Ari Juels and Sergey Nazarov published a new infrastructure called Chainlink (originally stylized as *ChainLink*), whose goal was to provide a solution to two central problems associated with the Turing-complete blockchain architecture (Ethereum style, or similar): (I) perform off-chain computations (*i.e.*, outside the blockchain), and (II) provide a truth source or decentralized oracle resistant to byzantine faults.

In this research, the current operation of the proposed system will be explored in detail, and a comparison of the functional characteristics and current properties will be made with respect to the initial proposals of the year 2017. Finally, the implementation of a specific use case called *CryptoPokes* will be detailed, which makes use of Chainlink to provide a decentralized game system based on VRFs (Verifiable Random Functions), whose objective is to generate collectible Pokémon™ cards that adhere to the ERC-721 standard (Non-Fungible Token).

Keywords: Chainlink, collusion, decentralized applications, decentralized oracle, Ethereum, freeloading, mirroring attack, non-fungible tokens, smart contracts, Sybil attack, verifiable random function.

2. Introduction

What is the truth?

Although determining the truth value of a certain assertion is a problem of general scope (considering, for example, the significant increase in recent years of *fake news*, the use of *lawfare* strategies, media manipulation and the polarization induced through of social networks and personalized digital content), the availability of information technologies incurs the same need under scenarios that operate under conditions of total mistrust or *zero trust* (see Rose et al., 2020).

The originally designed blockchain infrastructure (Nakamoto, 2008), allows guaranteeing the execution of verified transactions in a decentralized way under a Byzantine failure model, that is, assuming the existence of up to t compromised nodes (or traitors, or Byzantine nodes), a network could operate and reach consensus if its size is at least $n = 3t + 1$. That is, the transactions on the blockchain form the agreed truth, and the network architecture guarantees its preservation and evolution.

Quickly, the need to provide other types of sources of truth (which in this case only referred to electronic transactions), resulted in the creation of Ethereum (Buterin,

[1] Group ID: `0x95266274b5b876133ea4422ae9ef52c6c198925ff4df656a040908d4665bb0fc`.

2013). In this case, the blockchain evolves and becomes the consensual representation of the state of a decentralized virtual machine called EVM (Ethereum Virtual Machine). This automaton provides the ability to execute programs called *smart contracts*, where their execution and their final result operate under the consensus of the network.

Regarding Bitcoin, the system proposed by Buterin allows arbitrary code[2] to be executed, rather than merely restricting itself to transactions between different parties. However, Ethereum's own intrinsic feature imposes severe additional restrictions stemming from the same technology as its predecessor.

In particular, the source of truth is self-contained in the same blockchain: transactions are created as a product of the same generation of blocks and of the transactions recorded in them, and these transactions in turn potentially trigger the execution of smart contracts, which can operate only within the same blockchain.

The drafting of relatively complex smart contracts may involve the need to consult external sources of truth. Unfortunately, the communication channels between the *on-chain* (within the blockchain) and *off-chain* (outside of it) world are limited: the execution of methods and functions within a contract are carried out under manual or automatic external transactions (through a decentralized application or *D-App*), but it is not possible for a contract to obtain arbitrary information by its own means, but through the parameters obtained during these transactions. Furthermore, a contract can only call another contract, but not an external application directly; it can only broadcast coded messages through events, a feature of the network itself.

Therefore, the need to build an abstraction mechanism between both worlds, a *middleware* that allows functional interconnection of the applications described in contracts, and those that reside outside the blockchain, becomes evident.

The proposed solution is called Chainlink: a *decentralized oracle*.

3. Architecture

A *middleware* must allow two parties to communicate effectively. This implies both that one party can emit or generate messages, and that the other can receive or consume them. This connectivity should be bidirectional, ideally. However, the Ethereum network provides two approximate communication mechanisms:

- ❖ **Inbound Communication Mechanism:** The execution of methods and functions, the result of generating a transaction towards an address that houses a contract. This transaction may or may not carry a *payload* that the contract itself would use to compute a certain result. This mechanism is input in the sense that the information comes off-chain, and its objective is to enter the blockchain and produce a change of state.

- ❖ **Outbound Communication Mechanism:** A contract can emit events with arbitrary information as a result of a computation. This represents a way, not necessarily efficient, to move on-chain summarized[3] information outwards.

[2] The EVM is said to be Turing-capable, but due to the use of the *gas* concept, the execution of a smart contract is limited by the availability of this quantity, which actually makes the machine more like an LBA (Linear Bounded Automaton). In particular, an LBA is limited by the amount of memory available, and an EVM is limited by the number of instructions it can execute before consuming the available gas.

[3] Summarized in the sense that the information has a specific structure, a scheme. It is clear that an external application can directly monitor the generation of blocks, parsing their structure, but this is certainly a bizarre and unstructured way to acquire or extract information from the off-chain world.

In short, a contract could communicate, by emitting events, with a D-App, the contract being the one who originates the call (who acts as the client). On the other hand, a D-App can generate a transaction towards that contract in order to issue a call or return the result of a request, just as a server responds to the call of its clients.

Contracts do exist, as do web applications, servers, and distributed systems. The addition of a transport layer allows to eliminate or significantly reduce the need to interpret the concepts of operation of Ethereum (or even of any other similar blockchain), thus providing access and location transparency, since an application or contract can consult a source regardless of whether it is inside or outside the blockchain, and can also do it in a homogeneous way.

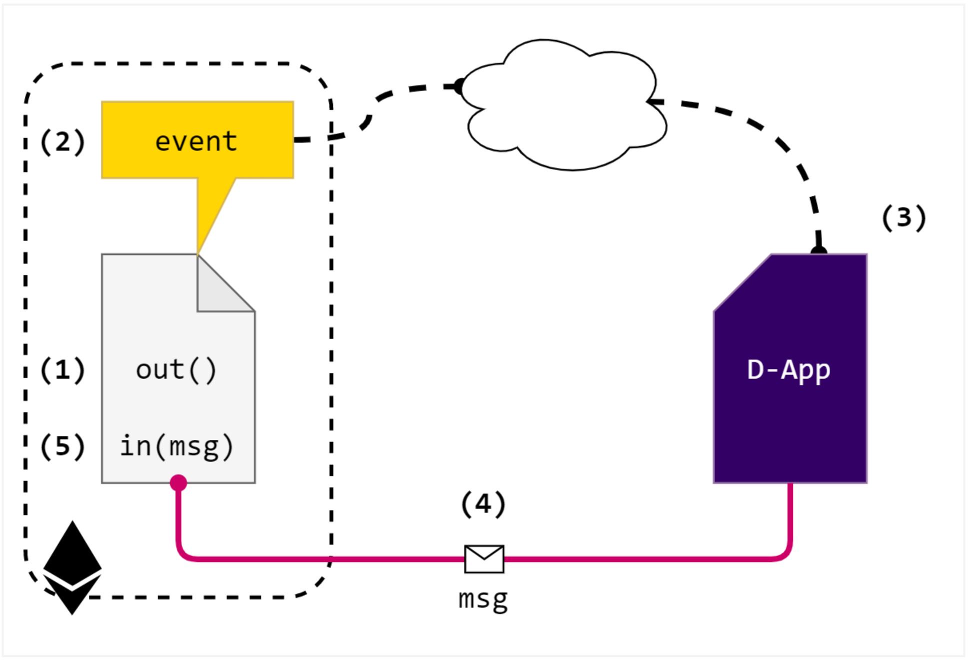

Figure 1: The basic flow of messages between the on-chain world and the off-chain world.

The architecture proposed by Chainlink is based on this basic flow of messages and can be seen in **Fig. (1)**:

1. A smart contract executes a method or function.
2. Eventually this execution produces the emission of a new event.
3. An external service monitors the broadcast event and consumes it.
4. Produces a result and executes a transaction towards an associated contract.
5. Finally the answer is transported in a call to the original contract.

This sequence of steps produces the transport of messages to the outside of the blockchain assuming that a certain service is available off-chain. The reverse model simply involves executing a transaction in order to call a method within a particular contract. This then returns the response as an event in the Ethereum *log*, or simply by accepting the transaction (i.e. without applying a *state-revert* process).

3.1. The Problem

While the basic Chainlink approach in middleware or transport layer mode works, this only solves the connectivity problem between contracts and external services.

Now, the mere existence of the blockchain, and the creation of it, is not based on the simple fact of offering a decentralized digital transaction system, but rather on the implications that occur when considering the scenarios in which that technology must operate.

So much so that the decentralized network assumes that all operations are carried out in a completely **hostile** environment. This implies that the parties are unknown (*pseudonymity*[4]), that the trust model is null (*zero trust*), and that the information flows must be validated by consensus.

Without understanding the need to agree to the existence of these assumptions, one inevitably incurs in the construction of a system that does not provide the same benefits as the underlying blockchain, with which the addition of an external system simply invalidates all the benefits that the previous one provides.

The external system must at least guarantee the property of pseudo-anonymity (for example, avoiding requesting personal information from whoever uses it). Failure to do so would violate the fact that the blockchain operates through digital signatures based on elliptic curves (ECDSA, in particular on the `secp256k1` curve), specifically to guarantee the identity privacy of those who make interactions with said infrastructure.

The trust model is null, since the operational nodes of the network are not aware of each other. If a system proposes itself as a source of truth, and an additional decentralized consensus layer is not provided, then the guarantees of the network protocols that a blockchain supports, as well as the same information with which the contracts operate, lose total validity at the moment in which transactions and contracts begin to behave based on implausible or unverifiable external data.

In short, the middleware must behave with a technology that is similar to the one supported by the blockchain.

3.2. The Bet

Chainlink's bet is a hybrid on/off-chain scheme, where direct interactions against middleware (or the Chainlink network) occur through specialized contracts, while the rest of external operations operate through consensus protocols under a system of byzantine faults.

In particular, the abstraction used as a source of truth is called an *oracle*. An oracle is a system, and as such, it can be made up of a single node or an arbitrarily complex distributed system that offers the same functionality for practical purposes.

The oracle will be a source of truth and as such, it connects with one or more sources of data. These data sources provide different views or schemes of a certain quantity, or of a certain record. The use of multiple data sources is intended to ensure the stability of the calculated quantities. This stability is given in the form of an aggregation function, in such a way that if $\delta_1, \ldots, \delta_k$ represent a set of observations on a certain quantity, where each observation comes from a different data source, then $A(\delta_1, \ldots, \delta_k)$ is the aggregation of these quantities.

The aggregations $A(\bullet)$ depend on the type of data that δ_i represents, with which it will make sense, for example, to compute an average if δ_i is the *i*-th price of *ether* in dollars (usually denoted as ETH/USD), but it will not if δ_i is an HTTP *response* with information obtained from some external service.

On the other hand, it should be noted that the data sources are not oracles, and therefore they are not concerned with the application of a consensus protocol. The

[4] It differs from the concept of *anonymity* by the fact that the latter implies the complete dissociation of a part with its activity (also known as *unlinkability*), while *pseudonymity* implies dissociation only with respect to the identity of who performs the activity.

situation is different for the systems that compute and issue the requested aggregations.

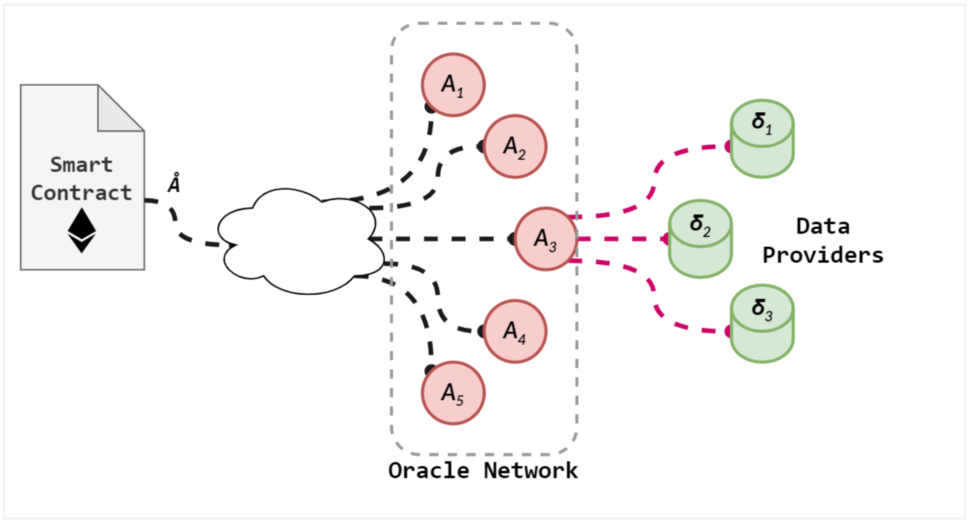

Figure 2: A contract requests the service of multiple oracles in order to compute an aggregation $\AA$. In turn, each oracle can be contacted with multiple data sources (*data providers*), with which the partial aggregations A_i are obtained.

A contract can use the middleware provided by Chainlink in order to acquire the value of an aggregation through multiple oracles. Each oracle will emit its own *partial aggregation* A_j, obtained from its own data sources (it is even possible that some data sources overlap between oracles). Now, all these aggregations must form an additional aggregation $\AA$ (see **Fig. (2)**), which will be finally sent to the contract requesting the service provided by the oracles. Computing this quantity requires a consensus process, since due to the assumption of byzantine faults, the presence of $n = 3t + 1$ oracles supports a maximum of t dishonest aggregations. If a consensus protocol were not used, a single oracle (instead of $t + 1$) could completely destroy the aggregation by issuing a partial aggregation that allows to compute the final quantity that it wants.

3.3. A Low-Coupling Design

The actual implementation of Chainlink is made up of a distributed and decentralized network of *operator nodes*. These nodes communicate with the blockchain through an Ethereum client.

As a premise in the original design proposed by Ellis, Juels and Nazarov, the development and functionalities provided by the Chainlink network must be easily scalable. This is due not only to good software engineering practice, but it is strongly tied to the fundamental objective of the model: to provide a source of truth.

But the truth may be heterogeneous enough, since not all contracts use the same amounts, not even in the same formats. Even under the existence of 2 contracts that require the same quantity, and that in turn have selected the same set of oracles from the network, it is possible that the truth for each one differs if both use different aggregations (*i.e.*, one could use a weighted average, and the other one could use the most repeated value).

What is certain is that this is not a problem with Chainlink, or its architecture, but an inherent problem with information. It is impossible to determine a set of oracles in advance that can serve all the interests of all the contracts had and for having. Because of this, it is required or rather, mandatory, for the architecture and implementation of Chainlink to be low (very low!) coupling.

For this, the operator nodes base their normal execution flow on a sequential processing model similar to the *pipes & filters* architecture, or that structure used by frameworks such as *Netty*[5], who use a design pattern called *intercepting filter*.

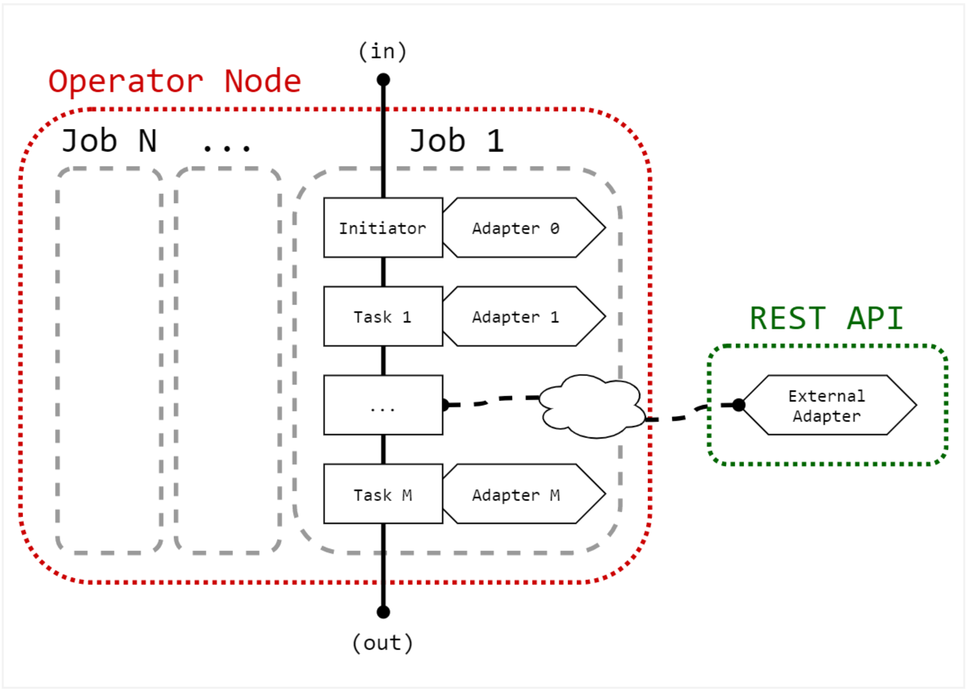

Figure 3: The structure of an operator node. It is made up of N jobs, where each job is made up of a list of tasks. Each task uses an adapter, possibly external, which exposes a REST API service.

Upon detecting a corresponding event in the underlying blockchain requesting an oracle service, the operator node consumes it and extracts an identifier called *job ID* that it carries. This ID allows you to associate a processing *pipeline* within the node in question (see **Fig. (3)**).

Each *job* is in turn composed of a list of *tasks*, separated into 2 groups: the so-called *initiators*, who are in charge of obtaining the job input data, and the rest of the tasks, who are in charge of sequentially processing the *payload* provided by the initiator.

The output of a task is the input of the subsequent one. Each task has, in turn, an *adapter* who effectively performs the transformation of the data.

By default, and thanks to the fact that Chainlink provides a standard implementation of the operator node, there is a list of *standard adapters*[6] that allow performing small tasks on a specific message. These adapters are already implemented.

As mentioned above, the need to provide flexibility when processing *requests* issued by a particular client (*i.e.*, a contract), makes it necessary to offer the possibility of coding custom adapters. To do this, a developer must write an *external adapter*, and register a *bridge* on the operator node UI and then in the official site[7]. Since these adapters are external to the node, an access URI must be registered that represents the endpoint where the service is exposed. Communication between an operator node and these adapters is carried out through a REST API, with which the adapter can be written in any platform and programming language, facilitating the extensibility of the ecosystem and their publication in the community.

[5] [https://netty.io/].
[6] The official list can be accessed at [https://docs.chain.link/docs/adapters].
[7] [https://market.link/].

Both external and internal adapters (those that are already implemented by Chainlink), can be configured through additional parameters in the corresponding job specification. Each task in turn can choose to indicate a number of *confirmations*[8] before proceeding. This specification must be encoded in JSON format.

4. Security

It is impossible to offer a source of truth, or even an approximation of it, if the entire Chainlink architecture or part of it is vulnerable, either against an active external attacker, or against other nodes on the network, those that cooperate to build a final aggregation, requested by a customer contract.

It is important to emphasize the fact that security has different aspects, and as will be seen later, even the mere presence of an architecture that guarantees the CIA triad (confidentiality, integrity and availability) in its different components, this does not imply that the client contract agrees to an honest aggregation: additional mechanisms are required to avoid tampering with the underlying working of the oracle network.

Therefore, it is necessary to subdivide the universal idea of security into different axes: that associated with the Chainlink network infrastructure, that associated with the cryptographic properties required to achieve a CIA-compliance system, and finally and perhaps the most difficult of achieve, that which implies a comprehensive vision of the concepts and the interrelationships between the components and the agents that interact with and between them.

4.1. Infrastructure

Like any distributed system and service exposed on the Internet, security should be an important concern when designing a suitable architecture.

Setting up one or more Chainlink nodes to provide a service that integrates with the decentralized network of oracles requires the following basic ingredients (for a basic example, see **Fig. (4)**):

- **Operator Node:** This is the Chainlink node that will actually receive the requests of the client contracts, send them to the corresponding job, execute its tasks, and provide a call to the Ethereum client to generate the response on the original blockchain. Chainlink already provides an implementation of this node as a Docker[9] image. Using Docker makes it easy to replicate the architecture, particularly if you have an additional orchestration system (i.e., container management), such as Kubernetes[10].

- **Ethereum Client:** The connection point between the operator node and the blockchain. Obviously, all events generated by a contract and all transactions towards a contract must occur through this client. This component can be located both locally (in the same network as the operator node), as well as externally through a provider such as Infura[11]. The client requires WebSockets, with which its use must be enabled in the same way in any proxy through which

[8] *Confirmation* is understood as the addition of a new block in the corresponding blockchain. *E.g.*, in Ethereum a new block is generated approximately every 15 seconds, with which 4 confirmations would imply a delay of up to 1 minute.
[9] This can be found in the official repository: [https://github.com/smartcontractkit/chainlink]. The code is *open source*.
[10] [https://kubernetes.io/].
[11] [https://infura.io/].

the connections pass. A list of local and external customers can be found in the Chainlink documentation[12].

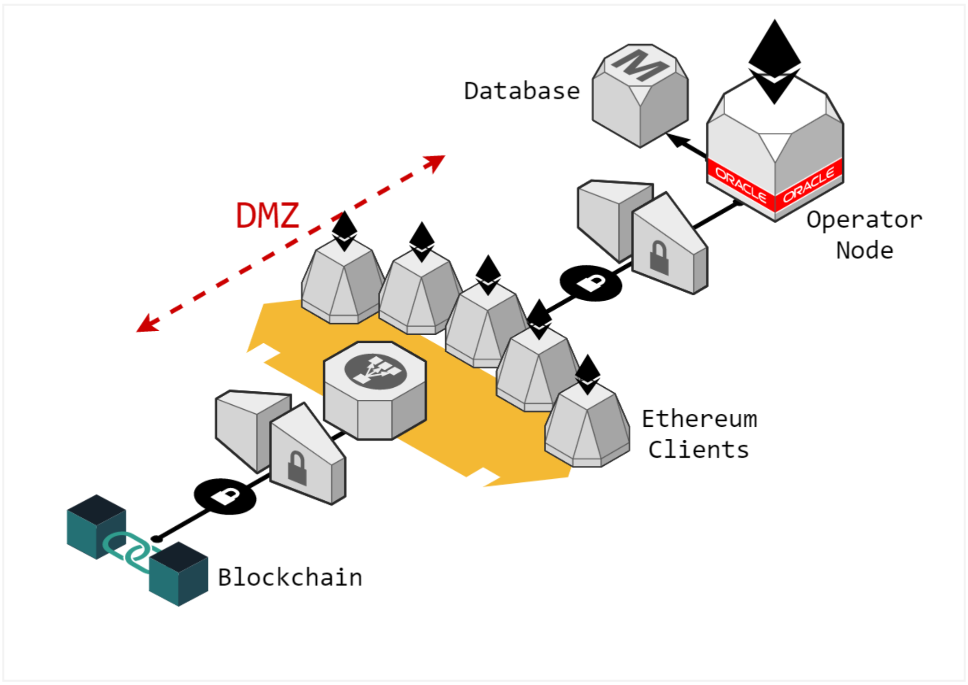

Figure 4: The recommended architecture of an oracle. All traffic runs over TLS, internally and externally. Only Ethereum clients are in the DMZ, as they are the only ones accessing the outside.

- **Database:** The operator node requires a persistence connection to a PostgreSQL database. The operations performed by the node are recorded in this database, as well as certain relevant node settings. It is important to note that only the operator node will be the one who accesses the database, so it should not be exposed to other components, nor to the global network. Since the developers of the Chainlink node decided to use a persistence layer based on ORM (Object-Relational Mapping), it is recommended to enable query logging, in order to facilitate debugging in case of failures.

- **Firewall + DMZ:** Using a firewall to protect the perimeter of the network appears to be obvious. However, it is necessary to understand exactly what the contact points are between an oracle and the external agents that consult it. In particular, on/off-chain connectivity is done exclusively through contracts and log events. This means that in reality, the Chainlink node is never directly accessed by an external user, but rather by the Ethereum client itself. On the other hand, the database is only accessed by the operator node. In short, only the Ethereum client is the one who maintains contact with the blockchain, and therefore it must cross the main perimeter of the network, that is, the firewall. Everything else is part of the backend of the oracle as a whole. Security recommendations indicate that all components that do not require external access should be behind a firewall. Since there are two firewalls, the logic indicates that a DMZ (Demilitarized Zone) scheme should be organized: the Ethereum client on the front, and the oracle behind the DMZ.

[12] [https://docs.chain.link/docs/run-an-ethereum-client].

- **TLS + PKI:** There is no use in an infrastructure that does not protect the information flows between its components. Defense in depth requires all connections to be confidential and to have integrity controls, even within the network itself. Chainlink also suggests the same idea. This is why an intensive use is made of encrypted channels between components, especially against the global network (*i.e.*, internet). The operator node must maintain an encrypted connection against the database and against the client. Access to the node control web interface must run over HTTPS (HTTP over TLS), of course. In turn, the installation of an X.509 certificate is required to identify themselves within the Chainlink network, so that customers can identify exactly who is resolving their requests. Because this certificate must be publicly verifiable, it must run within the certification granted by a member[13] of the PKI (Public Key Infrastructure). Internal certificates can be generated locally, for example, using OpenSSL.

- **Wallet + Ether:** For the oracle to provide transactions on the blockchain, a personal account or *wallet* is required. The address associated with this account will be used to retain the Ether consumed during the transactions that the oracle performs by returning a response from the client contract. The number of accounts and the distribution of the total Ether between them can be managed in different ways. It is possible to retain a secret main account to store all the available Ether and move the necessary to accounts associated with the oracle nodes (although this implies an additional cost due to the required transaction). Each node **must** have a particular account, since all of them locally register a *nonce* parameter used to resolve each requested request and therefore two operator nodes cannot share the same credential for a single wallet. As will be seen later, the account can also host Chainlink's cryptocurrency, the so-called LINK *token*, used to contract the services of an oracle, obtain rewards and deduct penalties.

Usually the availability of the oracle is an important concern, with which the replication of nodes, both operators and Ethereum clients, represents a natural way to solve this problem. Given this, usually the need arises to have some load balancing mechanism that distributes the processing, and facilitates scaling.

Regarding the database, load balancing occurs through replicas and promotions of nodes within a master-slave architecture (that is, a specific *load balancer* is not required, since the technology is part of PostgreSQL) .

The Chainlink developers went to the trouble of putting a series of additional tips on the official site[14], which, although it should be part of the knowledge already acquired by anyone developing the architecture of an oracle, it is always worth remembering:

- **Secure Access:** Although the oracle should only provide access to its Ethereum client abroad, it is possible that the administrators of the network or of some component within it need to access from another network, or even from outside. That is why it is vital to have a secure mechanism for this, being that the final solution in general should be a combination of, possibly, an SSH server with access credentials, and a VPN system (Virtual Private Network). In the same way, accesses from different components can be further restricted, if the network architecture is logically partitioned between different VLANs (Virtual

[13] *Let's Encrypt* grants valid certificates within the PKI, free of charge: [https://letsencrypt.org/].
[14] [https://docs.chain.link/docs/best-security-practices].

LANs). Access to the operator node's web interface must be done exclusively through secure mechanisms, such as SSH tunnels, or any other form of authorized confidential channel.

- **Replication and Failover:** Availability and low latency are guaranteed if operator nodes and Ethereum clients replicate. Similarly, Ethereum clients can take *snapshots* of blockchain transactions to ensure rapid recovery in the event of failure.

- **Active Monitoring:** It is impossible to offer high availability, detect intrusive attacks, misbehavior or network saturation without proper monitoring of the infrastructure. Chainlink recommends monitoring some particular aspects such as the balance in ethers of the portfolios that each node has and the error rate of the executed jobs, in addition to the classic ones (RAM, CPU and disk of the nodes, latency, bandwidth, uptime, etc.).

- **IaC (Infrastructure as Code):** A technology whose use has increased over the last few years and today it is practically a *must-have*. The idea is that the complete architecture of the network, of the components and the state of the configuration is supported by a code support, similar to the source code of an application. Examples of this in the market are Terraform[15] and AWS CloudFormation[16]. In this way, the architectural state of the oracle is maintained in a code repository, it is versioned and updated using the same methodologies as an application (branches, pull-requests, releases, issues, etc.). In turn, the fact that the architecture is recorded centrally facilitates its replication, thereby reducing configuration failures[17], and improving scalability and management of the entire system.

4.2. Low Level: Cryptographic Tools

The blockchain offers strong cryptographic guarantees of security. One question: Does Chainlink provide the same strengths?

It is important to remember the flow of information. A customer contract triggers a call to the representative Chainlink contract. This contract issues an event with the necessary information for an oracle to resolve the query. The oracle in question computes the result of an aggregation using its data sources, and issues a transaction with that aggregation in the associated contract. Finally this transfers the result to the original client.

Fig. (5) represents the real flow of execution and the interaction between contracts. It all starts with a contract that inherits from `ChainlinkClient`[18]. This contract allows you to build a new request, and to issue a `sendChainlinkRequest` call to an Oracle contract. This request must specify the address of the oracle, the job ID that you want to execute, and the *callback* function that the operator node must call to end the execution cycle.

The way in which the client contract consults an oracle is through the ERC-677 LINK token, which provides a `transferAndCall` method with which it is possible to transfer an ERC-20 token and execute a method with a certain payload, all in a single

[15] [https://www.terraform.io/].
[16] [https://aws.amazon.com/cloudformation/].
[17] See Martin Fowler's article about Phoenix Server (in bibliography).
[18] All contracts involved in the Chainlink system are publicly accessible from the official repository: [https://github.com/smartcontractkit/chainlink/tree/master/evm-contracts/src].

transaction. In this way the oracle receives a payment, at the same time as a request specification.

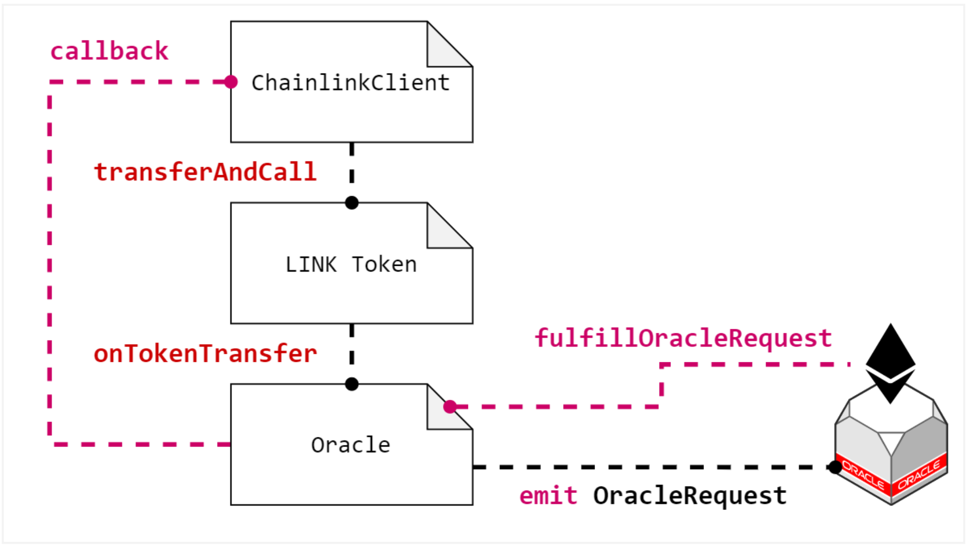

Figure 5: The basic execution flow between an on-chain client and an off-chain oracle.

The contract associated with the oracle must inherit from `Oracle`. This simply takes care of receiving the calls through a LINK token, and emitting an `OracleRequest` event with all the necessary information so that the Ethereum client can trigger the execution of the oracle through the corresponding operator node.

The client can select the oracle and its job according to a public list available in one of the official Chainlink browsers[19]. From there it extracts its *address* along with a job ID that solves the query it wants to perform. It is worth mentioning that due to the nature of the on/off-chain interface (the use of events and transactions), there is no kind of protection of the information flow between both parties, therefore, sensitive information should not be exposed in the call to an oracle. An attacker could monitor the broadcast of events in the log of a particular oracle, and therefore also monitor the behavior of a contract.

The integrity of a response and its associativity with a request previously initiated by a client contract is guaranteed by computing the hash value under the Keccak-256 algorithm of the request specification. This allows generating a hash that uniquely represents the sent message, which is non-deterministic due to the use of a unique *nonce* per oracle and client. If the response of an oracle that executes the `fulfillOracleRequest` re-entry method does not match in the value of the request ID (another hash generated by the ABI of the client contract and the nonce), neither in the hash of the request, nor in the address of the operator node that issued that transaction, then the response is rejected, and the client contract never finds out about the alleged attempted attack.

In turn, the use of the nonce avoids a *replay-attack* (*i.e.*, that an attacker re-sends an old response, captured by observing the transactions recorded in the blockchain or in the outgoing traffic from the oracle).

What happens if a failure or vulnerability is detected in an oracle's contract? In that case, the client can choose to use another oracle, modifying the address it points to (and the jobs it uses), or it can be proactive and use an indirection system called ENS[20] (Ethereum Name Service), which Chainlink supports natively.

19 [https://market.link/search/nodes].
20 [https://ens.domains/].

ENS is the equivalent of the DNS (Domain Name System) infrastructure, but in the world of smart contracts. In this case, the `useChainlinkWithENS` method allows indicating the address of the contract belonging to a *register* within the blockchain, and the address of the registered name of some oracle. In this way, ENS is used to consult the address associated with the name of the oracle, thus obtaining its contract.

Faced with the disclosure of a vulnerability and the subsequent update of the affected oracle contract, the owner of the client contract should only run `updateChainlinkOracleWithENS` to point to the updated version of the operator node.

In the event that a client executes a call to a certain oracle, and it never responds to the request, an expiration time assigned to the request allows it to be canceled. By default, the *timeout* is specified in 5 minutes. After this time interval, the customer can choose to continue waiting for the return, or issue a cancellation message, which allows him to recover the funds transferred to the oracle. This guarantees the client the investment used in the query; If it is not resolved, the customer gets the investment again (except for gas and ether consumed in transactions, both request and cancellation).

It is not possible to withdraw the funds, since they are enabled only when completing a request (in case the oracle wants to withdraw them without completing the query), or when canceling it, after the expiration time has elapsed (in case the client does not want to wait any longer for the answer).

The CIA triad (Confidentiality, Integrity and Availability) is only partially fulfilled. The **availability** depends exclusively on the replication of oracle nodes, and the availability of the truth sources (or *data providers*) that the oracles access depends on their infrastructure also having high availability. The blockchain natively provides high availability due to the replication of *miner-type* nodes, those that collect transactions and distribute the new blocks.

The **integrity** of the transactions is acquired within the blockchain thanks to the fact that all the modifications of the decentralized state are computed under the digital signatures of all the agents involved. As a side effect due to the fact that the integrity is computed by a digital signature instead of a simple MAC (Message Authentication Code), in addition to integrity, the message flow is *authenticated* and is *non-repudiable*. Off-chain communication, on the other hand, gets the signed events from the blockchain (in full), and communicates with its providers through secure channels (usually over TLS). As long as the transfers, both internal within the oracle's perimeter network, and external, between the oracle and its data providers operate under a secure channel, integrity will be guaranteed.

But Chainlink's weakest point has to do with the concept of **confidentiality**. Naturally, the blockchain is not confidential at all, except for the pseudo-anonymity of those who carry out transactions and issue smart contracts. There is then a visible flow of information, particularly when generating the log events that will be consumed by the off-chain oracles. On the other hand, within the perimeter network and outside it (again, in contact with the data providers), confidentiality is provided thanks to the use of TLS (Transport Layer Security).

What happens between the generation of a query and its publication in the event log? It is possible to imagine that a client contract stores the public key of a particular oracle, and therefore can encrypt the request before transmitting it to the oracle contract. In this way, the posted log event will only contain an encrypted payload, which only the oracle can open. The problem is that the information to generate this request is necessarily on-chain. Due to this, even though the request is encrypted by a contract, the message (just before being encrypted) is not encrypted, and therefore it is public.

If the message is entered as a reason for an external transaction, then there is a possibility that this trigger message is already encrypted. In this case, the contract must make the decision to issue a query to an oracle based exclusively on a ciphertext. The only way to do this is by using *secure multiparty computation*[21], part of the state of the art in cryptography, where it is possible to compute on a ciphertext, without ever revealing the content. However, this would simply transform the client contract into a mere proxy between two off-chain systems: one would send a transaction whose payload is encrypted to the contract, and the other would receive the log event associated with said payload. What's the point of using the handrail-only contract? Perhaps to attest that indeed that transaction existed, but nothing more.

The issue of contract confidentiality is inherent in the blockchain, and it is not Chainlink's fault. In the words of the authors:

> *Confidentiality is fundamentally hard to achieve in any oracle system. If an oracle has a blockchain front end such as a smart contract, then any queries to the oracle will be publicly visible. Queries can be encrypted on-chain and decrypted by the oracle service, but then the oracle service itself will see them. Even heavyweight tools such as secure multiparty computation, which permits computation over encrypted data, can't solve this problem given existing infrastructure.*
>
> **- Ellis, Juels, Nazarov (2017, §6.1: *Confidentiality*, p. 22).**

As a consequence, it should be clear that whoever decides to use the decentralized oracle must know that there will be a necessary, non-confidential information flow at the point of intersection between the on-chain and off-chain world.

4.3. High Level: Tamper-Proof Resistance

The previous section made a description of the security associated with a simple connectivity between an on-chain contract and an off-chain oracle. Additionally, Chainlink allows consultations under consensus, where a client hires multiple oracles and makes a final aggregation on the results in order to obtain an answer that is as close as possible to the real truth.

This introduces numerous high-level security considerations, that is, associated with the delivery protocol, which are not resolved by the cryptographic mechanisms evaluated in the previous section.

Remembering that the i-th aggregation of a given oracle is of the form $A_i(\delta_1, \ldots, \delta_k)$, where $\delta_1, \ldots, \delta_k$ represent the observations obtained from its k data providers, then the objective is to construct a new aggregation $Å = A(A_1, \ldots, A_n)$, where $Å$ is the final value that the customer contract receives, and n is the number of oracles involved.

There are at least 2 ways to compute $Å$. If $Å$ is computed on-chain, then each oracle must be in charge of obtaining the requested quantities from its data sources, computing a particular aggregation A_i, and then sending it to the aggregation contract, using a transaction. When the contract receives a sufficient number of aggregations, it computes $Å$ and sends that value to the requesting customer.

This scheme is completely logical, but falls into various vulnerabilities. When one of the selected oracles sends its estimated A_i value to the aggregation contract, an opposing oracle that is part of the selected group can observe the transaction and enter

[21] [https://en.wikipedia.org/wiki/Secure_multi-party_computation].

the same A_i value, without having computed anything at all. This type of attack is known as *freeloading*.

The consequence of allowing freeloading in a competitive aggregation scheme produces a behavior change in oracles: they **delay the sending** of aggregations, to reduce the risk of plagiarism. Ultimately, the whole system becomes slower, and the aggregate responses have low diversity because there is a higher interest in copying and waiting.

The classic solution to this type of problem is the implementation of a classic *commit-and-reveal* protocol. With the help of cryptographic primitives, it is possible to get *n* oracles to send their aggregations to a contract without the possibility of incurring freeloading. The scheme is as follows:

1. Ask all *n* oracles to produce their aggregations.
2. The oracles compute their aggregations A_i.
3. The oracles send the contract $H(A_i)$, where $H(\bullet)$ is a hash function.
4. When the contract receives the *n* hashes, it requests the final aggregations.
5. The oracles send the original add-ons A_i.
6. The contract accepts only those aggregations whose hash matches.
7. The contract computes the final Å aggregation.

Naturally in Ethereum, $H(\bullet)$ is the Keccak-256 algorithm. This scheme works, although it must be considered that the system also operates on Byzantine faults, with which if $n = 3t + 1$, then there are at most *t* possible traitors. The Chainlink developers apply this reasoning at each stage[22], so out of *n* selected oracles, only $2t + 1$ will send their respective hashes, then $t + 1$ their final aggregations, and finally at least one of them will be honest.

This presents a couple of problems. The first is that the number of transactions is high, since each oracle must carry out at least 2 transactions (one to send the hash and another to send the final aggregation), and therefore all these expenses are transferred to the requesting client. The second problem is particular to Chainlink: its aggregation scheme **does not use** *commit-and-reveal*, so freeloading is practically possible. The oracles send their responses to the aggregation contract[23], and upon achieving the minimum required, an aggregation is computed and sent to the client. And that's it!

The high-level security issues don't end there. There is a whole class of manipulations resulting from using data providers and oracle networks in a dishonest way. These situations are highly difficult to control because the decentralization of the network prevents the operator nodes from being *audited*, not to mention the amount of resources that such a certification would potentially require.

The idea of operating in bad faith on the oracle network comes in different flavors. If an adversary controls the largest number of oracles on the market, then he can manipulate the information of all contracts that require consulting the quantities that the adversary emits. This attack is similar to the *51% attack* on a blockchain, but it is called *Sybil attack*, since a hidden adversary takes the face of multiple oracles, where they identify themselves as independent operators, although in reality they are not.

A simpler version of the Sybil attack implies that multiple operators, who are actually independent, are subjected to a process of *collusion*, with which they coordinate the falsification or misrepresentation of certain types of data, possibly for economic reasons (although for the analysis, the reason is irrelevant).

[22] This is because they are based on the fact that no more than *t* traitors can change from round to round. Therefore in each batch of messages *t* additional nodes are mistrusted.

[23] See for example the `submit` method of the `FluxAggregator` contract (version *0.6*).

An even cheaper version is called *mirroring*. In this case, the oracles are declared independent as in the previous situations, but they all use the same data sources (or even just one). This type of situation is deliberate, and although it does not allow arbitrarily manipulating the behavior of the contracts that consume from these oracles (since the data provider is who emits the quantities), the performance of the system degrades due to the lack of diversity in the data sources.

Finally, and assuming that an oracle does not act in bad faith, it may be the case that an unintentional mirroring effect occurs: 2 or more oracles simply use the same data provider. This case is called *overlapping*, and it is not actually an attack in itself as it is not deliberate. It can occur due to a lack of controls, a lack of data sources or an excess of oracles. See **Fig. (6)** for a crude explanation.

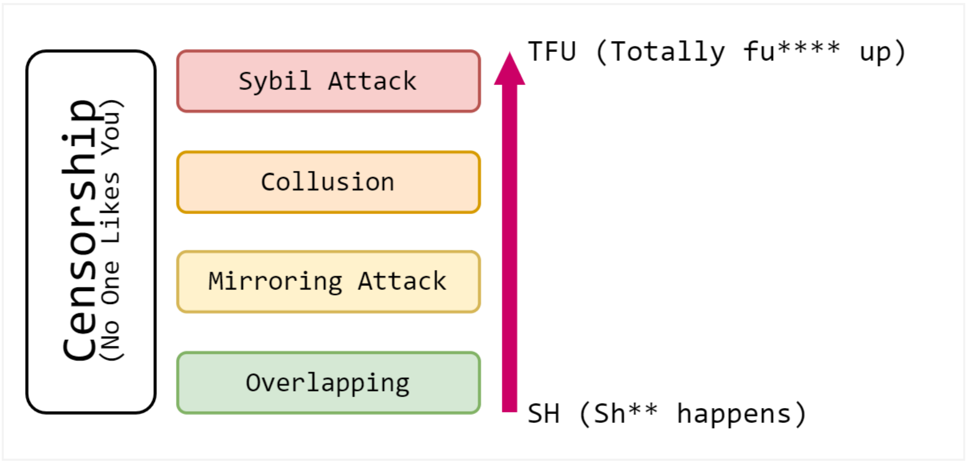

Figure 6: Severity of tampering attacks.
The oracles only wish they were unlucky, at best.

How to reduce the prevalence of this type of high-level attack? On a technical level it is practically impossible or expensive, at least to date. One way to mitigate these situations is by using the KYC (Know Your Customer) technique, in which an institution makes a considerable effort to ensure that a certain network of oracles operates legitimately (*i.e.*, without engaging in any of the mentioned dishonest behaviors). This involves conducting formal audits, monitoring its operation, creating curated lists, offering reputation metrics, validating the identification of a network, among many others.

If an oracle can be classified as *honest* then the institution delivers a certification that increases the confidence of those who consume its services.

A real implementation of this certification service is provided by Nodary, from CLC Group (Benligiray et al., 2019). To avoid a Sybil attack it requires an oracle to certify an operator node under its domain and associate it with a natural person. In this way, it is not possible to arbitrarily replicate the number of oracles, since there must be the same number of natural persons who certify them. However, it is possible to build a *demultiplexing*, which allows building a *nodarized* node, whose objective is to operate as a load balancer. In this case, an oracle can be made up of multiple operators, but all under a single identification, with which the Sybil concept does not apply. The downside is that the anonymity feature, and even the operator's pseudo-anonymity, one of the benefits of the blockchain, is clearly lost.

Last but not least, there is a possible high-level vulnerability associated with the availability of the oracle network, which perhaps is transversal to all of the above situations. This is called *censorship*, and it occurs when one or more oracles deny service to certain contracts (this is possible thanks to the oracles knowing their addresses). An oracle network can only guarantee *censorship resistance* if it has high availability and replication, that is, if there are several independent oracles available to resolve the same query.

5. Governance

In order to be able to perform *off-chain* operations, the Chainlink network pays the operator nodes to return *off-chain* data, such as sports results, weather updates, among other possibilities. On the other hand, a smart contract from another network such as Ethereum can pay an operator node to carry out these operations. The aforementioned payments are made through the payment of *LINK tokens*, the price of which is set based on the supply and demand of off-chain resources. LINK tokens are ERC20 tokens.

So far we have already talked about the security that Chainlink offers us. This security that we have on Chainlink gives us availability and correctness as long as the number of defective oracles providing responses to requests is less than or equal to *t*.

Now, as Chainlink users we need to make sure we choose the most reliable oracles and thereby reduce the probability of having a number of defective oracles greater than *t*. To measure the reliability of the oracles, we are going to talk about different mechanisms that Chainlink provides us to choose the most recommended oracles and with the lowest error rate.

5.1 Validation System

This system aims to obtain metrics about the operation of the oracles. For this, the validation system monitors the oracles with respect to availability, where the system records if the oracle does not respond in the intended times, and on the other hand, the oracles are evaluated with respect to correctness, where assesses the number of wrong answers an oracle provides, also based on the deviation from the original correct answer.

In order to evaluate the correctness, a reward system is established through a smart contract that rewards the oracles for giving evidence of answers deviating from the correct one, therefore, since the oracles are encouraged to report the incorrect answers, we can measure the correctness of the system. This system can be implemented because the oracles digitally sign their responses.

On the other hand, in order to measure availability, oracles are asked to digitally sign testimonies about the responses of other oracles, and oracles that present testimony about the lack of effectiveness and response of any of the oracles are rewarded.

5.2 Reputation System

This system[24] allows to establish a reputation for oracles, so that users can choose between the oracles with the best reputation. This system is strongly based on the validation system explained above, and on the other hand it also takes into account users' ratings towards oracle providers and nodes. In this way, oracles can constantly raise and lower their reputation.

In this way, the oracles are encouraged to work correctly and quickly, giving high rates of availability and performance. If at some point an oracle lowers performance, automatically its reputation will fall and therefore users will not choose it.

[24] See [https://reputation.link/].

5.3 Certification Service

This service is not yet mature and is currently being implemented[25]. The service aims to certify oracles that they do not carry out attacks, in particular *Sybil and mirroring attacks* are analyzed, whereby an adversary who handles several oracles can provide incorrect information, and this can result in a worse problem if the adversary handles a number of oracles greater than *t*. To address the issue, this service relies on high-quality oracle providers, and is compared against these oracles to assess the correctness of responses and eliminate attacks.

6. Provided Services

Now we are going to talk about 3 services that Chainlink provides us, which are detailed below. All this information was obtained from the official Chainlink documentation[26].

6.1. Consuming an API

Chainlink allows contracts to access any external data source, through our decentralized oracle network. In this way the contract can communicate with the outside and access data such as sports results, the weather and other data of public knowledge that are part of the *off-chain* data.

Creating *smart contracts* is already a complicated task, and making them also compatible with *off-chain* data is even more complicated. Therefore, Chainlink created a *framework* for this purpose.

Chainlink's decentralized oracle network provides smart contracts with the ability to push and pull data, facilitating the interoperability between on-chain and off-chain applications.

Now let's explain how to make an HTTP GET request to an external API from a smart contract. To consume an API response, the contract should inherit from `ChainlinkClient`. This contract exposes a struct called `Chainlink.Request`, which the contract should use to build the API request. The request should include the oracle address, the job id, the fee, adapter parameters, and the callback function signature.

We can also use an *existing* Oracle Job in order to make the smart contract code more succinct. The contract should own enough LINK to pay the specified fee.

Oracles enable smart contracts to retrieve data from the outside world. Each oracle node can be configured to perform a wide range of tasks depending on the adapters it supports. For example, if your contract needs to make an HTTP GET request, it needs to use an oracle that supports the HTTP GET adapter.

Oracles jobs can be specialized even further by implementing the configuration using External Adapters. For example, an Oracle job could implement URL, parameters, and conversion to Solidity compatible data, to retrieve a very specific piece of data from a specific API endpoint.

6.2. Multiple Oracles: Price Feed

[25] See Nodary: [https://www.clcg.io/nodary/].
[26] [https://docs.chain.link/docs].

Chainlink Price Feeds are the quickest way to connect your smart contracts to the real-world market prices of assets. They enable smart contracts to retrieve the latest price of an asset in a single call.

To consume price data, the smart contract should reference `AggregatorV3Interface`, which defines the external functions implemented by Price Feeds. This is to get the last price of an asset, but getting the latest price is not the only data that can be retrieved from aggregators. You can also retrieve historical price data. Price Feeds are updated in rounds. Rounds are identified by their *roundId*, which increases with each new round. Knowing the *roundId* of a previous round allows contracts to consume historical price data.

6.3. VRF (Verifiable Random Function)

Chainlink VRF (Verifiable Random Function) is a provably-fair and verifiable source of randomness designed for smart contracts. Smart contract developers can use Chainlink VRF as a tamper-proof PRNG to build reliable smart contracts for any applications which rely on unpredictable outcomes, such as Blockchain games and Random assignment of duties and resources.

Chainlink VRF enables smart contracts to access randomness without compromising on security or usability. With every new request for randomness, Chainlink VRF generates a random number and cryptographic proof of how that number was determined. The proof is published and verified on-chain before it can be used by any consuming applications. This process ensures that the results cannot be tampered with or manipulated by anyone.

To consume randomness, the contract should inherit from `VRFConsumerBase` and define two required functions:

- `requestRandomness`, which makes the initial request for randomness.
- `fulfillRandomness`, which is the function that receives and does something with verified randomness.

7. Future Improvements

The current state of Chainlink is still too premature, and any project that wants to evaluate the possibility of integrating such a system into its architecture should at least consider the aforementioned features and implications.

Beyond being a project in its younger years, the authors of Chainlink are aware of the security aspects involved, and of the possible improvements to be made. Some of them are mentioned:

- **Off-Chain Aggregation (OCA):** The basic *commit-and-reveal* protocol that Chainlink implements in its `FluxAggregator` contract does not provide protection against freeloading. In the first place, it would be useful if it does this, and uses the cryptographic scheme indicated in these pages or even in the paper written by Ellis, Juels and Nazarov. But there is another improvement not yet implemented which consists of moving the processing towards the off-chain world. The advantages are clear: it is cheaper. In particular, an off-chain aggregation scheme is proposed by the authors in the foundational Chainlink whitepaper (Ellis, Juels, Nazarov, 2017, §A.1: *OCA Protocol*). This scheme is based on the use of *threshold signature*, a protocol that allows several parties to sign the same payload, where the signature can be verified under a unique key (see **Fig. (7)**). It is the analog to Shamir's threshold encryption, but in the world of digital signatures. In this way, it would be possible for each intervening oracle to sign an aggregation with its key, but for the on-chain contract to verify that it is really valid, that is, that it has been signed by all the intervening oracles. This reduces costs because only one consensual message is sent to the customer, rather than one for each oracle. The cost reduction also translates into a benefit for customers who consume the service. On the other hand, it has additional implications: oracles must employ a key distribution system, and they are also subject to network monitoring to ensure that at most one signature is sent to the client.

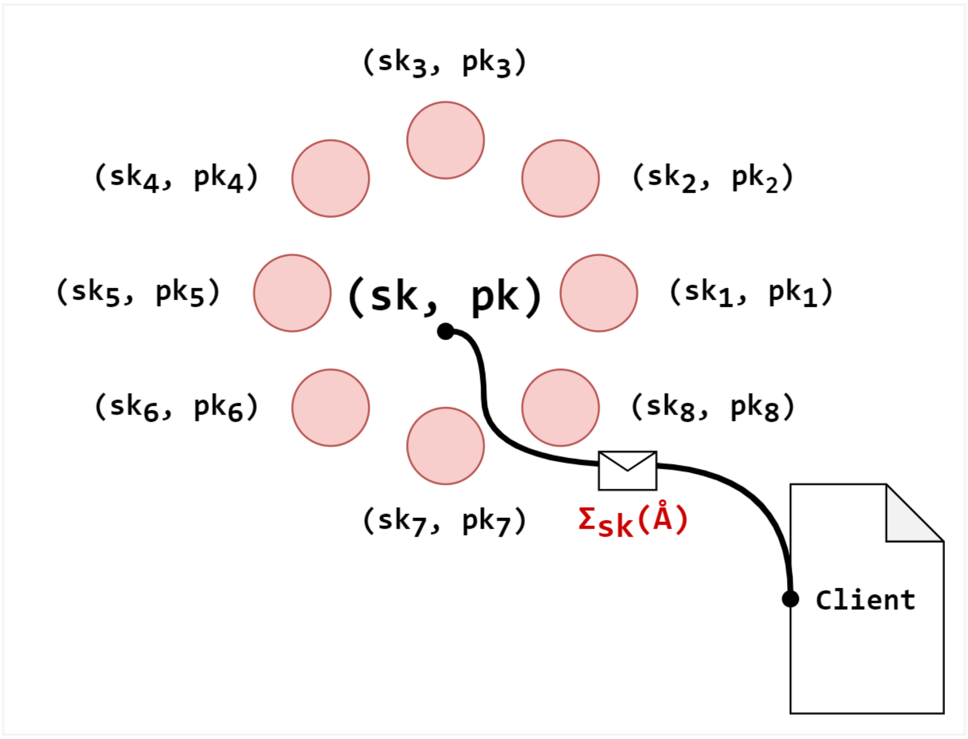

Figure 7: A threshold digital signature scheme, in an OCA protocol.

- **Signed Data:** An improvement proposed by the authors of Chainlink consists of not only signing the oracle aggregations, but also the data obtained from the data providers. This would allow to obtain an additional attestation of originality on the data, as well as a way to verify the overlapping or the application of a mirroring attack, since if two data are signed under the same key, then they come from the same source. Obviously it is not that simple, since each oracle can use multiple data sources, and the total number of firms involved can generate substantial traffic.

- **Sybil Attack Resistance:** Perhaps one of the most difficult problems to solve. Nodary tries to reduce the attack surface of these attacks by employing some guarantee of identification, although this clearly undermines the pseudo-anonymity that the blockchain provides. In general, this type of attack is reduced through economic incentives, since to date there is no formal test that can fully distinguish between agents acting in good or bad faith. It should be more profitable to behave honestly than to betray the trust of customers. Some incentive systems are based, for example, on distributed attestations, where the oracles issue some proof that they actually received a valid aggregation from their neighboring oracles, availability tests, data originality tests, etc. Definitely moving in the direction of building a Sybil-resistant system would add a lot to the Chainlink network.

- **Penalizations:** Just as there may be financial incentives for those who behave honestly, there may be penalties for those who do not. This further reduces the possibility of collusion or manipulation of the network, since now it is not only possible to win: who does not win also loses money. There is a *trade-off* between penalizing those who do not behave correctly, and reducing the latency of the consensus protocol between oracles. Latency is reduced by issuing a final aggregation as soon as possible, that is, for a system with $n = 3t + 1$ oracles, an aggregation is issued by obtaining $t + 1$ results in the final round of consensus. This methodology is called *asynchronous consensus*. The consequence is that those who act honestly but are not part of the first $t + 1$ answers will be penalized, even to the detriment of their good actions. The synchronous version waits for all responses, which is fair on penalties and rewards, but necessarily slower.

8. Additional Resources

Some additional resources are included to keep the reader informed about the activity in the Chainlink project:

- **Chainlink's Pivotal Tracker**
  [https://www.pivotaltracker.com/n/projects/2129823]
  Where the features that developers are currently implementing are published, and those that could be implemented in the future (not necessarily near). In turn, each feature can be consulted with additional detailed information. For example, the application of the threshold signature protocol can be accessed at [link].

- **Provable™**
  [https://provable.xyz/]
  Another system of oracles, where its security lies in the issuance of *authenticity proofs*, some of them based on software tests and others based on attestations of virtual machines that run in a confinement provided by specific hardware. However, the veracity of these systems is certainly centralized, and depends exclusively on the security or trust of these components. A confined execution test can guarantee that a particular program executed a certain code, and that its emitted response is signed, but in short it is necessary to believe that the confinement is real.

- **API Reference**
  [https://docs.chain.link/reference]
  The official API guide provided by Chainlink operator nodes. This allows you to build applications that interact with the operator node, both locally and externally. As the API operates over HTTPS, it is possible to develop applications in any language and platform.

- **Glossary**
  [https://docs.chain.link/docs/glossary]

Anyone who has read the founding whitepaper and the Chainlink website will notice slight variations in the naming of certain concepts. The Chainlink Glossary is the official source that provides the exact definitions of each concept involved in this technology.

- **GitHub Wiki**
  [https://github.com/smartcontractkit/chainlink/wiki]
  The official Chainlink code repository also provides a wiki with additional relevant information not found on the website, such as the interaction between some contracts, or certain details about the architecture.

9. Implementation: CryptoPokes

Chainlink oracles can be used for a wide variety of use cases. Any smart contract that needs to interact outside of the blockchain to make payments, consult API information or see updated status of events will make use of Chainlink services.

The project[27] implements ERC 721 Non-Fungible Tokens that, unlike other types of Tokens such as Bitcoin or Ethereum, they cannot be partially repeated and transferred. These types of tokens are commonly used to represent unique objects. In this case, the tokens represent collectible interchangeable cards from the Pokémon saga.

The objective is to be able to create and draw a special card to reward a user participant within the application. The lucky winner will be selected from among all users who have at least one card from a list containing all the cards that were assigned to some user address. In this way, users who have more cards are rewarded for their loyalty and are more likely to win, since they participate once for each card they have.

Historically, obtaining a random number within the blockchain has always been a security problem, since existing methods could be hacked, and therefore impossible to be implemented in any random situation that assumes that no one knows the result in advance. In fact, one of the most widespread and common solutions was to use the current Blockhash, this represented a huge threat since a group of miners could easily decide whether to discard a block at their discretion, managing to influence the result of the random number. For example, if the project offered a card with a very high market value as a reward, the attackers could maliciously discard all the blocks that were not convenient for them to mine, incentivized by the possibility of obtaining this higher reward by mining the block. with the correct hash.

This is why Chainlink is ideal for this type of project as it offers a provably fair (verifiable proof of randomness). Currently, there are already contracts that are using Chainlink to mine special items. This is the case of Polyient Games, which in June of this year announced through Twitter the integration of Chainlink VRF to randomly reward its users with NFTs and other types of prizes to users who have a membership.

The decentralized application was created using Buidler's boilerplate[28], today known as Hardhat. Every user who connects to the site for the first time is asked to connect their virtual wallet and is given a first letter to start collecting. The game administrator then proceeds to draw a card that will be minted and delivered to a user of the application. In the draw contract you can see the random number generated and the id of the card that the winner had, with which he participated in the draw. In addition, the results of the draw are saved on a map where the drawn cards appear together with the ids of the "ticket" cards with which the respective winners entered the draw.

[27] Visit the live version: [https://obalfour.github.io/CryptoPokes/].
[28] Tutorial is available here: [https://hardhat.org/tutorial/hackathon-boilerplate-project.html].

The card draw works as follows: the administrator calls the Chainlink oracle with the number of the card to be drawn and a seed, the contract makes the initial request for randomness and the id of the special card to be drawn is saved. Once the verified randomness is obtained, the modulus is calculated by the number of cards currently dealt in the game. The random number generated for this draw is temporarily saved and the winning id with the card that was drawn is saved on the map. The item is minted by interacting with the Token contract and distributed to the winner. It is important that the contract has enough Link Tokens to complete the operation.

Additionally, Chainlink could be used for various improvements within the game. In the first place, it is possible to achieve that special cards can be minted according to special events or seasons of the game. For example, on Halloween, *shiny* cards could be minted for dark-type characters. Another possible case is that according to the performance that a user has within the game of a tournament or competition, he could obtain a special card. In addition, if we use augmented reality, we could make the damage of a card increase or decrease according to the environment in which it is found, if it is a wet area, the Water-type fighters could have a certain advantage. Lastly, cards can be offered as a reward for achieving certain goals outside of the blockchain.

10. Conclusions

A research was conducted on Chainlink technology, its fundamentals, risks, advantages and disadvantages, in order to facilitate decision making when implementing this system within your business model or D-App. Possible attacks and vulnerabilities of the system were described, ranking them according to their perceived severity and complexity. A list of possible future enhancements was listed, many of which have already been identified by the authors of the Chainlink platform. Finally, a particular implementation of the VRF (Verifiable Random Function) service was made and described, based on the generation of NFTs (Non-Fungible Tokens).

11. Bibliography

Antonopoulos, A. M. and Dr. Wood, G. (2018). *Mastering Ethereum: Building Smart Contracts and DApps* (1.ª ed.). O'Reilly Media, Inc. [link]

Benligiray, B. and Connor, D. and Tate, A. and Vänttinen, H. (2019). *Honeycomb: An Ecosystem Hub for Decentralized Oracle Networks*. CLC Group. [link]

Bormann, C. and Hoffman, P. (2013). *RFC 7049: Concise Binary Object Representation (CBOR)*. Internet Engineering Task Force (IETF). [link]

Buterin, V. (2013). *A Next-Generation Smart Contract and Decentralized Application Platform*. Ethereum Foundation. [link]

Collins, P. (2020, October 20). *Create Dynamic Non-Fungible Tokens (NFT) Using Chainlink Oracles*. Retrieved November 09, 2020, from [link]

Collins, P. (2020, October 29). *How to Generate Truly Random Numbers in Solidity and Blockchain*. Retrieved November 09, 2020, from [link]

Ellis, S. (2017). *ERC: transferAndCall Token Standard*. (Draft) [link]

Ellis, S. and Juels, A. and Nazarov, S. (2017). *ChainLink: A Decentralized Oracle Network* (v1.0). [link]

Entriken, W. and Shirley, D. and Evans, J. and Sachs, N. (2018). *EIP-721: ERC-721 Non-Fungible Token Standard*. Ethereum Improvement Proposals, N° 721. [link]

Fowler, M. (2012). *PhoenixServer*. ThoughtWorks®. [link]

Nakamoto, S. (2008). *A Peer-to-Peer Electronic Cash System*. Bitcoin Project. [link]

Papadopoulos, D. and Wessels, D. and Huque, S. and Naor, M. and Včelák, J. and Rezyin, L. and Goldberg, S. (2017). *Making NSEC5 Practical for DNSSEC*. The International Association for Cryptologic Research. [link]

Rose, S. and Borchert, O. and Mitchell, S. and Connelly, S. (2020). *Zero Trust Architecture*. NIST Special Publication 800-207. [link]

Russo, C. (2020, July 13). *Chainlink VRF to Support NFT Innovation within Polyient Games Ecosystem*. Retrieved November 09, 2020, from [link]