ChainLink

A Decentralized Oracle Network

Steve Ellis, Ari Juels[†], and Sergey Nazarov

4 September 2017 (**v1.0**)

Abstract

Smart contracts are poised to revolutionize many industries by replacing the need for both traditional legal agreements and centrally automated digital agreements. Both performance verification and execution rely on manual actions from one of the contracting parties, or an automated system that programmatically retrieves and updates relevant changes. Unfortunately, because of their underlying consensus protocols, the blockchains on which smart contracts run cannot support native communication with external systems.

Today, the solution to this problem is to introduce a new functionality, called an *oracle*, that provides connectivity to the outside world. Existing oracles are centralized services. Any smart contract using such services has a single point of failure, making it no more secure than a traditional, centrally run digital agreement.

In this paper we present ChainLink, a decentralized oracle network. We describe the on-chain components that ChainLink provides for contracts to gain external connectivity, and the software powering the nodes of the network. We present both a simple on-chain contract data aggregation system, and a more efficient off-chain consensus mechanism. We also describe supporting reputation and security monitoring services for ChainLink that help users make informed provider selections and achieve robust service even under aggressively adversarial conditions. Finally, we characterize the properties of an ideal oracle as guidance for our security strategy, and lay out possible future improvements, including richly featured oracle programming, data-source infrastructure modifications, and confidential smart-contract execution.

Contents

1 Introduction

Smart contracts are applications that execute on decentralized infrastructure, such as a blockchain. They are tamperproof, in the sense that no party (even their creator) can alter their code or interfere with their execution. Historically, contracts embodied in code have run in a centralized manner that leaves them subject to alteration, termination, and even deletion by a privileged party. In contrast, smart contracts' execution guarantees, which bind all parties to an agreement as written, create a new and powerful type of trust relationship that does not rely on trust in any one party. Because they are self-verifying and self-executing (i.e., tamperproof as explained above), smart contracts thus offer a superior vehicle for realizing and administering digital agreements.

The powerful new trust model that smart contracts embody, though, introduces a new technical challenge: *connectivity*. The vast majority of interesting[27][1] smart contract applications rely on data about the real world that comes from key resources, specifically data feeds and APIs, that are external to the blockchain. Because of the mechanics of the consensus mechanisms underpinning blockchains, a blockchain cannot directly fetch such critical data.

We propose a solution to the smart contract connectivity problem in the form of ChainLink, a secure oracle network. What differentiates ChainLink from other oracle solutions is its ability to operate as a fully decentralized network. This decentralized approach limits the trust in any single party, enabling the tamperproof quality valued in smart contracts to be extended to the end-to-end operation between smart contracts and the APIs they rely on. Making smart contracts externally aware, meaning capable of interacting with off-chain resources, is necessary if they are going to replace the digital agreements in use today.

Today, the lion's share of traditional contractual agreements that have been digitally automated use external data to prove contractual performance, and require data outputs to be pushed to external systems. When smart contracts replace these older contractual mechanisms, they will require high-assurance versions of the same types of data inputs and outputs. Examples of potential next-generation smart contracts and their data requirements include:

- *Securities smart contracts* such as bonds, interest rate derivatives, and many others will require access to APIs reporting market prices and market reference data, e.g. interest rates.

[1]The main use of smart contracts in Ethereum today is management of tokens, which are a common functionality in most smart contract networks. We believe that the current focus on tokens to the exclusion of many other possible applications is due to a lack of adequate oracle services, a situation ChainLink specifically aims to remedy.

- *Insurance smart contracts* will need data feeds about IoT data related to the insurable event in question, e.g.: was the warehouse's magnetic door locked at the time of breach, was the company's firewall online, or did the flight you had insurance for arrive on time.
- *Trade finance smart contracts* will need GPS data about shipments, data from supply chain ERP systems, and customs data about the goods being shipped in order to confirm fulfillment of contractual obligations.

Another problem common to these examples is the inability for smart contracts to output data into off-chain systems. Such output often takes the form of a payment message routed to traditional centralized infrastructure in which users already have accounts, e.g., for bank payments, PayPal, and other payment networks. ChainLink's ability to securely push data to APIs and various legacy systems on behalf of a smart contract permits the creation of externally-aware tamperproof contracts.

Whitepaper roadmap

In this whitepaper*, we review the ChainLink architecture (Section 2). We then explain how we define security for oracles (Section 3). We describe the ChainLink approach to decentralization / distribution of oracles and data sources (Section 4), and follow with a discussion of the four security services proposed by ChainLink, as well as the role played by LINK tokens (Section 5). We then describe a proposed long-term development strategy, which includes better confidentiality protections, the use of trusted hardware, infrastructure changes, and general oracle programmability (Section 6). We briefly review alternative oracle designs (Section 7), and conclude with a short discussion of the design principles and philosophy guiding ChainLink development (Section 8).

2 Architectural Overview

ChainLink's core functional objective is to bridge two environments: on-chain and off-chain. We describe the architecture of each ChainLink component below. ChainLink will initially be built on Ethereum [16], [35], but we intend for it to support all leading smart contract networks for both off-chain and cross-chain interactions. In both its on and off-chain versions, ChainLink has been designed with modularity in mind. Every piece of the ChainLink system is upgradable, so that different components can be replaced as better techniques and competing implementations arise.

2.1 On-Chain Architecture

As an oracle service, ChainLink nodes return replies to data *requests* or *queries* made by or on behalf of a user contract, which we refer to as *requesting contracts* and denote by USER-SC. ChainLink's on-chain interface to requesting contracts is itself an on-chain contract that we denote by CHAINLINK-SC.

Behind CHAINLINK-SC, ChainLink has an on-chain component consisting of three main contracts: a *reputation contract*, an *order-matching contract*, and an *aggregating contract*. The reputation contract keeps track of oracle-service-provider performance metrics. The order-matching smart contract takes a proposed service level agreement, logs the SLA parameters, and collects bids from oracle providers. It then selects bids using the reputation contract and finalizes the oracle SLA. The aggregating contract collects the oracle providers' responses and calculates the final collective result of the ChainLink query. It also feeds oracle provider metrics back into the reputation contract. ChainLink contracts are designed in a modular manner, allowing for them to be configured or replaced by users as needed. The on-chain work flow has three steps: 1) oracle selection, 2) data reporting, 3) result aggregation.

Oracle Selection An oracle services purchaser specifies requirements that make up a service level agreement (SLA) proposal. The SLA proposal includes details such as query parameters and the number of oracles needed by the purchaser. Additionally, the purchaser specifies the reputation and aggregating contracts to be used for the rest of the agreement.

Using the reputation maintained on-chain, along with a more robust set of data gathered from logs of past contracts, purchasers can manually sort, filter, and select oracles via off-chain listing services. Our intention is for ChainLink to maintain one such listing service, collecting all ChainLink-related logs and verifying the binaries of listed oracle contracts. We further detail the listing service and reputation systems in Section 5. The data used to generate listings will be pulled from the blockchain, allowing for alternative oracle-listing services to be built. Purchasers will submit SLA proposals to oracles off-chain, and come to agreement before finalizing the SLA on-chain.

Manual matching is not possible for all situations. For example, a contract may need to request oracle services dynamically in response to its load. Automated solutions solve this problem and enhance usability. For these reasons, automated oracle matching is also being proposed by ChainLink through the use of order-matching contracts.

Once the purchaser has specified their SLA proposal, instead of contacting the oracles directly, they will submit the SLA to an order-matching contract. The submission of the proposal to the order-matching contract triggers a log that oracle providers can

monitor and filter based on their capabilities and service objectives. ChainLink nodes then choose whether to bid on the proposal or not, with the contract only accepting bids from nodes that meet the SLA's requirements. When an oracle service provider bids on a contract, they commit to it, specifically by attaching the penalty amount that would be lost due to their misbehavior, as defined in the SLA.

Bids are accepted for the entirety of the bidding window. Once the SLA has received enough qualified bids and the bidding window has ended, the requested number of oracles is selected from the pool of bids. Penalty payments that were offered during the bidding process are returned to oracles who were not selected, and a finalized SLA record is created. When the finalized SLA is recorded it triggers a log notifying the selected oracles. The oracles then perform the assignment detailed by the SLA.

Data Reporting Once the new oracle record has been created, the off-chain oracles execute the agreement and report back on-chain. For more detail about off-chain interactions, see Sections 2.2 and 4.

Result Aggregation Once the oracles have revealed their results to the oracle contract, their results will be fed to the aggregating contract. The aggregating contract tallies the collective results and calculates a weighted answer. The validity of each oracle response is then reported to the reputation contract. Finally, the weighted answer is returned to the specified contract function in USER-SC.

Detecting outlying or incorrect values is a problem that is specific to each type of data feed and application. For instance, detecting and rejecting outlying answers before averaging may be necessary for numeric data but not boolean. For this reason, there will not be a specific aggregating contract, but a configurable contract address which is specified by the purchaser. ChainLink will include a standard set of aggregating contracts, but customized contracts may also be specified, provided they conform to the standard calculation interface.

2.2 Off-Chain Architecture

Off-chain, ChainLink initially consists of a network of oracle nodes connected to the Ethereum network, and we intend for it to support all leading smart contract networks. These nodes independently harvest responses to off-chain requests. As we explain below, their individual responses are aggregated via one of several possible consensus mechanisms into a global response that is returned to a requesting contract USER-SC. The ChainLink nodes are powered by the standard open source core implementation which handles standard blockchain interactions, scheduling, and connecting with common external resources. Node operators may choose to add software

extensions, known as external adapters, that allow the operators to offer additional specialized off-chain services. ChainLink nodes have already been deployed alongside both public blockchains and private networks in enterprise settings; enabling the nodes to run in a decentralized manner is the motivation for the ChainLink network.

ChainLink Core. The core node software is responsible for interfacing with the blockchain, scheduling, and balancing work across its various external services. Work done by ChainLink nodes is formatted as *assignments*. Each assignment is a set of smaller job specifcations, known as subtasks, which are processed as a pipeline. Each subtask has a specific operation it performs, before passing its result onto the next subtask, and ultimately reaching a final result. ChainLink's node software comes with a few subtasks built in, including HTTP requests, JSON parsing, and conversion to various blockchain formats.

External Adapters. Beyond the built-in subtask types, custom subtasks can be defined by creating *adapters*. Adapters are external services with a minimal REST API. By modeling adapters in a service-oriented manner, programs in any programming language can be easily implemented simply by adding a small intermediate API in front of the program. Similarly, interacting with complicated multi-step APIs can be simplified to individual subtasks with parameters.

Subtask Schemas. We anticipate that many adapters will be open sourced, so that services can be audited and run by various community members. With many different types of adapters being developed by many different developers, ensuring compatibility between adapters is essential.

ChainLink currently operates with a schema system based on JSON Schema [36], to specify what inputs each adapter needs and how they should be formatted. Similarly, adapters specify an output schema to describe the format of each subtask's output.

3 Oracle Security

In order to explain ChainLink's security architecture, we must first explain why security is important—and what it means.

Why must oracles be secure? Returning to our simple examples in Section 1, if a smart contract security gets a false data feed, it may payout the incorrect party, if smart contract insurance data feeds can be tampered with by the insured party

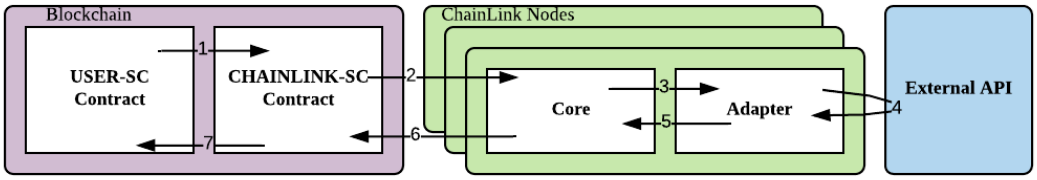

Figure 1: ChainLink workflow: **1)** USER-SC makes an on-chain request; **2)** CHAINLINK-SC logs an event for the oracles; **3)** ChainLink core picks up the event and routes the assignment to an adapter; **4)** ChainLink adapter performs a request to an external API; **5)** ChainLink adapter processes the response and passes it back to the core; **6)** ChainLink core reports the data to CHAINLINK-SC; **7)** CHAINLINK-SC aggregates responses and passes them back as a single response to USER-SC.

there may be insurance fraud, and if GPS data given to a trade finance contract can be modified after it leaves the data provider, payment can be released for goods that haven't arrived.

More generally, a well-functioning blockchain, with its ledger or bulletin-board abstraction, offers very strong security properties. Users rely on the blockchain as a functionality that correctly validates transactions and prevents data from being altered. They treat it in effect like a trusted third party (a concept we discuss at length below). A supporting oracle service must offer a level of security commensurate with that of the blockchain it supports. An oracle too must therefore serve users as an effective trusted third party, providing correct and timely responses with very high probability. The security of any system is only as strong as its weakest link, so a highly trustworthy oracle is required to preserve the trustworthiness of a well-engineered blockchain.

Defining oracle security: An ideal view. In order to reason about oracle security, we must first define it. An instructive, principled way to reason about oracle security stems from the following thought experiment. Imagine that a trusted third party (TTP)—an ideal entity or functionality that always carries out instructions faithfully to the letter—were tasked with running an oracle. We'll denote this oracle by ORACLE (using all caps in general to denote an entity fully trusted by users), and suppose that the TTP obtains data from a perfectly trustworthy data source Src. Given this magical service ORACLE, what instructions would we ask it to carry out?

To achieve the property of integrity, also referred to as the authenticity property [24], we would simply ask that ORACLE perform the following steps:

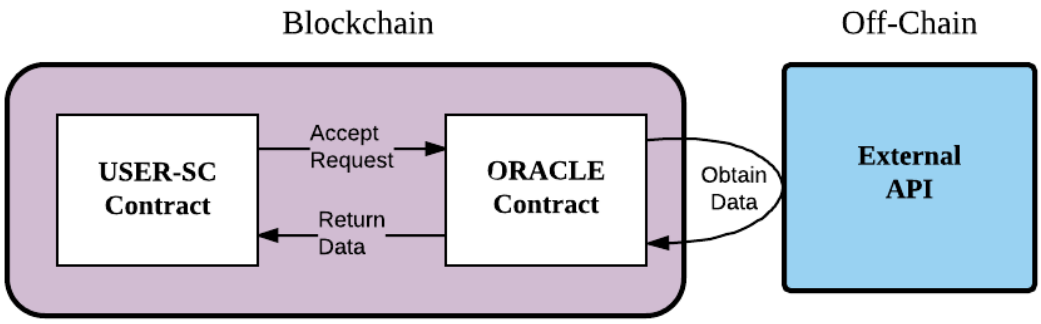

Figure 2: Behavior of an ideal oracle ORACLE is defined by steps: **1)** Accept request; **2)** Obtain data; **3)** Return data. Additionally, to protect the confidentiality of a request, upon decrypting it, ORACLE never uses or reveals the data it contains, except to query Src.

1. *Accept request*: Ingest from a smart contract USER-SC a request Req $= (\mathsf{Src}, \tau, q)$ that specifies a target data source Src, a time or range of times τ, and a query q;
2. *Obtain data*: Send query q to Src at time τ;
3. *Return data*: On receiving answer a, return a to the smart contract.

These simple instructions, correctly carried out, define a strong, meaningful, but simple notion of security. Intuitively, they dictate that ORACLE acts as a trustworthy bridge between Src and USER-SC.[2] For example, if Src is `https://www.FountOfKnowledge.com`, τ is 4 p.m., and $q =$ "`price for ticker INTC`", the integrity of ORACLE guarantees that it will provide USER-SC with exactly the price of INTC as queried at 4 p.m. at https://www.FountOfKnowledge.com.

Confidentiality is another desirable property for oracles. As USER-SC sends Req to ORACLE in the clear on the blockchain, Req is *public*. There are many situations in which Req is sensitive and its publication could be harmful. If USER-SC is a flight insurance contract, for example, and sends ORACLE a query Req regarding a particular user's flight ($q =$ "`Ether Air Flight 338`"), the result would be that a user's flight plans are revealed to the whole world. If USER-SC is a contract for

[2]Of course, many details are omitted here. ORACLE should communicate with both USER-SC and source Src over secure, i.e., tamperproof, channels. (If Src is a web server, TLS is required. To communicate with USER-SC, ORACLE must be sure to scrape the right blockchain and digitally sign A appropriately.)

financial trading, Req could leak information about a user's trades and portfolio. There are many other examples, of course.

To protect the confidentiality of Req, we can require that data in Req be encrypted under a (public key) belonging to ORACLE. Continuing to leverage the TTP nature of ORACLE, we could then simply give ORACLE the information-flow constraint:

Upon decrypting Req, never reveal or use data in Req except to query Src.

There are other important oracle properties, such as availability, the last of the classical CIA (Confidentiality-Integrity-Availability) triad. A truly ideal service ORACLE, of course, would never go down. Availability also encompasses more subtle properties such as censorship resistance: An honest ORACLE will not single out particular smart contracts and deny their requests.

The concept of a trusted third party is similar to the notion of an ideal functionality [7] used to prove the security of cryptographic protocols in certain models. We can also model a blockchain in similar terms, conceptualizing it in terms of a TTP that maintains an ideal bulletin board. Its instructions are to accept transactions, validate them, serialize them, and maintain them permanently on the bulletin board, an append-only data structure.

Why the ideal oracle (ORACLE) is hard to achieve. There is, of course, no perfectly trustworthy data source Src. Data may be benignly or maliciously corrupted due to faulty web sites, cheating service providers, or honest mistakes.

If Src isn't trustworthy, then even if ORACLE does operate exactly like a TTP as instructed above, it still doesn't completely meet the notion of security we want. Given a faulty source Src, the integrity property defined above no longer means that an oracle's answer a is correct. If the true price of Intel is \$40 and `https://www.FountOfKnowledge.com` misreports it as \$50, for example, then ORACLE will send the incorrect value $a = \$50$ to USER-SC. This problem is unavoidable when using a single source Src. ORACLE simply has no way to know whether the answers Src provides to its queries are correct.

A bigger issue, of course, is the fact that our TTP for ORACLE is just an abstraction. No service provider is unconditionally trustworthy. Even the best-intentioned may be buggy or hacked. So there is no way to for a user or smart contract to have absolute assurance that a service ORACLE will carry out its instructions faithfully.

ChainLink reasons about its security protocols in terms of this ideal functionality ORACLE. Our goal in ChainLink is to achieve a real world system with *properties as close as possible to those of ORACLE* under realistic trust assumptions. We now explain how.

For simplicity in what follows, we now denote by CHAINLINK-SC the complete set of ChainLink contracts, i.e., its full on-chain functionality (not just its interface

to requesting contracts). We thereby abstract away the multiple individual contracts actually used in the system architecture.

4 ChainLink Decentralization Approach

We propose three basic complementary approaches to ensuring against faulty nodes: (1) Distribution of data sources; (2) Distribution of oracles; and (3) Use of trusted hardware. We discuss the first two approaches, which involve *decentralization*, in this section. We discuss our long-term strategy for trusted hardware, a different and complementary approach, in Section 6.

4.1 Distributing sources

A simple way to deal with a faulty single source Src is to obtain data from *multiple* sources, i.e., distribute the data source. A trustworthy ORACLE can query a collection of sources $\mathsf{Src}_1, \mathsf{Src}_2, \ldots, \mathsf{Src}_k$, obtain responses $a_1, a_2, \ldots, a_k$, and aggregate them into a single answer $A = \mathtt{agg}(a_1, a_2, \ldots, a_k)$. ORACLE might do this in any of a number of ways. One, for example, is majority voting. If a majority of sources return the identical value a, the function agg returns a; otherwise it returns an error. In this case, provided that a majority ($> k/2$) sources are functioning correctly, ORACLE will always return a correct value A.

Many alternative functions agg can ensure robustness against erroneous data or handle fluctuations in data values over time (e.g, stock prices). For example, agg might discard outliers (e.g., the largest and smallest values a_i) and output the mean of the remaining ones.

Of course, faults may be correlated across data sources in a way that weakens the assurances provided by aggregation. If site Src_1 = `EchoEcho.com` obtains its data from Src_2 = `TheHorsesMouth.com`, an error at Src_2 will always imply an error at Src_1. More subtle correlations between data sources can also occur. Chainlink also proposes to pursue research into mapping and reporting the independence of data sources in an easily digestible way so that oracles and users can avoid undesired correlations.

4.2 Distributing oracles

Just as sources can be distributed, our ideal service ORACLE itself can be approximated as a distributed system. This is to say that instead of a single monolithic oracle node O, we can instead have a collection of n different oracle nodes $\{O_1, O_2, \ldots, O_n\}$. Each oracle O_i contacts its own distinct set of data sources which may or may not

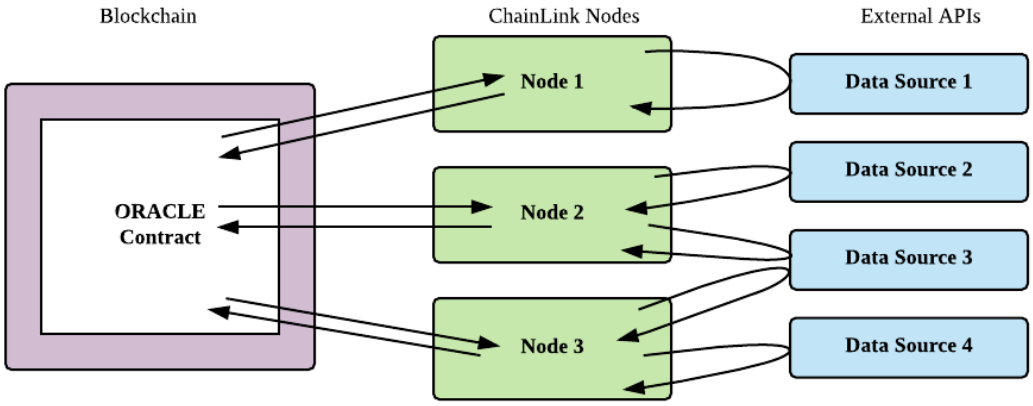

Figure 3: Requests are distributed across both oracles and data sources. This figure shows an example of such two-level distribution.

overlap with those of other oracles. O_i aggregates responses from its data sources and outputs its own distinct answer A_i to a query Req.

Some of these oracles may be faulty. So clearly the set of all oracles' answers $A_1, A_2, \ldots, A_n$ will need to be aggregated in a trustworthy way into a single, authoritative value A. But given the possibility of faulty oracles, where and how will this aggregation happen in ChainLink?

Initial solution: In-contract aggregation. Our initial proposed solution in ChainLink will be a simple one called *in-contract aggregation*. CHAINLINK-SC—which, again, denotes the on-chain part of ChainLink—will itself aggregate oracle responses. (Alternatively, CHAINLINK-SC may call another aggregation contract, but for conceptual simplicity we assume that the two components form a single contract.) In other words, CHAINLINK-SC will compute $A = \mathsf{Agg}(A_1, A_2, \ldots, A_n)$ for some function Agg (similar to agg, as described above), and send the result A to USER-SC.

This approach is practical for small n, and has several distinct benefits:

- *Conceptual simplicity:* Despite the fact that the oracle is distributed, a single entity, CHAINLINK-SC, performs aggregation by executing Agg.
- *Trustworthiness:* As CHAINLINK-SC's code can be publicly inspected, its correct behavior can be verified. (CHAINLINK-SC will be a relatively small, simple piece of code.) Additionally, CHAINLINK-SC's execution is fully visible on-

chain. Thus users, i.e., creators of USER-SC, can achieve a high degree of trust in CHAINLINK-SC.

- *Flexibility:* CHAINLINK-SC can implement most desired aggregation functions Agg—the majority function, averaging, etc.

Simple as it is, this approach presents a novel and interesting technical challenge, namely the problem of *freeloading*. A cheating oracle O_z can observe the response A_i of another oracle O_i and copy it. In this way, oracle O_z avoids the expense of querying data sources, which may charge per-query fees. Freeloading weakens security by undermining the diversity of data source queries and also disincentivizes oracles from responding quickly: Responding slowly and freeloading is a cheaper strategy.

We suggest a well known solution to this problem, namely the use of a commit / reveal scheme. In a first round, oracles send CHAINLINK-SC cryptographic commitments to their responses. After CHAINLINK-SC has received a quorum of responses, it initiates a second round in which oracles reveal their responses.

Algorithm 1 shows a simple sequential protocol that guarantees availability given $3f + 1$ nodes. It uses a commit / reveal scheme to prevent freeloading. Oracle responses are decommitted, and thus exposed to a potential freeloader only *after* all commitments have been made, thereby excluding the freeloader from copying other oracles' responses.

On-chain protocols can leverage block times to support synchronous protocol designs. In ChainLink, however, oracle nodes obtain data from sources that may have highly variable response times, and decommitment times by nodes can vary due to, e.g., use of different gas prices in Ethereum. To ensure the fastest possible protocol responsiveness, therefore, Alg. 1 is designed as an asynchronous protocol.

Here, $\mathsf{Commit}_r(A)$ denotes a commitment of value A with witness r, while SID denotes the set of valid session ids. The protocol assumes authenticated channels among all players.

It is easy to see that Alg. 1 will terminate successfully. Given $3f + 1$ nodes in total, at most f are faulty, so at least $2f + 1$ will send commitments in Step 4. Of those commitments, at most f come from faulty nodes, so at least $f + 1$ come from honest nodes. All such commitments will eventually be decommitted.

Additionally, it is easy to see that A will be correct in Alg.1. Of the $f + 1$ decommitments on the single value A, at least one has to come from an honest node.

In-contract aggregation via Alg. 1 will be the main approach supported by ChainLink in the short term. The proposed initial implementation will involve a more sophisticated, concurrent variant of the algorithm. Our longer-term proposal is reflected in the rather more complicated protocol OCA (Off-Chain Aggregation) specified in Algorithms 2 and 3 in Appendix A. OCA is an off-chain aggregation protocol that

Algorithm 1 InChainAgg($\{O_i\}_{i=1}^n$) (code for CHAINLINK-SC)

1: Wait until Req is received from USER-SC.
2: $\mathsf{sid} \leftarrow_\$ \mathsf{SID}$
3: Broadcast $(\texttt{request}, \mathsf{sid})$.
4: Wait until set C of $2f+1$ messages $(\texttt{commit}, c_i = \mathsf{Commit}_{r_i}(A_i), \mathsf{sid})$ from distinct O_i are received.
5: Broadcast $(\texttt{committed}, \mathsf{sid})$.
6: Wait until set D of $f+1$ distinct valid decommitments $(\texttt{decommit}, (r_i, A_i), \mathsf{sid})$ are received where, for some A, all $A_i = A$.
7: Send $(\texttt{Answer}, A, \mathsf{sid})$ to USER-SC.

minimizes on-chain transaction costs. That protocol also includes payment to oracle nodes and ensures against payments to freeloaders.

Medium-term strategy: Off-chain aggregation. In-contract aggregation has a key disadvantage: Cost. It incurs the cost of transmitting and processing on-chain $O(n)$ oracle messages (commits and reveals for $A_1, A_2, \ldots, A_n$). In permissioned blockchains, this overhead may be acceptable. In permissionless blockchains with on-chain transaction fees such as Ethereum, if n is large, the costs can be prohibitive. A more cost-effective approach is to aggregate oracle responses off-chain and transmit a single message to CHAINLINK-SC A. We propose deployment of this approach, called *off-chain aggregation*, in the medium-to-long term.

The problem of achieving a consensus value A in the face of potentially faulty nodes is much like the problem of consensus that underpins blockchains themselves. Given a predetermined set of oracles, one might consider using a classical Byzantine Fault Tolerant (BFT) consensus algorithm to compute A. Classical BFT protocols, however, aim to ensure that at the end of a protocol invocation, all honest nodes store the same value, e.g., in a blockchain, that all nodes store the same fresh block. In our oracle setting, the goal is slightly different. We want to ensure that CHAINLINK-SC (and then USER-SC) obtains aggregate answer $A = \mathsf{Agg}(A_1, A_2, \ldots, A_n)$ *without participating in the consensus protocol and without needing to receive answers from multiple oracles*. The problem of freeloading, moreover, still needs to be addressed.

The ChainLink system proposes the use of a simple protocol involving threshold signatures. Such signatures can be realized using any of a number of signature schemes, but are especially simple to implement using Schnorr signatures [4]. In this approach, oracles have a collective public key pk and a corresponding private key sk that is shared among $O_1, O_2, \ldots, O_n$ in a (t, n)-threshold manner [3]. Such a sharing means that every node O_i has a distinct private / public keypair $(\mathsf{sk}_i, \mathsf{pk}_i)$. O_i can

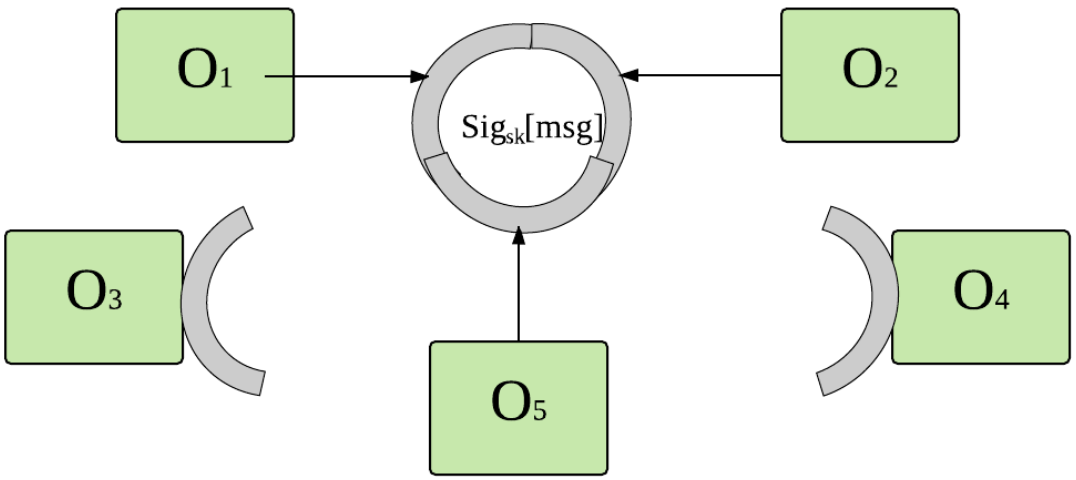

Figure 4: $\mathsf{Sig}_{\mathsf{sk}}[A]$ can be achieved by any n/2+1 of the oracles.

generate a partial signature $\sigma_i = \mathsf{Sig}_{\mathsf{sk}_i}[A_i]$ that can be verified with respect to pk_i.

The key feature of this setup is that partial signatures on the same value A can be aggregated across any set of t oracles to yield a single valid *collective* signature $\Sigma = \mathsf{Sig}_{\mathsf{sk}}[A]$ on an answer A. No set of $t-1$ oracles, however, can produce a valid signature on any value. The single signature Σ thus implicitly embodies the partial signatures of at least t oracles.

Threshold signatures can be realized naïvely by letting Σ consist explicitly of a set of t valid, independent signatures from individual nodes. Threshold signatures have similar security properties to this naïve approach. But they provide a significant on-chain performance improvement: They reduce the size and cost of verifying Σ by a factor of t.

With this setup, it would seem that oracles can just generate and broadcast partial signatures until t such partial signatures enable the computation of Σ. Again, though, the problem of freeloading arises. We must therefore ensure that oracles genuinely obtain data from their designated sources, rather than cheating and copying A_i from another oracle. Our solution involves a financial mechanism: An entity PROVIDER (realizable as a smart contract) rewards only oracles that have sourced original data for their partial signatures.

In a distributed setting, determining which oracles qualify for payment turns out to be tricky. Oracles may intercommunicate off-chain and we no longer have a single authoritative entity (CHAINLINK-SC) receiving responses and are therefore un-

able to identify eligible payees directly among participating oracles. Consequently, PROVIDER must obtain evidence of misbehavior from the oracles themselves, some of which may be untrustworthy. We propose the use of consensus-like mechanisms in our solution for ChainLink to ensure that PROVIDER does not pay freeloading oracles.

The off-chain aggregation system we propose for ChainLink, with accompanying security proof sketches, may be found in Appendix A. It makes use of a distributed protocol based on threshold signatures that provides resistance to freeloading by $f < n/3$ oracles. We believe resistance to freeloading is an interesting new technical problem.

5 ChainLink Security Services

Thanks to the protocols we have just described in the previous section, ChainLink proposes to ensure availability and correctness in the face of up to f faulty oracles. Additionally, trusted hardware, as discussed in Section 6, is being actively considered as a secure approach toward protecting against corrupted oracles providing incorrect responses. Trusted hardware, however, may not provide definitive protection for three reasons. First, it will not be deployed in initial versions of the ChainLink network. Second, some users may not trust trusted hardware (see Appendix B for a discussion). Finally, trusted hardware cannot protect against node downtime, only against node misbehavior. Users will therefore wish to to ensure that they can choose the most reliable oracles and minimize the probability of USER-SC relying on $> f$ faulty oracles.

To this end, we propose the use of four key *security services*: a *Validation System*, a *Reputation System*, a *Certification Service*, and a *Contract-Upgrade Service*. All of these services may initially be run by one company or group interested in launching the ChainLink network, but are designed to *operate strictly accordingly to ChainLink's philosophy of decentralized design*. ChainLink's proposed security services cannot block oracle node participation or alter oracle responses. The first three services only provide ratings or guidance to users, while the Contract-Upgrade Service is entirely optional for users. Additionally, these services are designed to support independent providers, whose participation should be encouraged so that users will eventually have multiple security services among which to choose.

5.1 Validation System

The ChainLink Validation System monitors on-chain oracle behavior, providing an objective performance metric that can guide user selection of oracles. It will seek to monitor oracles for:

- *Availability:* The Validation System should record failures by an oracle to respond in a timely way to queries. It will compile ongoing uptime statistics.

- *Correctness:* The Validation System should record apparent erroneous responses by an oracle as measured by deviations from responses provided by peers.[3]

In our initial, on-chain aggregation system in ChainLink, such monitoring is straightforward, as all oracle activity is visible to CHAINLINK-SC.

Recall, however, that in the off-chain aggregation system envisaged for ChainLink, it's the oracles themselves that perform aggregation. Consequently, CHAINLINK-SC does not have direct visibility into oracle responses and cannot itself monitor availability and correctness.

Fortunately, oracles digitally sign their responses, and thus, as a side effect, generate non-repudiable evidence of their answers. Our proposed approach will therefore be to realize the validation service as a smart contract that would reward oracles for submitting evidence of deviating responses. In other words, oracles would be incentivized to report apparently erroneous behavior.

Availability is somewhat trickier to monitor, as oracles of course don't sign their failures to respond. Instead, a proposed protocol enhancement would require oracles to digitally sign attestations to the set of responses they have received from other oracles. The validation contract would then accept (and again reward) submission of sets of attestations that demonstrate consistent non-responsiveness by an underperforming oracle to its peers.

In both the on-chain and off-chain cases, availability and correctness statistics for oracles will be visible *on-chain*. Users / developers will thus be able to view them in real time through an appropriate front end, such as a Dapp in Ethereum or an equivalent application for a permissioned blockchain.

5.2 Reputation System

The Reputation System proposed for ChainLink would record and publish user ratings of oracle providers and nodes, offering a means for users to evaluate oracle performance holistically. Validator System reports are likely to be a major factor in determining oracle reputations and placing these reputations on a firm footing of trust. Factors beyond on-chain history, though, can provide essential information about oracle node

[3]"Deviation" must be defined in a data-specific manner. For simple boolean responses—for example, whether a flight arrived on time—deviation simply means a response opposite that of the majority. For, say, the temperature of a city, which may vary legitimately across sensors and sources, deviation may mean significant numerical deviation. Of course, for various reasons, e.g., broken sensors, even a well-functioning oracle may deviate from the majority answer some fraction of the time.

security profiles. These may include users' familiarity with oracles' brands, operating entities, and architectures. We envision the ChainLink Reputation System to include a basic on-chain component where users' ratings would be available for other smart contracts to reference. Additionally, reputation metrics should be easily accessible off-chain where larger amounts of data can be efficiently processed and more flexibly weighted.

For a given oracle operator, the Reputation System is initially proposed as supporting the following metrics, both at the granularity of specific assignment types (see Section 2), and also in general for all types supported by a node:

- *Total number of assigned requests*: The total number of past requests that an oracle has agreed to, both fulfilled and unfulfilled.
- *Total number of completed requests*: The total number of past requests that an oracle has fulfilled. This can be averaged over number of requests assigned to calculate completion rate.
- *Total number of accepted requests*: The total number of requests that have been deemed acceptable by calculating contracts when compared with peer responses. This can be averaged over total assigned or total completed requests to get insight into accuracy rates.
- *Average time to respond*: While it may be necessary to give oracle responses time for confirmation, the timeliness of their responses will be helpful in determining future timeliness. Average response time is calculated based on completed requests.
- *Amount of penalty payments*: If penalty payments were locked in to assure a node operator's performance, the result would be a financial metric of an oracle provider's commitment not to engage in an "exit scam" attack, where the provider takes users' money and doesn't provide services. This metric would involve both a temporal and a financial dimension.

High-reputation services are strongly incentivized in any market to behave correctly and ensure high availability and performance. Negative user feedback will pose a significant risk to brand value, as do the penalties associated with misbehavior. Consequently, we anticipate a virtuous circle in which well-functioning oracles develop good reputations and good reputations give rise to incentives for continued high performance.

5.3 Certification Service

While our Validation and Reputation Systems are intended to address a broad range of faulty behaviors by oracles and is proposed as a way to ensure system integrity in the vast majority of cases, ChainLink may also include an additional mechanism called a *Certification Service*. Its goal is to prevent and/or remediate rare but catastrophic events, specifically *en bloc* cheating in the form of Sybil and mirroring attacks, which we now explain.

Sybil and mirroring attacks. Both our simple and in-contract aggregation protocols seek to prevent freeloading in the sense of dishonest nodes copying honest nodes' answers. But neither protects against *Sybil attacks* [9]. Such attacks involve an adversary that controls multiple, ostensibly independent oracles. This adversary can attempt to dominate the oracle pool, causing more than f oracles to participate in the aggregation protocol and provide *false data* at strategic times, e.g., in order to influence large transactions in high-value contracts. Quorums of cheating oracles can also arise not just under the control of a single adversary, but also through collusion among multiple adversaries. Attacks or faults involving $> f$ oracles are especially pernicious in that they are undetectable from on-chain behavior alone.

Additionally, to reduce operational costs, a Sybil attacker can adopt a behavior called *mirroring*, in which it causes oracles to send individual responses based on data obtained from a *single data-source query*. In other words, misbehaving oracles may share data off-chain but pretend to source data independently. Mirroring benefits an adversary whether or not it chooses to send false data. It poses a much less serious security threat than data falsification, but does slightly degrade security in that it eliminates the error correction resulting from diversified queries against a given source Src. For example, if `https://www.datasource.com` emits erroneous data due to, say, a sporadically triggered bug, multiple queriers may still obtain a correct majority result.

Sybil attacks resulting in false data, mirroring, and collusion in general may be eliminated by the use of trusted hardware in our long-term strategy (see Section 6).

Certification Service design. The ChainLink Certification Service would seek to provide general integrity and availability assurance, detecting and helping prevent mirroring and colluding oracle quorums in the short-to-medium term. The Certification Service would issue *endorsements* of high-quality oracle providers. We emphasize again, as noted above, that the service will only rate providers for the benefit of users. It is not meant to dictate oracle node participation or non-participation in the system.

The Certification Service supports endorsements based on several features of oracle deployment and behavior. It would monitor the Validation System statistics

on oracles and perform post-hoc spot-checking of on-chain answers—particularly for high-value transactions—comparing them with answers obtained directly from reputable data sources. With sufficient demand for an oracle provider's data, we expect there to be enough economic incentive to justify off-chain audits of oracle providers, confirming compliance with relevant security standards, such as relevant controls in the Cloud Security Alliance (CSA) Cloud Controls Matrix [26], as well as providing useful security information that they conduct proper audits of oracles' source and bytecode for their smart contracts.

In addition to the reputation metrics, automated on-chain and automated off-chain systems for fraud detection, the Certification Service is planned as a means to identify Sybil attacks and other malfeasance that automated on-chain systems cannot. For example, if all nodes agree that the moon is made of green cheese, they can cause USER-SC to ingest this false fact. `MOON COMPONENTS = {GREEN CHEESE}` will be recorded on the blockchain, however, and visible in a post-hoc review.

5.4 Contract-Upgrade Service

As recent smart contract hacks have shown, coding bulletproof smart contracts is an extremely challenging exercise [1], [20], [22]. And even if a smart contract has been correctly programmed, environmental changes or bugs can still result in vulnerabilities, e.g., [2].

For this reason, we propose a Contract-Upgrade Service. We emphasize that use of this service is entirely optional and in control of users.

In the short term, if vulnerabilities are discovered, the Contract-Upgrade Service would simply make a new set of supporting oracle contracts available in ChainLink. Newly created requesting smart contracts will then be able to migrate to the new set of oracle contracts.

Unfortunately, though, existing ones would be stuck with the old, potentially vulnerable set. In the longer term, therefore, CHAINLINK-SC would support a flag (`MIGFLAG`) in oracle calls from requesting contracts indicating whether or not a call should be forwarded to a new CHAINLINK-SC should one become available. Set by default (i.e., if the flag is missing) to false, `MIGFLAG` would enable requesting contracts to benefit from automatic forwarding and thus migration to the new version of CHAINLINK-SC. In order to activate forwarding, a user will cause her requesting contract to issue ChainLink requests with `MIGFLAG` = true. (Users can engineer their smart contracts so that they change this flag upon receiving an instruction to do so on-chain from an authorized contract administrator.)

Migration of users to new oracle contracts functions as a kind of "escape hatch," something long advocated for by blockchain researchers (see, e.g., [23]) as a mechanism to fix bugs and remediate hacks without resorting to such cumbersome approaches as