Essays on Smart Contracts and Oracles

by
Mudabbir Kaleem

A thesis submitted to the Department of Computer Science,

College of Natural Sciences and Mathematics

in partial fulfillment of the requirements for the degree of

Master of Science
in Computer Science

Chair of Committee: Weidong (Larry) Shi

Committee Member: Aron Laszka

Committee Member: Lei Xu

University of Houston
December 2021

Copyright 2021, Mudabbir Kaleem

ACKNOWLEDGMENTS

I wish to express my gratitude to my advisor, Dr. Weidong (Larry) Shi for his mentorship and guidance over the course of my graduate program. His enthusiasm, motivational advice and perhaps, above all, his kindness have been inspirational. I am indebted to him for all his support.

I also wish to acknowledge all my collaborators and peers here at the University of Houston with whom I got a chance to work with and learn from. I am also thankful to Dr. Aron Laszka, for initiating me in the research domain of smart contracts and for his help in my admittance to the program.

My deepest gratitude also goes to my beloved mother and father, Zahida and Kaleem, who have always been the bedrock of my stride. And to my teachers from school, who bared with the inquisitive and mischievous child.

To Hassan, Arsalan, Joseph, Dean, Jennifer and Grace, who made times memorable and kept my spirits high. And to my uncle Zafar, aunt Arusa and Fakhar for assuring me that home was always close by.

Finally, a token of my humble gratitude to the great men and women who built this institution, state and country, for I have enjoyed the shade of trees which they planted. And lastly, to the people of Texas, for supporting the institution, the research and graduate students like myself, both through their fund as well as their hospitable spirit.

PREVIOUSLY PUBLISHED MATERIAL

Part 1 revises a previous publication [26]:

An Event Driven Framework for Smart Contract Execution

M. Kaleem, K. Kasichainula, R. Karanjai, L. Xu, Z. Gao, L. Chen, W. Shi.

Proc. ACM DEBS, 2021.

Part 2 revises a previous publication [28]:

Demystifying Pythia: A Survey of ChainLink Oracles Usage on Ethereum

M. Kaleem, W. Shi.

Proc. FC DeFi, 2021.

ABSTRACT

The thesis is composed of two essays on smart contract platforms presented as the two sections of this document. The essays represent the major chunk of my core research work during the course of my Masters program. My research journey started with learning about Bitcoin, blockchain and smart contract programming back in the fall of 2019 and spring of 2020. After studying and publishing work related to smart contract programming vulnerabilities in Vyper and Solidity [27], I moved on to studying more about blockchains at L1 as well as about semantic web and event-driven programming models in the summer of 2020. This inspired the first essay of the thesis which explores the design and implementation of a smart contract platform that is designed on the event-driven execution model and compares it to the traditional smart contract platforms that employ the transaction-driven execution model. The resulting work was published in the fall of 2020 and is presented here as the first essay i.e. Part I.

During the design and testing phase of our event-driven smart contract platform, we identified the deployment and adaption of oracles as one of the biggest use-cases to benefit from our proposed design. Naturally, this lead to a comprehensive study of oracle usage on the Ethereum mainnet to identify and quantitatively predict the improvements that our proposed design might bring to the usage of smart contract oracles. This study is presented as the second essay i.e. Part II.

Although, both the studies can be viewed and read independently, the second study was done as a supplemental project to our first project and should be viewed as such.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Part I

AN EVENT-DRIVEN SMART CONTRACT PLATFORM DESIGN

1 Abstract

Blockchain-based smart contract platforms have traditionally employed the transaction-driven execution model. This chapter presents an alternate framework for blockchain-based smart contract execution called EDSC. Our platform design presents a novel approach to tackle the scalability and performance challenges facing the smart contract ecosystem. We base EDSC's design on the Ethereum template, and it can be readily implemented for other existing smart contract platforms. To evaluate our design, we perform an experimental implementation using the Ethereum client. Our experiments with performance modeling show, on average, a 2.2 to 4.6 times reduced total latency of event-triggered smart contracts, demonstrating the effectiveness of the design in supporting time-sensitive applications. Additionally, we comment on the design's potential security aspects and demonstrate its utility by discussing potential use cases.

2 Introduction

The advent of Bitcoin, in late 2008, demonstrated how a digital payment system could be implemented using a novel decentralized public data structure, i.e., a blockchain [40]. Additionally, Bitcoin also incorporated a built-in scripting framework that could be used for controlling the tokens, storing data, and specifying logic on the blockchain itself. This was a decentralized implementation of smart contracts [45]. Smart contracts are essentially pieces of code that enforce the terms and procedures of an agreement or protocol digitally. However, Bitcoin's smart contract functionality was limited in its application, and it was not until the introduction of Ethereum [10] that smart contracts took center stage in the cryptocurrency arena. Ethereum offered an integral Turing-complete programming language to create blockchain-based smart contracts, allowing for their employment in a wide range of potential use cases [29].

Presently, smart contracts continue to grow in their utility and outreach. Since the launch of Ethereum, many alternative smart contract platforms have also emerged, which have gained considerable adaption and sizeable user bases [21] [7] [31] [30] [37]. The majority of these platforms aim to overcome or readdress Ethereum's limitations and trade-offs, e.g., achieving higher throughput, decreasing computation costs, deploying a different consensus mechanism, etc. Although initially limited to token control and on-chain data access, smart contracts today are increasingly interfacing with real-world data and events, rapidly extending their application sphere. This interaction is enabled through oracles [4], which are services designed to provide external information in the smart contract environment. In order to avoid a single point of compromise for such integration, many recent oracle projects [20] [43] [42] are adapting a decentralized approach for collecting and aggregating data.

Despite numerous innovations and advancements, the smart contract ecosystem's evolution has been stifled by various impediments, mostly prevailing in transaction performance (e.g., latency, throughput) and scalability domains. Although several projects [3] [46] [48] have sought to address these concerns through imaginative solutions like sharding and execution parallelization; many

complex design challenges remain unsolved for practical purposes. With the recent unprecedented growth of decentralized financial services (DeFi), an ever-increasing percentage of smart contracts are interfacing with oracle networks to fetch real-world information [34] [28]. Since this interfacing is accomplished through two-way transactions on most platforms, the trend is bound to further burden an already congested system [38].

In the smart contract space, most existing popular platforms conform to the *transaction-driven execution model*. This means that all contract executions on these platforms are triggered by initiating transactions on the system through non-contract accounts. In this chapter, we present an alternative smart contract platform design built on the event-driven execution model. We call our design EDSC (an Event Driven Smart Contract platform). The event-driven architecture pattern is a simple yet powerful distributed architecture pattern, proven to produce highly scalable and adaptable applications. The model enables communication by allowing participants to publish notifications of occurring events, along with subscribing to events of interest and being asynchronously notified of their occurrence by the system. The event-based methodology has previously been extensively studied in the context of systems and software engineering [22] [41] [44]. We reason that a smart contract platform framework centered around the publish/subscribe paradigm will be a good fit for many emerging smart contract applications that demand or can benefit from timely execution. We also demonstrate that it will be, by design, better positioned to address the aforementioned issues hampering the ecosystem's progression.

This chapter intends to describe an event-driven smart contract platform's architectural layout and implementation. We use the Ethereum architecture as the base template and outline the modifications required in its design to actualize our system. The rationale for this approach is that we assume most readers to be acquainted with Ethereum's mechanics, given that it is the pioneering and most widely used smart contract platform to date. This familiarity, we hope, will allow the readers to draw parallels between the two models while enabling us to communicate our design succinctly. It is worth mentioning that although presented using Ethereum as the reference, the concept of event-driven smart contracts illustrated in this work can be extended to other smart

contract platforms, consensus protocols, and programming models with trivial adjustments. The proposed design also has numerous advantages over the prior art that attempted to support events in the application layer. To the best of our knowledge, the proposed system is the first smart contract platform designed upon the event-driven execution model. We hope that our work will inspire further research in the direction of applying the event-driven communication paradigm to blockchains.

To summarize, the main contributions of the work and this chapter are:

- *We propose an event-driven smart contract platform with native support for real-time event processing.*
- *We provide the design of an event-based system using Ethereum as a reference target.*
- *We describe the design's advantages in potential use cases and comment on its security aspects.*
- *We have performed an implementation using the Golang Ethereum client and conducted experiments where performance modeling results show on average a 2.2 to 4.6 times reduction in total latency of event triggered smart contracts, which demonstrates its effectiveness for supporting contracts that demand timely execution based on events.*

The remainder of the chapter is organized as follows: We begin by giving an overview of related work in Section 3. Then in Section 4, we provide the basic description of an event-based platform and its desired functionality. In light of the functionality description, we comment on the benefits and limitations of such a system in Section 5. Section 6 presents our proposed system design and details the alterations it makes to a transaction-driven framework. Section 7 provides commentary on the design's security aspects. We present our experimental implementation in Section 8 and discuss the results and findings in Section 9. Section 10 discusses potential use cases and examples to explain the design. Finally, we present our concluding remarks in Section 11.

3 Related Work

In recent years, the advancement of blockchains, smart contracts, and decentralized infrastructures have created an emerging frontier that combines traditional concepts of event-driven systems with blockchains. As such, there have been ongoing efforts to harness the benefits of this paradigm in blockchain-based smart contracts. Oracles and subscription-based payment models have sought to achieve this but have encountered limitations. Currently, oracle systems typically follow a pull-based model with the client contract requesting data from an off-chain source. Present designs favor off-chain implementations to incorporate the event-based subscription model and then interact with the blockchain.

We take as a case study the implementation of the IBM blockchain, which is built on Hyperledger [11]. Prior work by Hull [24] shows how event-based processing is used for data-centric applications. The commercial implementation and offering by IBM [25] uses Java micro-services to listen for events from the blockchain using OpenLiberty. Blockchain provides the integrity of the process, whereas the java micro-service layer and OpenLiberty ensure it can have event-based transactions. However, apart from the implementation layer, it does not use the smart-contracts for any event-based transactions. Another commercial offering is provided by Amazon [9], which uses the Hyperledger Fabric and Ethereum as the underlying layer. Their implementation allows three distinct kinds of events to interact with the blockchain network, namely: (i) Block event, which occurs when a new block is added to the ledger; (ii) Transaction event and; (iii) Chaincode event, which can hold conditions for triggering events. The triggering mechanism for these events relies on AWS Fargate to act as an event listener and then on Amazon Simple Queue Services to be processed by lambdas. Just like IBM, Amazon's implementation also relies on a layer of auxiliary services to enable event-driven architecture.

Recent works remedying this limitation include EventWarden [33], where the authors propose a decentralized event-driven proxy that can interact with Ethereum-like blockchain networks and pass the transactions. This approach eliminates the use of auxiliary services similar to what IBM

and Amazon were employing since this can be implemented directly onto the Ethereum network. It allows a user to create a proxy smart contract describing an event into the contract. Anyone in the blockchain network can trigger a release of the reserved transaction by calling the proxy contract and showing that the concerned event has been recorded into blockchain logs.

Another recent work Ethereum Alarm Clock [39] allows a user to deploy a request contract with a future time limit on the Ethereum network. However, this project supports only one type of event, the arrival of a predefined time-frame. Work by Chao and Palanisamy [32] takes a similar approach to handle only events based on time.

In contrast, this work tackles the fundamental limitations seen in [25, 9, 39, 32] by proposing a smart contract platform based on the event-driven execution model, complete with a pub-sub scheme, which can be applied as a modification on the present Ethereum architecture or other smart contract platforms.

4 Overview of EDSC- Event-Driven Smart Contract Platform

EDSC is built on the event-driven execution model using the publish/subscribe communication paradigm. In the publish/subscribe interaction scheme, components subscribe to events of interest, or to a pattern of events, and are subsequently asynchronously notified by the system when any event published matches their registered interest. In order to incorporate this paradigm into a smart contract platform, the platform design should provide the following basic features to the participating smart contracts and external accounts:

- **Event Definition:** Any external account or smart contract in the system is able to define/register new and unique event types in the system. This is analogous to defining a class in an object-oriented programming paradigm.

- **Event Subscription:** Any smart contract in the system is able to subscribe or unsubscribe to a particular event type that is already defined in the system. At the time of subscription, the subscriber contract may specify additional logic that will be used by the system to evaluate

whether to invoke it in response to the event of interest's occurrence in the system.

- **Event Publishing:** Any smart contract is able to publish an event that has already been defined in the system.

In order to provision the three fundamental features mentioned above, the smart contract system needs to incorporate the following functionality specific to the event-driven execution model:

- **Event Definition Maintenance:** The event templates are saved immutably in the system. This may be achieved in practice by referencing the event definitions on the blockchain itself, similar to how smart contract code is stored on-chain by reference in Ethereum.

- **Subscription Information Maintenance:** The subscription information is also saved immutably in the system. This can also be achieved in practice by referencing the subscription information on the blockchain itself, similar to how smart contract code is stored on-chain by reference in Ethereum.

- **Event Matching:** Every time a published event is processed, the system determines all the smart contracts which are subscribed to that particular event. The system also evaluates the corresponding subscription logics of all those subscriptions to determine which smart contracts to invoke in response to the publishing event.

- **Event Queueing:** Based on the event matching, the system queues all the matching subscribed smart contracts for execution. Since the system is asynchronous, there are no guarantees as to when the subscription triggers will be executed. The system guarantees the queueing of these executions.

Since the publish/subscribe method is an anonymous and indirect communication paradigm, the system decouples the communicating entities i.e., the smart contracts in space and execution flow:

- **Space Decoupling:** The publishing and the subscribing smart contracts do not need to know each other since they are not required to address each other for communication. Hence,

the event publisher does not maintain a record of all the smart contracts which will be evoked in response to its event publication. Likewise, the subscriber may subscribe to events from multiple sources without specifying them individually.

- **Time Decoupling:** There is no provision for the publisher or the subscriber to run within any time constraint. The subscriber execution can be queued for a later time window (depending on future events).

- **Execution Flow Decoupling:** The inherently asynchronous communication decouples execution flow from inter-contract communication. A smart contract is not blocked when sending a notification to an external contract. The system can handle the subscriber execution in response to the notification by running it concurrently or queueing it for later. The subscriber and publisher of events do not have to be synchronized in their execution.

5 Advantages of EDSC

Based on our basic design framework from Section 4, the proposed smart contract system will offer attractive advantages to the ever-evolving ecosystem of smart contracts:

- **Lower Fee for All:** We reason that the proposed platform will result in a majority of the system participants having to pay a lower gas fee for their executions, especially in a system that is highly interfaced with external oracle systems through oracle contracts. User smart contracts only pay for the gas for executing themselves. In other words, the transaction cost, which is now the cost of putting the event on-chain, is shared by all the subscribers collectively.

- **Improved Security:** Ethereum smart contracts developed in Solidity have been marred with security issues centered around reentrancy and unexpected reverts [16]. This is because, by design, an Ethereum transaction has to complete the contract execution in the current as well as called contracts before the transaction is considered complete. On the other hand,

the event-driven paradigm is free from such vulnerabilities, since events are asynchronously published without waiting for the subscriber contracts to run. This offers better security guarantees.

- **Less Network Clogging:** Having multiple smart contracts subscribe to a single event translates to lesser network usage as opposed to smart contracts requiring transactions to be broadcasted every time they need to execute or interface with an oracle provider. Also, only a single event needs to be recorded on-chain as opposed to multiple transactions. This is beneficial for freeing up vital network bandwidth, which has recently been clogged [38].

- **Better Scalability:** We also claim that an event-driven system based on the proposed basic design is better positioned for employing parallel-processing and sharding solutions for scalability. This is because all executions are restricted in context to the currently executing contract, and event-based subscription triggering occurs asynchronously. All events can be posted to a shared global non-sharded trie for inter-shard communication borrowing from a similar concept in the Zilliqa project [46].

In contrast to the advantages offered, the proposed platform also has the following limitations compared with the transaction-driven model:

- **System State Predictability:** Once the proposed system gets large enough, it becomes more computationally expensive to predict beforehand, a single event's effects on the entire system state.

- **Increased node costs:** Compared to the transaction-driven model, running and maintaining a node will be more computationally expensive in the event-driven model. As mentioned in Section 3, each node has to maintain the subscription and event definition state of the entire system. Each node also performs evaluations related to subscription logic and has to maintain the triggered subscriptions' execution buffer. However, we argue that the increased throughput and scalability, along with the decreased network traffic will justify this trade-off.

6 EDSC System Design

We present the detailed design of the event-driven smart contract system based on the current Ethereum design framework. The rationale for this approach is that Ethereum is arguably the simplest and unarguably the most widely adopted smart platform to date. Using Ethereum's design as the base reference will allow the readers to grasp the design proposals clearly and draw parallels between the two approaches. Note that the design presented is general and independent of any specific platform.

6.1 Event Definition Trie

All events have to be defined in the system before smart contracts can subscribe to or publish them. The event definitions must be stored in the system immutably and free from loss. In the Ethereum context, this may be achieved by requiring all nodes to maintain the global event definition data locally. This event definition data can then be referenced on the blockchain for immutability. This is analogous to how the current Ethereum design maintains the state of the system. In other words, the event definition trie will need to be added to the Ethereum system, as illustrated in Figure 1.

As mentioned in Section 4, any smart contract or external account in the system has the ability to define a new event type. This can be done by posting a special type of event that is already predefined in the system. The special event's payload consists of the definition of the new event's template. Any node of the network, when processing this event, adds the event definition to their local event definition database. An event definition consists of the attributes listed in Table 1.

Table 1: Event definition attributes.

Attributes	Description
Event Identifier	This is a unique identifier for every event type that is defined in the system. One possible implementation is to use the hash of the entire event definition.
List of Variables	A list of variables and their types which are posted as payload whenever this event is published in the network.
Comments	This part is for documentation purposes and can be used to write a description of the event, document the variables in the payload, for specifying other information for potential subscribers like event generation frequency or for any other information which the event definition originator desires.

6.2 Event Subscription Trie

Once an event has been defined in the network's subscription trie, the system participants, i.e., the smart contracts, can subscribe to such events. Subscribing to an event allows a smart contract to be executed in response to the particular event getting posted in the network. Whenever the subscribed event occurs, the subscriber smart contract's default callback function is asynchronously invoked, and the event's unique identifier and the payload are passed as arguments. The smart contract can then execute the desired functionality accordingly.

Like event definitions, event subscription information also needs to be stored in the system immutably and without loss. We propose storing the subscription information in a similar manner to the event definition storage where it is stored across all nodes and referenced on the blockchain. In fact, the event definitions and subscription information can be combined into one trie which can be referenced on-chain, as shown in Figure 1.

Like event definition, event subscription also occurs through a special type of event that is predefined in the system. Smart contracts post this event to signal their desire to subscribe to a particular event type, which is passed as the payload of this predefined subscription event. The nodes of the network then update their subscription trie when this event is processed. In addition to the event type, the system also allows subscribers to specify other parameters of their subscription which are summarized in Table 2.

All the subscription parameters are stored in the subscription trie against each subscription and used by the system to determine which subscriptions to trigger in response to a generated event. Every time a smart contract makes a subscription to an event type, an entry against that

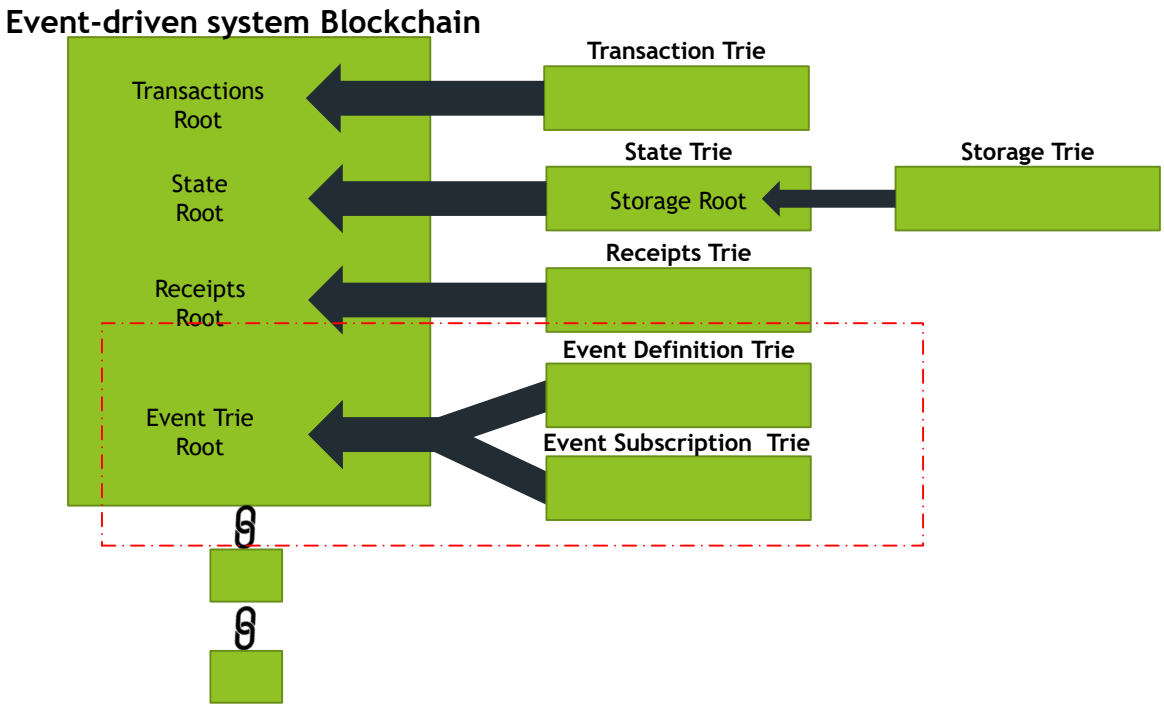

Figure 1: Subscription Trie addition to Ethereum design.

event type is added in the subscription trie with the subscriber smart contract's address and all the subscription parameters provided.

6.3 Event Generation

Events can be generated in the system in three ways. Firstly, by external or non-contract accounts, which is similar to generating a transaction in the Ethereum system. An external account has to digitally sign such an event, similar to Ethereum transactions. Such events are generated outside the system and then broadcast in the network. Secondly, events can also be generated by smart contracts using a specific event generation opcode. Such events can be thought of as the analogous functionality for the CALL opcode in the Ethereum domain but working asynchronously. Thirdly, certain special events can be generated by the system itself as described in Section 6.4. The second and third types of events do not have to be digitally signed and only exist in the execution environment. Only the external account generated events are recorded on-chain, similar to Ethereum's

Table 2: Event subscription attributes.

Attributes	Description
Event Identifier	The unique identifier for the event to subscibe to.
Gas Rate	Each subscriber contract at the time of subscription specifies the gas price that they will pay for their execution as a result of the subscription. The nodes decide which subscriptions to execute based on the gas price that they are willing to pay. So setting a higher gas price decreases the delay between event generation and subscription execution. Note that the design can be adapted to other public blockchain systems that apply alternative incentive mechanisms for running smart contracts.
Subscription Fee	The maximum fee that the subscriber is willing to pay the event publisher in order to be triggered by its generation (e.g., gas limit for event subscription in Ethereum).
Publisher Identifier	The subscriber also has the provision to only subscribe for execution when the event is published signed by a specific public key. This feature exists to allow a subscriber to only run in response to event publishers which they trust and to prevent spamming in the system. A subscriber may provide more than one public key.
Block Rate	The subscriber can also specify a block rate to run the subscription. For example, a subscriber may only want their subscription to be executed once every one hundred blocks for a frequently occurring event.
Event Rate	The subscriber can also specify the event rate to run the subscription. For example, a subscriber may only want their subscription to be executed once every hundredth instance of a specific event being generated.
Subscription Logic (Constraints)	The subscriber may also use the subscription logic field to specify any complex expression involving the block number, block time, event payload, event publisher public key, etc. This expression can then be evaluated to determine if the corresponding subscription should be triggered. The computational cost of evaluating the expression will be paid out of the subscriber's account at a fixed system rate.

Table 3: Event message attributes.

Attributes	Description
Event Identifier	The unique identifier of the event in the system.
Publisher Identifier	The public key of the event publisher is required when generating an event. The subscribers may use this information to subscribe to events from specific entities only. If the event is generated through an external account, then a digital signature corresponding to that public key is required to establish identity.
Payload	The event object also contains the event's payload arguments as defined in the event definition.
Subscription Fee	This is the fee that any contract which subscribes to this event must pay to the event publisher when it runs in response to the event. This is different from the gas fee which is paid to the miners as the computation, network, and storage costs of running the smart contract. The subscription fee exists purely to incentivize event publication on the network.
Inclusion Fee	This is the fee that the publisher is willing to pay the miners for the inclusion of their generated event in the block. This is only required of external account generated events.

transactions. More on the topic follows under Section 6.8.

An event generation message needs to contain the parameters summarized in Table 3.

6.4 Special Event Types

In addition to event definition and subscription/unsubscription events, there are two other special types of events in the system: the transaction event and the deploy event. A transaction event is an event to which every smart contract is subscribed by default and is triggered if the event contains that smart contract's address in its payload. The transaction event is used to transfer tokens

from one account/contract to another. The system automatically increments and decrements the receiver and sender's balances depending upon the value specified in the payload when this event is processed. Since the transactions are also now events, the proposed system processes all transfer of tokens asynchronously too.

The deploy event is used to deploy a new smart contract in the system and specifies the contract code in its payload. It is analogous to using a transaction in Ethereum to deploy a new contract.

In addition to these two events, there can also be other special events that are system generated. Currently, we propose to have one special system generated event, which is the new block event. This event can be generated by the system once for every block and contain information like the block number and other block parameters. This special event does not need to have an Inclusion and Subscription fee, and neither requires a signature. Smart contracts in the system may subscribe to this event, in order to be triggered automatically at certain block intervals.

6.5 Gas Fee for Computation

In a transaction-based system, the entity that generates a transaction has to pay for the gas fee associated with the computation, storage, and other costs of any smart contract code executed due to the transaction. This includes the contract code to which the transaction is sent and those contracts that the recipient code calls or invokes.

Such a design cannot be adapted with an event-driven model to avoid subscriber spamming. Hence, the natural design is to have any subscriber pay for the gas fee associated with running its code. This is also in line with the event-driven paradigm's space decoupling since the publisher does not have to concern itself with the subscribers to its event. The event publishers pay an inclusion fee (specified as part of the event data structure) each time they publish an event to the chain. This fee is specified explicitly if the event is published externally. In case the event is published internally, i.e., through a smart contract, the fee is determined by the gas fee being paid by the publisher contract for its execution.

The other side of this problem is a malicious contract spamming the network with events and

draining subscriber contracts of their ether. This can be addressed by allowing smart contracts to specify, at the time of subscription, to only run for events if generated by a particular public key (contain the corresponding digital signature) as described under heading 6.3. Such a system works, for example, even if an oracle platform has multiple nodes because they can all still generate events with the same public key signature. Additional logic and using the block rate and event rate variables also help prevent these spams from occurring.

All smart contracts specify at the time of subscription the gas price they are willing to pay for their execution. The system prioritizes subscription execution, depending upon the gas price offered. Perhaps in the future, a provision can also be made for a floating gas price, with a maximum value and a weight value defined in the subscription parameters and the gas price computed by the network dynamically based on the variable system traffic and the constant weight parameter.

6.6 Incentivization for Event Publishing

Smart contracts that publish events have no incentive to do so unless they are being compensated. For example, an oracle interface contract providing external data to the system through events needs to be compensated for its services. In a transaction-based system, this is pretty straightforward. The user contract pays the interface contract when it interfaces with it, i.e., generates the first transaction. In an event-driven system, this can be achieved by having the publisher describe a compensation rate each time it publishes an event. This is done through the subscription fee parameter in the event data structure. Each subscriber then has to pay the publisher the set fee in order for the system to execute the subscriber contract code's subscription. So the subscriber pays both the miners and the publisher for the subscription execution. Subscribers specify the maximum rate they are willing to pay through the subscription fee field at the time of subscribing.

6.7 Execution Independence and Atomicity

In the event-driven paradigm, smart contracts only interact with each other through posting and listening to events. Smart contracts do not have to make calls and wait for the execution of other

smart contracts before resuming their execution. This is the execution flow decoupling which the event-driven paradigm provides. Hence, once a smart contract starts executing in the proposed system, it completes its execution independently of other smart contracts. The system must ensure that the currently running smart contract finishes its execution before any new contracts triggered by the completed contract are run. The system guarantees atomicity in smart contract execution, and any exceptions raised in execution result in the current smart contract execution being reverted. However, no other contracts and their state is affected by the reversion.

This approach saves the systems from many troubles that exist in the transaction-based systems like cyclic executions and execution livelocks/deadlocks. This paradigm also prevents the publisher contract's execution state from being reverted by any subscriber contracts running out of gas or throwing an exception.

The system can still allow the reuse of smart contract code by having a similar opcode like that of DELEGATECALL in Ethereum. Since the called contract's code is executed in the state of the current contract, this will not violate the system's paradigm.

6.8 Subscription Execution and Selection

In a transaction-based system, the selection of contracts to execute is relatively straightforward. Upon receiving a new block, any node begins executing transactions in the order that they are present in the block transactions list, and for each transaction execution, all subsequent in-lying smart contract calls and executions are executed first in the form of a LIFO stack.

The situation is somewhat trickier in an event-based system. If executing a smart contract generates one or more events, we have to decide which subscriptions to execute first. The previous heading has already established that a running smart contract execution will complete before any other subscriptions are processed. Hence in the event-driven system, a buffer is maintained of all pending subscription executions and events to process. We mandate that the list is ordered based on whichever subscriptions pay more gas fee. If two subscriptions pay the same gas fee, then the older defined subscription gets the preference.

Whenever a new event is generated while running a smart contract, all its subsequent relevant subscriptions are added to the execution buffer, which is somewhat similar to the pending transaction pool in Ethereum. Subscriptions go into the buffer list, and their position is determined by the gas fee that they are paying. Whenever a subscription execution completes, the system will pick the next subscription from the list to execute. This buffer is not discarded between blocks and allows event-triggered subscription from previous blocks to run too, provided space is available. However, unlike the transaction-based model, there is no guarantee of a subscription being executed in the current or even succeeding blocks. Subscriptions paying more gas fee will always get the precedence in the system. Events that are generated externally have to be processed independently. The processing of an external event refers to determining the corresponding subscriptions to execute against it. External events also exist in the execution buffer competing for processing with subscription triggers, and the inclusion fee specified when generating these events determines when the system will process them. Once an event is processed by the system, its corresponding subscription triggers replace it in the pending executions buffer as illustrated in Figure 2.

Having a deterministic fee-determined execution rule allows us to do away with putting all events on-chain. Since the system follows a gas price determined precedence rule for subscription executions, all nodes will arrive at the same state, and there is no need to put contract generated events on-chain. This allows the system to be as efficient in its chain space usage as the transaction-based system in the worst-case scenario. Any advantage provided by using event subscriptions is a bonus.

6.9 Block Validation

For block validation, each node looks at only the events generated by the external accounts present in the event list for the current block. The node then proceeds to place these events in the pending subscription buffer. It then begins subscription executions/event processing from the pending subscription buffer, and subscription triggers for all the internal events generated by contracts are also placed in the buffer as they occur. The node keeps on executing subscriptions/event

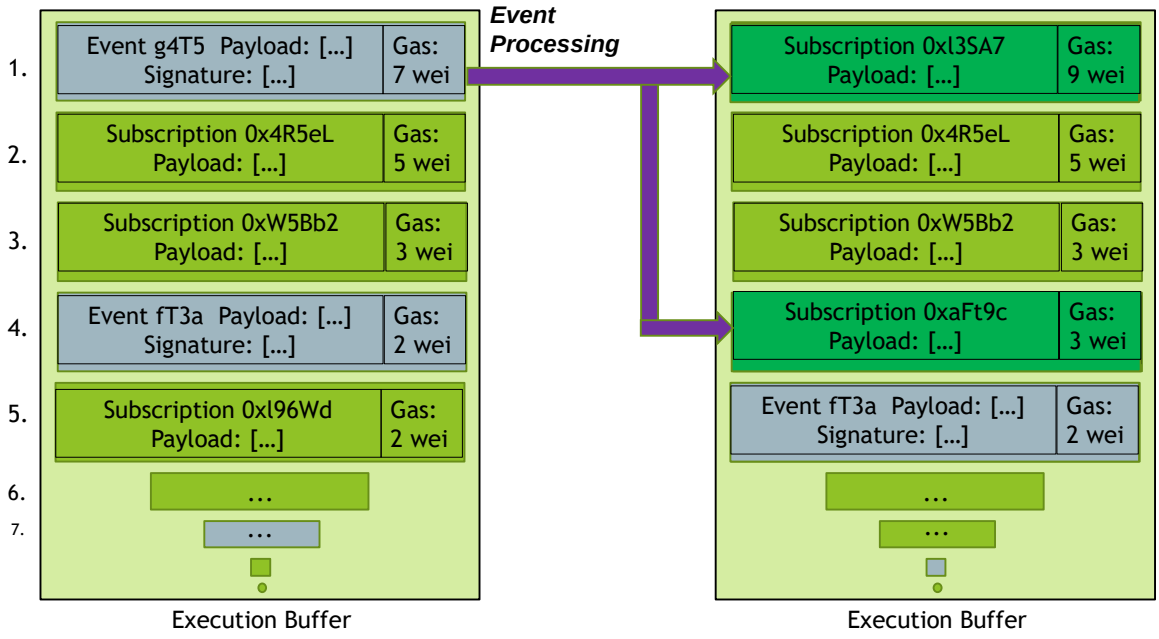

Figure 2: Event processing in execution buffer.

processing until the block gas limit is reached, or the buffer is empty. The node then compares the state, receipt, and event subscription/definition trie references to the ones provided in the block. If they match, the block is approved.

6.10 Parallel Processing and Sharding

The system design allows it to be a better candidate for sharding and parallelization solutions than transaction-driven models. Since a smart contract being executed is not dependent on other contracts' states, other contracts can be simultaneously executed in parallel on other shards. Because contract execution atomicity is guaranteed, there is no need for state locks. In case sharding is to be implemented, we propose having a shared subscription/event definition trie between the shards and a shared pending execution buffer. The shards can then divide contracts among themselves and only execute the relevant ones. Since the pending subscription execution list will be shared, they will have visibility to any events generated for the contracts in their domain. Any new events

generated will be broadcast on the network for all shards to see.

A summarized comparison between the transaction-driven execution model and the event-driven design proposed in this section is presented in Table 4.

Table 4: Comparison of Transaction-driven and Event-driven models.

	Transaction-driven model	**Event-driven model**
Communication	Smart contracts and external accounts communicate through transactions.	Smart contracts and external accounts communicate through events.
Execution Trigger	Smart contract executions are triggered by transactions initiated through external accounts.	Smart contract executions are triggered by events generated by external accounts and smart contracts.
Computation Cost	Execution costs of the triggered contract and all its calls are paid by the transaction initiator.	Subscribing smart contracts pay for their own execution costs and specify the gas fee when subscribing.
Synchronization	All communication between participants is synchronous with the triggering transaction.	All communication between participants is asynchronous.
Token Transfer	Tokens are transferred between participants through transactions.	Tokens are transferred between participants through a special transaction event.
Direct On-chain record	Only external account generated transactions are directly recorded on-chain.	Only external account generated transaction events are directly recorded on-chain.
Indirect On-chain record	The root of the transaction trie, state trie(includes storage trie) and receipts trie is referenced on chain.	The root of the transaction trie, state trie(includes storage trie), event state trie (includes event definition and event subscription trie) and receipts trie is referenced on chain.
Inter-contract Sends	Inter-contract token transfers are completed when triggering transaction is processed.	All inter-contract token transfers are asynchronous.
Transaction/Event Queueing	A pending transactions pool stores all the transactions that are yet to be processed.	A pending buffer stores all the events and subscription triggers that are yet to be processed.
Transaction/Event Ordering	Miners are free to decide the order of the transactions in a block.	Miners order the events/subscription triggers based on the gas price offered.
Transaction/Event effects	A transaction's effects on the system state only occur when the transaction is processed.	An event's effects on the system state may occur indefinitely (until the queueing buffer clears).

7 Security Considerations

The event-driven smart contract platform design offers numerous security benefits over a traditional transaction-driven model. For instance, when EDSC is integrated with Ethereum, the system can prevent inter-smart contract communication-related vulnerabilities, including reentrancy and denial-of-service attacks [16]. Since the proposed design provides execution flow decoupling, smart contract execution independence and atomicity, these vulnerabilities are no longer present in such a design. Aside from these vulnerabilities at the programming and toolchain layer, EDSC can also mitigate vulnerabilities arising from transaction ordering dependence [36]. EDSC achieves this by enforcing an order for event processing and subscription execution based on the gas fee. This design

choice also provides the additional benefit of not having to put smart contract generated events on-chain and allows for reduced block confirmation times. Below, we discuss some attack scenarios and mitigation approaches.

Denial-of-service (DoS) is a realistic risk for public blockchains. For instance, an attacker may try to flood the event processing module with event messages, or launch starvation attacks by polluting the event buffer. These attacks can be mitigated with a variety of countermeasures, for instance, imposing a limit on event update and event creation rates for an account. In addition, event publishing also has a gas cost associated with it, which is paid by the publishers, to discourage them from publishing events unnecessarily. Depending on the gas limit of an event, a registered event can be kept in the system for only a bounded number of blocks (limit can be re-fueled later by the creator with a transaction). Similarly, generating a new subscription/event definition and deploying a new contract also have an associated gas fee, which the event publisher must pay analogous to Ethereum's associated gas fee for deploying a new contract. These fees serve as a deterrent against spamming and DoS attacks on the system. Furthermore, event manager enforces that for each event and user account, there is a maximum number of transactions that can be triggered in each epoch, which prevents event buffer pollution. To further mitigate the risk of event publishers spamming the system, EDSC allows subscribers to use variables like the Event Rate and Block Rate as well as the Subscription Logic expression to control their frequency of subscription execution.

Malicious market exploiting and related cheating behaviors are another type of threats. In many DeFi applications such as DEX, a smart contract is applied to execute financial transactions. Such systems are exposed to various market-exploiting behaviors (e.g., frontrunning) [18]. Similar market-exploiting behaviors may pose a risk to EDSC. For instance, when a node observes an event update where financial value can be extracted, the node may send a shortcut message that registers to the event or updates its event registration to boost its priority in the event buffer. Similarly, when a miner detects an opportunity that value is extractable, the miner may be incentivized to directly insert a new event subscription or modify existing subscriptions. Since miner controls

event processing, the miner may take advantage of this position to order events or transactions generated from event subscriptions in a particular epoch in ways to extract values besides block reward and transaction fees. In EDSC framework, such attacks are prevented by the global event subscription state. The event subscription state is protected with Merkle hash tree and the root hash is included as part of block header. Updates to the global event subscription state are initiated through on-chain transactions and confirmed using the underlying blockchain consensus mechanism. The EDSC system enforces minimal delay for changes to the global event subscription state to be effective (minimal block delay). When a block is propagated and received by a peer, the peer will validate the transactions that are triggered by events according to the global event subscription state. This means that any foul play or dishonest manipulation of event triggered transactions can be detected by other peers and the block will be rejected.

EDSC is also susceptible to "freeloading" risk, i.e., freeloaders in the system can observe the events being published and copy the payload and publish the same events themselves at a lower subscription fee. This problem also exists for oracle systems like ChainLink [20]. Several methods can be applied to address this issue. For instance, ChainLink uses a commitment scheme to prevent such attack, which can be easily incorporated into EDSC.

Since EDSC can be implemented on any smart contract platform, vulnerabilities that are present in smart contract itself are not considered. We summarize all the security analysis in Table 5.

8 Implementation

8.1 Event Enabled Blockchain Node

The design of the proposed EDSC model can be implemented by extending Ethereum's implementation. We used the Golang implementation of the Ethereum client for these modifications. Extensions include adding support for messages for event definitions, event subscriptions and event publishing as well as mechanisms in the client for event and subscription processing and an execution buffer implementation for queuing subscription and event processing.

Table 5: Summary of event-driven design security considerations.

Layers	Threats	Analyses
Economic	Market exploiting attack (from miners or event subscribers)	Addressed by enforcing block delay for any update to event subscription states and Merkle hash.
	Miner ordering attack to realize extractable values	Addressed by validation of scheduling of triggered smart contracts according to the global subscription states by peers (protected by hash of subscription states stored in block header).
Programming & Toolchain	Reentrancy Attack	Addressed in the system by execution flow decoupling and having execution independence and atomicity.
	DoS with unexpected revert	
	DoS with gas limit exceeded	
	Unchecked call return value	
	Call stack depth limit exceeded	
Protocol	Transaction ordering dependence	Addressed by having gas price based execution order for subscriptions and events.
	Event publishing freeloading	Addressed by commitment scheme.
	Event spam attack on subscriber by publisher	Addressed by allowing subscriber to specify subscription logic and frequency.
	DoS by event spamming	Addressed by having gas fee associated with event registration and publication.
	Fairness	Addressed by enforcing upper bounds of triggered smart contracts per user account and/or per event in each epoch.
Data	Various	Addressed in the system by execution flow decoupling and having execution independence and atomicity.
Consensus	Various	
Network	Various	

In Ethereum, the P2P module is responsible for communicating with the underlying P2P network using a gossip-style strategy. It receives and routes various messages by communicating with its neighbors (nodes that are peers) under protocol manager. In case of EDSC, the blocks and transactions (now called external events) are propagated similarly. An external event can also be an event definition (a special type of event). A node receives and delivers external events to the extended protocol manager module that handles queues them in the execution buffer which is used to decide the next step of processing. Smart-contract generated (not external) event-related messages like event definition, event publishing, event subscription, event un-subscription, and event subscription updates do not have to be propagated on the network similar to Ethereum's internal transactions. A gas fee is charged for operations such as creating an event, making an event subscription, or updating an existing event subscription.

Event messages are signed using the ECDSA and secp256k1 digital signatures by the senders. There is a nonce in each event-related message. Event definition messages are used to register an event, identified with a 160-bit long unique identifier (created from the sender's account address and the event's definition). In addition to the event payload data, each event update message

identifies its associated event identifier and sender's address.

Each node implements an event buffer and an event manager for processing event messages. When a new external event message is generated / received, the ProtocolManager module first sends the new message to the event manager for validation, including verifying the signatures and checking other constraints and security requirements such as event update rates. When validation is passed, external event messages will be forwarded to the PendingPool of Ethereum TxPool where incoming and pending transactions are stored. Internal event messages are handled by the event buffer (evtBuffer module) because they are not processed as transactions (external events).

The TxPool module and the event manager notify the ProtocolManager module that there is a new event message that can be forwarded to other neighbors. Then, the ProtocolManager randomly selects $\sqrt{N}$ downstream peers that do not know the event messages as the targets to forward this message. For the remaining N - $\sqrt{N}$ downstream peers, the event message hash will be forwarded. A peer will receive event message hashes from its neighbors. When a node randomly selects one of the neighbors that have sent it the new event message each peer receives the hash of a new message, the node waits for a while (e.g., 500 ms). During this period, if there is no other neighbor sending the same event message to it, it sends a GetEvt message to the selected neighbor for requesting the new event message. After the requested neighbor returns the event message, the node first validates it. After validation, it is added to the TxPool of the node.

The extended protocol manager module processes the received event messages and delivers them to the evtBuffer module. A node maintains and keeps track of the event subscription state, as illustrated in Figure 3. This is a map from where the node can retrieve a list of subscriptions for each event that is published. Each subscription links to a smart contract and a function. For each event identifier, subscriptions are ranked based on priority (determined by gas price). For each event identifier, event definition updates are ordered using the nonce. For each epoch, based on the event subscription state and events in evtBuffer, a new set of internal event messages are generated. The internal event messages are added to the node's PendingPool. In the Ethereum client, PendingPool maintains the pending transactions that have not been included in the blocks

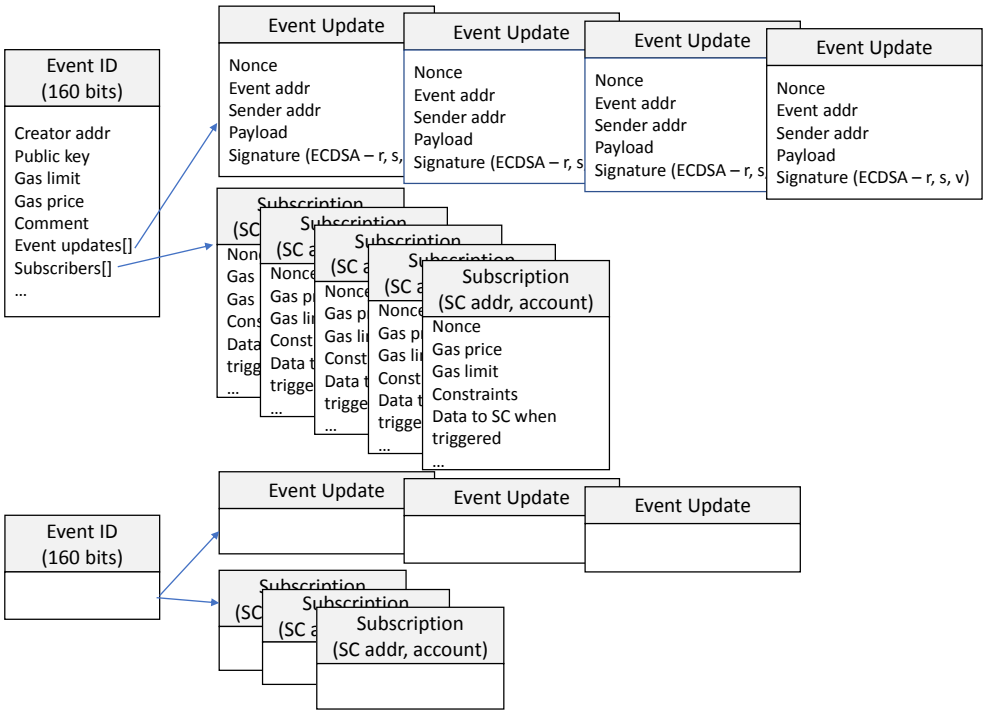

Figure 3: Illustration of event data structures and states.

on the blockchain but are ready to be packaged into a new block. Similar to how PendingPool tracks pending transactions for each account, the set of internal events enforces upper bounds for the number of internal event messages for an account.

After event messages are added to the PendingPool, the way they are sorted and picked for execution and block creation very much follows the same design as that of the Ethereum client. In case of proof of work consensus, the incentives for the miners to process and include the messages in the next block, are the block reward and the transactions fees. A miner uses the gas cost mechanism to calculate the fee for executing the subscribing smart contracts. To determine the fee for event subscription executions in a block, it uses the two attributes: gas limit and gas price. In short, the used gas multiplied by the gas price, corresponds to the fee that the miner receives, where used gas depends on the computational requirements of the smart contract [8, 5], but never exceeds the gas limit of the block. This is analogous to Ethereum's implementation. The complete algorithm is given in Algorithm 1 and the corresponding operations described in Table 6.

Algorithm 1: Event processing algorithm.

Input : tx pool *txPool*, event updates *newEvts*, event subscription state *evtSubState*
Output: *block*, updated *txPool*

```
Create a new empty block
Set tmpEvts = newEvts
While
    stop when txPool is empty
    stop when block limit is reached (gas limit or block size)
Set tmpEvts = validate-and-filter-evts(tmpEvts)
Set newTxs = create-tx-based-on-evts(evtSubState, tmpEvts)
Set txPool = merge newTxs with txPool
Set pendingTxs = tx-filter(txPool)
Set sortedTxs = sort(pendingTxs)
Set selectedTxs = pick top n best txs from sortedTxs
block, tmpEvts = execute-txs(selectedTxs, block)
End while loop
```

Table 6: Operations.

Operation	**Meaning**
validate-and-filter-evt	Validate and filter events including verification of signatures and constraints such as rate of event updates.
create-tx-based-on-evts	Create txs based on event subscription states (enforce rules such as k txs at most for each event based on priority).
tx-filter	Filter pending txs, for instance m txs at most for each account.
sort	Sort txs based on priority (e.g., gas fees).
execute-txs	Execute pending txs and add to block.

8.2 Modeling Tools

For modeling EDSC's functionality and experimenting with the design options in a scalable manner, we extended BlockSim [6], a framework and software tool based on discrete-event dynamic models for blockchain systems [1]. BlockSim supports the analysis of a variety of blockchain deployments as well as for design exploration and experimentation. It implements models for Bitcoin, Ethereum and other consensus algorithms. Results of BlockSim have been validated by comparing them with design properties and measurement studies available from real-life blockchains such as Bitcoin and Ethereum (see Table 7 for some results comparing real measurements). We modified BlockSim's full modeling technique for Ethereum to support the EDSC framework and event triggered transactions.

The model includes all the design features i.e. external and internal events, event subscriptions and definitions, blocks, transaction pool, and the blockchain ledger. Transactions (external event messages) created by a node are propagated to all other nodes in the network. Upon receiving a

[1]The reason to use Blocksim in addition to a private testnet is that Blocksim can model Ethereum transactions at a large scale and the tool is validated against real-world data.

Table 7: Blocksim result fidelity according to [6]

	Measured	**Modeling result**
Number of blocks included to the chain (per day)	6083	6079
Stale (uncle) rate	12.56%	12.55%
Throughput (transactions per second)	5.99	6.96

transaction, the recipient node appends it to the corresponding pool/buffer for event processing. The high level process workflow is shown in Figure 4.

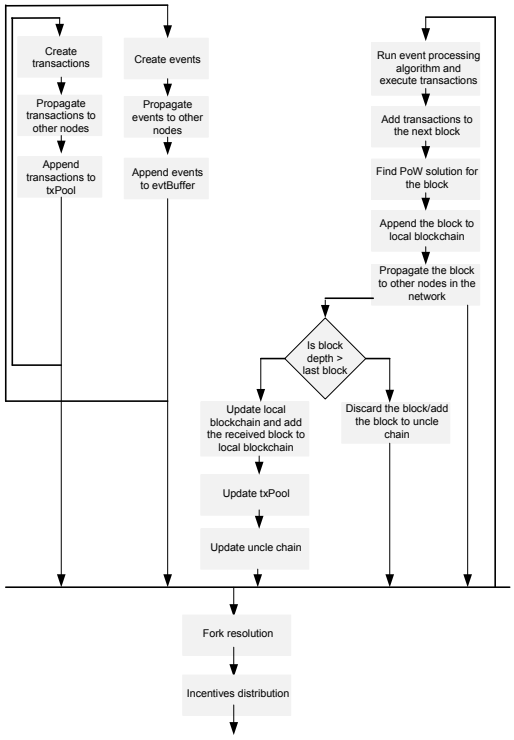

Figure 4: Model process after extending Blocksim.

9 Experiment Results and Analysis

We performed experiments with the extended clients and the BlockSim modeling tool. The implemented model of BlockSim for Ethereum has been validated using real data [6]. The model takes a set of parameters as inputs. This current implementation of the Ethereum baseline model compromises of 12.42s block intervals and a 2.3s block delay [6]. The model is configured to use the same

parameters as currently in Ethereum. The results are based on the averages of independent simulation runs of about 10,000 blocks. We compared the EDSC smart contract's delay of execution with the baseline delay of oracle based smart contracts in Ethereum. For this purpose we implemented a simple oracle contract that fetches external data for a smart contract once requested, and submits the fetched data in a new transaction. The delay is measured as the time when an event update is sent out by an oracle node(in case of EDSC) or a transaction for the fetched data is returned by the oracle node (in case of Ethereum) to the time when the transaction triggering the smart contract is added to a block of the longest global chain (for both EDSC and Ethereum).

As indicated by the results, EDSC achieves shorter delays for running contracts that subscribe to events, on average often less than the time of a block interval. In contrast, the baseline model incurs delays longer than three blocks (similar delays observed in Ethereum contracts in real life using oracle contracts: +3 block delays - see Figure 10 taken from [28]). This pattern is observed under different block intervals, varying from 8s to 60s.

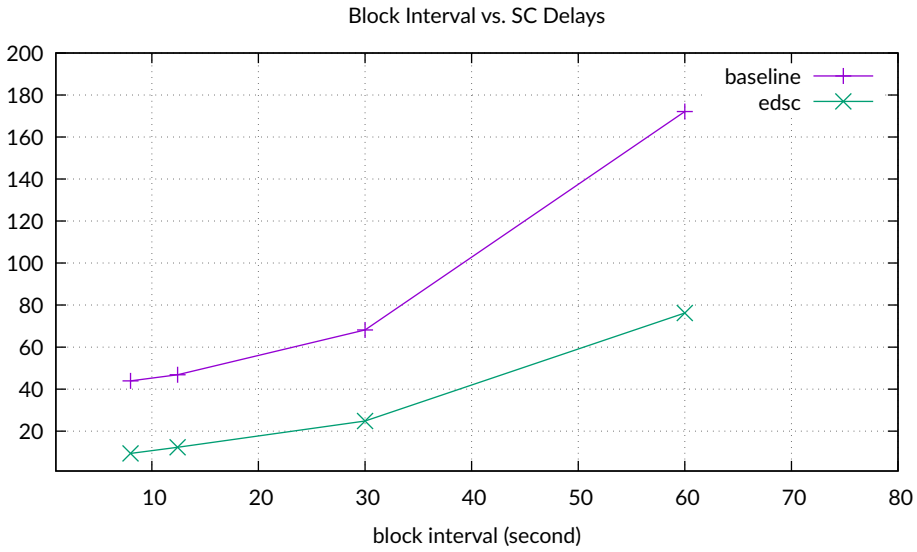

Figure 5: Block intervals and SC delays (seconds).

Both block delay and transaction(external event) delay can affect the latency between the event update and inclusion of event messages from the triggered contracts. One can assume that this

latency likely increases when either block delay or as the event/external event delay grows larger. Results in Figures 6 and 7 confirm this hypothesis. However, delays in the baseline model appear to be more affected negatively by block delay or transaction delay increase as illustrated by the expanding distance between EDSC delay and the baseline model delay.

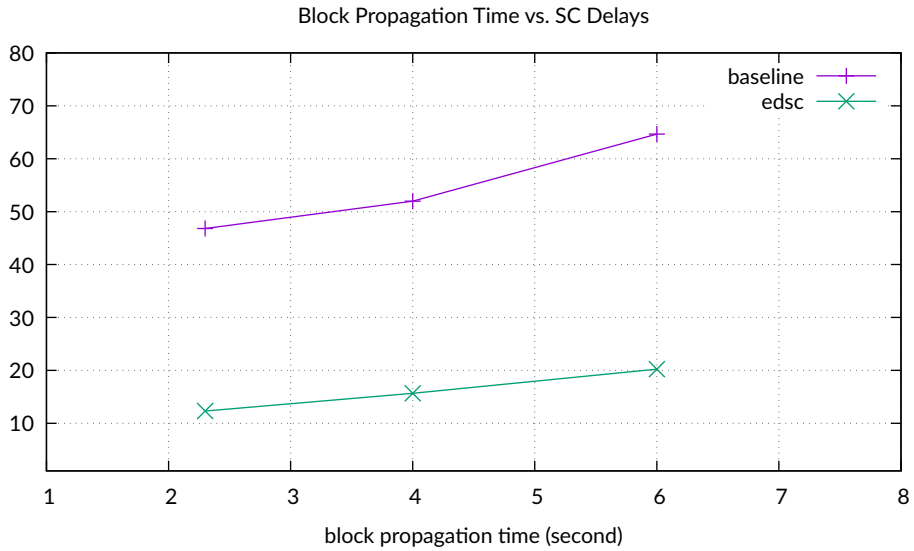

Figure 6: Block propagation time and SC delays (seconds).

Other factors may also have an influence on the latency of event driven contracts. Block size is one such factor. As suggested by results in Figure8, decreasing the block size will negatively impact contract latency under both models. However, the latency benefit of the EDSC model over the baseline is not affected.

On average, the delays under the EDSC model could be from 2.2 to 4.6 times less than the delays of the baseline model (depending on block interval, block delay, etc), which demonstrates its effectiveness for supporting contracts that demand timely execution based on events.

10 Example Use Cases

The event-driven paradigm is, by design, a better fit for many emerging smart contract applications. The event-triggered execution, asynchronous communication, and contract execution independence

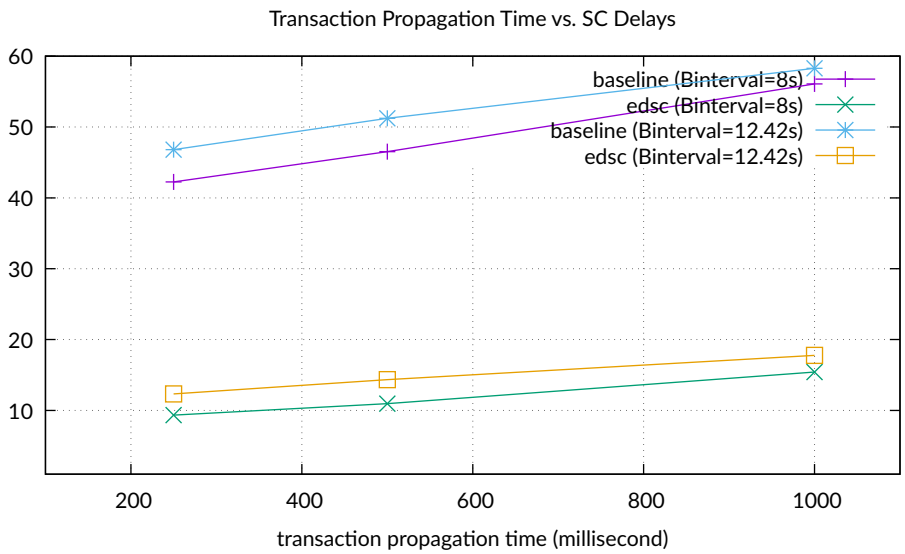

Figure 7: Event/transaction propagation time (millisecond) and SC delays (seconds).

and atomicity features are particularly instrumental in building scalable, adaptable, and easy to maintain applications on the smart contract enabled blockchains. The paradigm also enables these applications to be more reactive to external or internal triggers without overloading the system. In particular, financial applications like algorithmic trading, deploying financial instruments, real-time analysis, or digital asset management are naturally suited for the event-based model. The model also has been proven instrumental in a diverse application range consisting of supply chain management, online betting, oracle systems, gaming, etc. [23]. Here we elaborate on the design's benefits by discussing two broad real-world smart contract use cases.

Digital Asset Trading and DeFi Applications DeFi applications relay on third parties to report real-time information about the market price of the assets from real-world (off-chain) sources [35]. Consider implementing a digital asset trading platform on a blockchain-based smart contract platform. The system would need regular and timely updates on various market indicators like stock prices, trade volume, market trends, etc. Most of this external data is retrieved by employing oracle systems. In fact, the demand for oracle-provided data is dominated by DeFi projects' demand for external market price feeds [28]. In a traditional transaction-based system, this would require

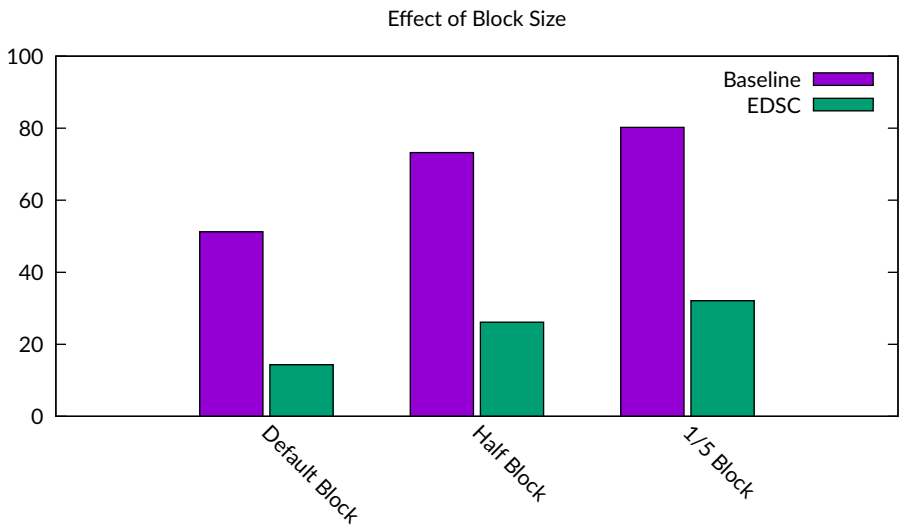

Figure 8: Block capacity and SC delays (seconds).

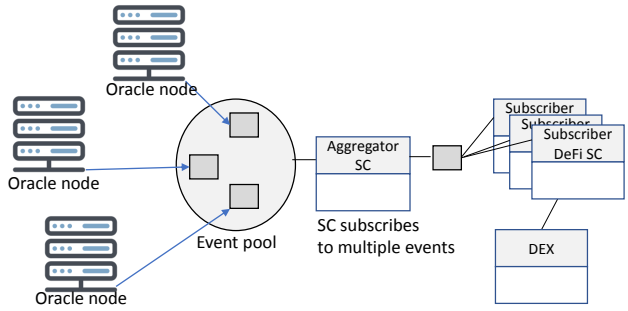

Figure 9: Illustration of DeFi application.

tedious and meticulous interfacing with multiple oracle system interfaces. Regularly managing the application would also not be easy. The implementer would have to figure out the interfacing details multiple times and familiarize themselves with the data formatting across multiple interfaces and providers. The two-way transactions for oracle fetches would burden the system if such applications were widely deployed.

In an event-driven platform, the integration is much simpler, cleaner, and easier to manage. The subscriber needs to know only the trusted publisher's address and the identifier of the event that they are interested in. The publisher might be a single entity or an oracle system. The event payload

format and documentation would be available on-chain and would not differ if multiple sources were publishing the same event. For example, if ten smart contracts are listening for a particular stock price from a publisher, it would not result in ten or twenty transactions going on-chain. Instead, only one external event is recorded on-chain, and all ten contracts can run by subscribing to this one event. In addition to a cleaner interfacing mechanism, easier maintenance, and lesser data on-chain, the event-driven paradigm also allows subscribers to listen for particular transactions. Since transactions are just a type of event in our design, participants using such financial applications might subscribe to transactions only from a particular party, only to a particular party, or random transactions exceeding a particular amount, etc. This is not achievable in the transaction-driven model. A transaction model might use external listeners to observe such events on the chain and then make transactions to trigger specific executions but cannot do it without incurring a block delay. We looked at the transaction data for ChainLink [20], which is the most popular oracle service provider for DeFi applications and see it having a 3-4 block delay on average while responding to oracle requests in the last eighteen months as shown in Figure 10.

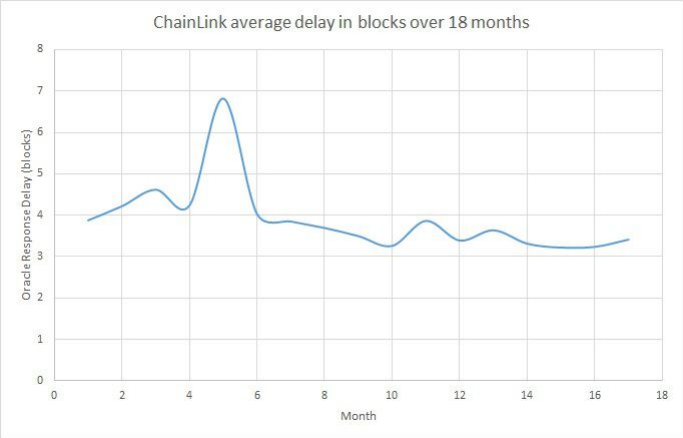

Figure 10: ChainLink response average block delay (May 2019 to Oct 2020).

Prediction Market Application Similar to the first scenario, a prediction application on the blockchain also benefits from an event-driven paradigm's features. Smart contracts can lay dormant unless executed by external events like the result of a sports match or an election. For long-term or

small bets, it might not be feasible to use the transaction model to poll for these external events or pay the fee for interfacing with an oracle system. The event-driven design makes such applications more feasible for smaller amounts since the event generation cost (maybe from a reputable news agency) is spread over numerous participants (subscribers). Again, the model also allows such an application to, for example, monitor newly placed predictions and adjust odds accordingly.

Hence, it shows that for many use cases, the event-driven design would be more cost-efficient (both computation and oracle fee), scalable, cleaner to implement, easier to maintain and allow for applications to have greater visibility on-chain data and token exchange.

It is worth mentioning that research on oracle service is complementary to event driven model of smart contract execution. These two are related but separate research topics. Our system can integrate various types of oracle services such as TEE based oracle service [49], oracle service employing secure multi-party computation [50], decentralized oracle service, etc. In fact, an event driven smart contract platform can arguably provide better support for integrating oracle services.

11 Conclusion

We proposed the concept for a novel event-driven smart contract platform with built-in event processing support on the blockchain. We presented a basic design as well as implementation of such a system in practice and commented on its potential benefits to smart contract use cases. We also presented analysis on its security aspects. Experiment results based on BlockSim extension are shown to illustrate performance advantages of event driven smart contract model. Being the first attempt to combine the two avenues of blockchain-based smart contracts and event-driven design, our work paves the way for future research on the subject in various directions. Future work can explore the application of this paradigm to implementing a sharding solution for scalability.

Acknowledgements: This material is based upon work supported in part by the National Science Foundation under award 1433817.

Part II

A SURVEY OF CHAINLINK ORACLES USAGE ON ETHEREUM

1 Abstract

Smart contracts are dependent on oracle systems for their adoption and usability. We perform an empirical study of oracle systems' usage trends and adoption metrics to provide better insight into the health of the smart contract ecosystem. We collect ChainLink usage data on the Ethereum network using a modified Ethereum client and running a full node. We analyze the collected data and present our findings and insights surrounding the usage trends, adoption metrics, oracle pricing and service quality associated with ChainLink on the Ethereum network. We infer that ChainLink's usage and growth are dominated by the DeFi ecosystem and for its demand for decentralized price feeds.

2 Introduction

Since the launch of the Ethereum [10] network in 2015, smart contracts [45] have become one of the central features of blockchain-based systems. Although initially limited in usage to token control and on-chain data access, smart contracts today are rapidly expanding their domain of applications [29] due to the availability of oracles [4]. Oracles provide the interface between the blockchain's isolated execution environment and external off-chain data sources, enabling smart contracts to retrieve and post real-world data and events. Consequently, the potential utility and future mass adoption of smart contract platforms is inextricably tied to the oracle service providers within the ecosystem.

Bearing that in mind, the motivation of this study was to survey oracle usage in the smart contract ecosystem. Currently, different projects like ChainLink [20], Provable [43] and Augur [42] are offering third party oracle services to smart contracts. These projects have adapted a decentralized approach for collecting and aggregating oracle data, thereby addressing "the oracle problem" [19] of having centralized points of failures in blockchain environments. For our survey, we target ChainLink, which evidently captures the majority share of the oracle middleware market at the time of writing. To establish this, we surveyed the top forty DeFi projects by market capitalization [17] and

found all among them which had a use case for external oracles to be using ChainLink except two projects. ChainLink provides a comprehensive list of their project integrations on their website [15] and it includes major DeFi projects such as Aave, Ampleforth, Chiliz, Polygon, Kyber Network and 0x among others. Although ChainLink provides its oracle services over multiple chains, we concern our study with ChainLink oracle usage on Ethereum since it is the most widely adopted smart contract platform at this time. We believe that ChainLink oracle usage on Ethereum represents the significant bulk of oracle traffic on smart contract platforms. Our study finds that Chainlink's growth and usage is strongly centered around the DeFi ecosystem where a few projects have been responsible for most of the oracle service traffic for price feeds. We also show that Chainlink's price feeds feature has seen a steady growth since its inception whereas the external API feature has seen negligible traffic. The oracle traffic statistics and trends provided by this survey can be used to gauge the adoption and health of the smart contract ecosystem in general. At the time of writing, we are not aware of any other formal study providing oracle usage insights in the smart contract environment.

3 Background

ChainLink is an oracle service provider for smart contracts that is currently live on three platforms: Ethereum, Binance Chain and the Matic Network. ChainLink went live in May 2019 and is currently the most popular oracle service provider for smart contracts. ChainLink maintains a decentralized oracle network and aggregates data from multiple oracle nodes on the network to provide data feeds that do not rely on a single oracle node or data source[12]. ChainLink employs an ERC-20 and ERC-677 compliant token called LINK which is used by oracle consumers to pay the oracle nodes for data provision. ChainLink currently provides three features for consumer smart contracts on the Ethereum mainnet.

3.1 Price Feeds

are a ChainLink feature to provide different market prices and conversion rates data in the blockchain environment for usage by smart contracts. ChainLink achieves this by having a decentralized price feed for each of these data points, which is fed price data through multiple oracle nodes using different sources. This is implemented by having an aggregator contract for each feed on-chain which is fed data by multiple oracle nodes through their interface contracts. The feed aggregator contract then aggregates all the nodes' answers to provide a final answer to any consumer contract via public Solidity functions. Consumers of the price feeds data call these aggregator contracts when the data is desired. The ChainLink documentation lists the aggregator contract addresses for the available price feeds[13]. The price feeds are sponsored by various projects and currently available for public usage without any LINK token charge.

3.2 External APIs

is a ChainLink feature that allows smart contracts in the blockchain environment to perform external API calls through ChainLink oracle nodes. These API calls can be HTTP Get Requests on the web or other APIs provided by the oracle node for different use cases. ChainLink API requests are currently handled 1:1 by an oracle and ChainLink currently does not provide decentralization benefits by default for API calls although a user might implement it on their own. The consumers of ChainLink's API feature have to pay their request servicing oracle node in LINK tokens for the service. The cost varies depending on the node and the nature of the request but is around 0.1 LINK on average and the highest being 1 LINK at the time of writing. Commonly used public API endpoints are available as "jobs" in ChainLink which allows user to only specify the job ID and not having to specify the URL, format etc. This makes the consumer side code more succinct and the implementation easier.

3.3 Verifiable Random Numbers (VRF)

is a ChainLink feature to provide verifiable random number generation functionality on-chain. ChainLink achieves this by having off-chain random number verifier contracts which verify the randomness of the number generated by an oracle node in response to a consumer request. VRF feature allows for provable random numbers, which protects the consumer from attacks even if the node servicing the request has been compromised.

4 Study Design

4.1 Data Collection

For both the Price Feeds and the External APIs we collected data from the launch of ChainLink mainnet in May 2019 up till the end of October 2020 (Ethereum block 11167816). The VRF feature data was not collected and is not part of this study since it only went live at the end of October 2020 and the resulting data was insufficient for a formal study.

Modified Ethereum Client

For collecting the Price Feed usage data, we looked at the price feed addresses available on the ChainLink website [13]. There were 88 price feed addresses at the time of writing which are proxy aggregator addresses. ChainLink has also, since its launch, made upgrades to the aggregator contracts. The current version of aggregators are labeled as v3. We used the wayback machine web archives [47] to retrieve old aggregator addresses and had a total of 169 addresses for our study (88 v3, 80 v2, 1 v1). The ChainLink team also later provided us with historical addresses which we used to verify our list. For capturing the price feed data we could not use the Web3 API since all price feed consumer requests were direct calls or "internal transactions". Hence we modified the Golang Ethereum client code to log data when internal function calls were made to these 169 addresses. We captured the block number, calling address, opcode, value and input data parameters for these internal calls to these addresses and stored them in a local MySQL database.

Ethereum Full Node and Web3

For collecting data related to ChainLink API usage we used the Web3 API with an Ethereum full node that we ran locally. ChainLink implements the API feature using the CallAndTransfer() functionality of the ERC-677 token standard. Every time a consumer requests an oracle, it generates a ChainlinkRequest event and sends the LINK to the oracle node along with data describing the API to fetch, the job ID, the format of the output, the callback address and function which the oracle must respond to and other data if required. The oracle node interface contract generates an OracleRequest event upon receiving the LINK and data and the external node listens to this event. It responds with the result after some time and makes a transaction to the callback function with the data response. The consumer contract then raises a ChainlinkFulfilled event. We use the Web3 APIs to capture these events and extract the required data which includes: the block number of the request, the requesting address, the oracle node requested, the job ID specified, the callback function and address provided, the LINK token paid, the ChainLink request ID, the request transactions hash, any additional data provided, the response block number, the response and the response transaction hash. We store the results in our local MySQL database for all such oracle service request-response cycles on ChainLink.

We used Etherscan [1] to verify various samples of our collected data to ensure that our data collection process was performed correctly.

4.2 Study Objectives

The study was aimed at providing insights into the usage of ChainLink oracles on Ethereum. For this purpose we looked at the following five aspects:

- **Oracle usage trends and demographics**
- **Oracle Adoption**
- **Oracle Pricing**
- **Oracle Servicing Delays**

Table 8: Price Feeds and API: collected data summary.

Feature	Total Requests	Excluding ChainLink Internal Requests	Distinct Caller /Consumer Addresses	Distinct Callee Addresses (Price-Feeds/Oracle Nodes)
Price Feeds	2409074	N/A	294	129
External APIs	2717049	6634	271	159

5 Results

5.1 Usage Trends and Demographics

After the data collection was completed and the required data was populated into our MySQL server, we had the quantitative information summarized in Table 8. A total of 2,717,049 API requests were made to Oracles during the entire duration of our study and in total 2,409,074 price feed calls were made to ChainLink's public price feed contracts for fetching the market place data. Although the numbers appear encouraging at first sight, upon further investigation, we found that 99.75% of API requests to ChainLink oracle nodes were made by ChainLink price feed aggregator addresses themselves. This is because prior to the v3 aggregator release in August 2020 [14], all price feed aggregator contracts made API requests to oracle nodes to fetch prices. After removing these API requests, we are only left with 6634 API requests performed on ChainLink for the entire 18 month period! We also see that the number of distinct users that made use of these features is very low.

Next, we present a list of the most popular price feeds based on their share of the historical price-feed traffic in Figure 11 as well as the total calls made to these price feeds in Figure 12. Figure 13 shows the most regular ChainLink price feeds consumers. We also present the corresponding consumer projects/contracts of these price feeds ordered by their share of the historical price-feed traffic. To get the corresponding projects/contracts, we grouped the most regular consumer addresses (Top 26 addresses, which represent more than 90% of all price-feed traffic) by their public tags available on Etherscan [1]. Our results show that Synthetix [2], which is a blockchain-based derivatives trading platform, is responsible for more than 47% of the historic price feed traffic. If we subtract ChainLink's internal traffic from the numbers, Sythetix's share of the historical price feed traffic rises to 75%.

Feed Name	Percentage of Traffic
ETH / USD	10.5993
EUR / USD	6.6621
AUD / USD	6.6246
XAG / USD	6.2909
XAU / USD	6.2669
GBP / USD	5.7996
CHF / USD	5.7952
JPY / USD	5.4070
USDC / ETH	4.4811
DAI / ETH	3.2655

Consumer Tag	Percentage of Price Feed Requests
Synthetix	57.4234
ChainLink Proxy Price Provider	17.0403
Unknown	12.8212
ChainLink Aggregator Facade	6.1165
CREAM Finance	3.5961
ENS	1.0431
Nexus Mutual	0.9570
ChainLink Adapter	0.5170
BZX	0.4854

Figure 11: Leaderboards: Price Feeds attracting the most traffic and projects generating the most price feed traffic.

Feed Name	Total Number of Calls
ETH / USD	255352
EUR / USD	160501
AUD / USD	159596
XAG / USD	151558
XAU / USD	150978
GBP / USD	139722
CHF / USD	139614
JPY / USD	130263
USDC / ETH	107956
DAI / ETH	78671

Figure 12: Most popular ChainLink Price Feeds.

5.2 Oracle Adaption in the Market

To study ChainLink oracles' adaption trends in the market, we look at the historical data for the average number of price-feed and API requests made to ChainLink oracles per month Figure 14. Plotting the data, we can see that the price-feed feature appears to be far more popular among users and has been rapidly gaining more traffic volume. The API feature does not appear to have a large demand among the users. We believe that this can be attributed to the fact that most projects and use-cases are able to fulfill their data needs using the ChainLink provided price feeds and do not have to employ a custom API. We also show in Figure 15 that ChainLink has continuously increased the number of price feeds being offered to users. The increase in price feed offerings has kept up with the increase in adaption as evidenced in these figures. In contrast to the price-feeds, Oracle nodes have not seen a marked increase in the variety of API calls and jobs being requested.

Consumer Address	Total Requests	Etherscan Tag
0x9D7F70AF5DF5D5CC79780032d47a34615D1F1d77	447844	Synthetix
0xba727c69636491ecdfE3E6F64cBE9428aD371e48	329178	Synthetix
0x76B47460d7F7c5222cFb6b6A75615ab10895DDe4	320801	ChainLink Proxy Price Provider
0xE95Ef4e7a04d2fB05cb625c62CA58da10112c605	274154	Synthetix
0xd8Aa8F3be2fB0C790D3579dcF68a04701C1e33DB	241133	Unknown
0xbCc4ac49b8f57079df1029dD3146C8ECD805acd0	112522	Synthetix
0x4B7dbA23beA9d1a2d652373bcD1B78b0E9e0188a	78439	CREAM Finance
0xdB2Ae36C2e9C00070e5bF752Be1FA2d477E98BDa	72154	Synthetix
0xB103ede8AcD6F0c106b7a5772e9D24e34f5Ebc2C	54489	ChainLink Aggregator Facade
0xBB1d497EDa7533e71A96d2Be6c38B96CF6611903	25596	Unknown
0x63FaF46Dadc9676745836289404B39136622B821	22753	ENS
0xc628050Cc70d761FA62b8AF7D1ef4cA883C2Fd3b	20875	Nexus Mutual
0x3561Eafb0a3cBF8E7e0FBd55D6D97daff6A9c987	20488	ChainLink Proxy
0x85aB3512465f39b8BB40a8872f8FBfD5f08AcE1E	20442	ChainLink Aggregator Facade
0xA9A88F8bdffA157C7A0D6e82e27E5f7164DAF8Fe	19404	ChainLink Aggregator Facade

Figure 13: Most regular ChainLink Price Feeds consumers.

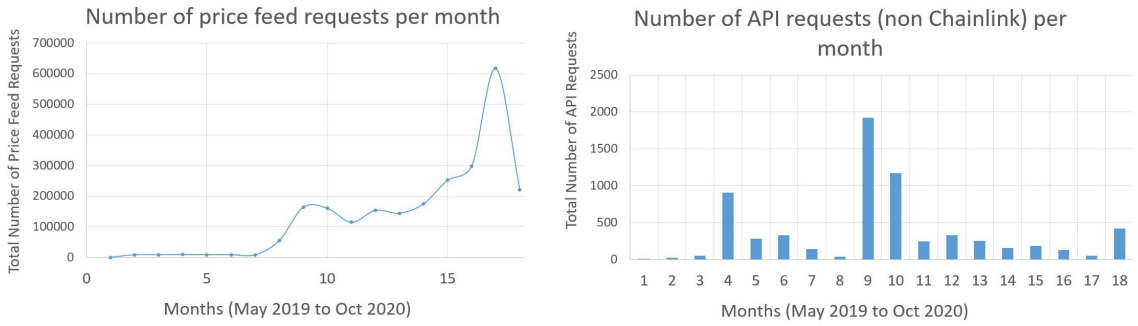

Figure 14: Number of price feed and API requests on ChainLink by month.

5.3 Oracle Pricing

ChainLink is currently providing the price feeds feature to all smart contract users on the Ethereum chain without cost. These price feeds are sponsored by various blockchain projects using these feeds in their contracts. A user does need to pay an Oracle node in LINK token if they make a direct API request. The current cost of ChainLink API usage varies and can be as high as 1 LINK depending on the oracle and the data being requested. We look at the historical price paid for running a single API request in Figure 16. We also look at the historical average income which the data providing oracle nodes from these requests. We see the average LINK paid for oracle requests on ChainLink is increasing of late, and that coupled with the increase in the LINK token price is bound to discourage the use of oracle APIs for trivial use cases.

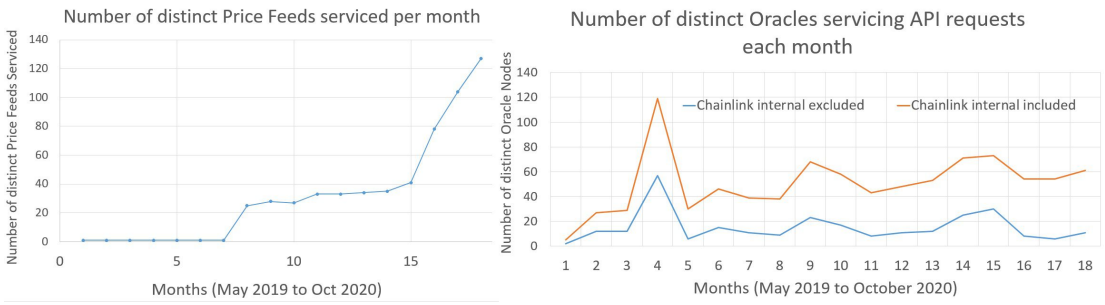

Figure 15: Number of distinct Price Feeds serviced and active Oracles by month.

5.4 Oracle Servicing Delays

Different smart contract use cases require their oracle service requests to be processed within a time constraint. For the wide adoption of smart contracts, it is essential that the oracle system is able to service time-critical requests. We analyze our available API data in Figure 17 to determine the historical average delay experience on ChainLink API requests. Due to a small number of outliers, the average obtained was around six hundred blocks. After filtering out these outliers and only keeping the requests that were serviced within one hundred blocks, we obtained the data shown in our figures. We can see that for ChainLink oracles most API requests are serviced within the next four to five blocks with the historical average block delay being close to four Ethereum blocks which corresponds to roughly one minute.

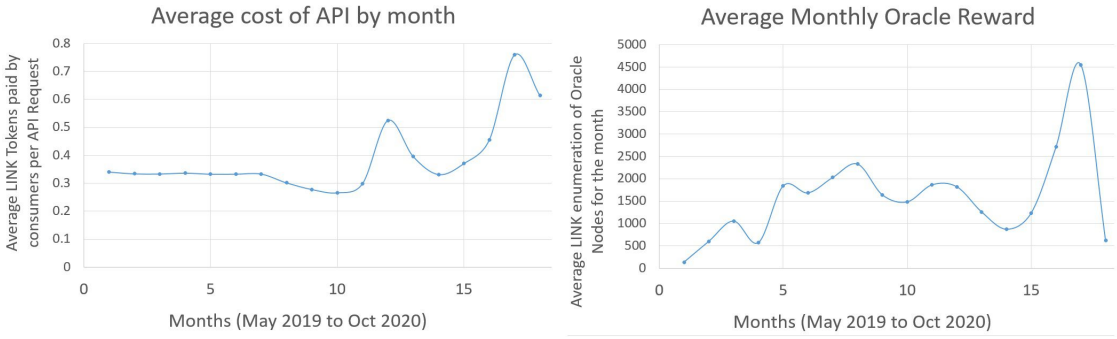

Figure 16: Average cost of a single API request and the Average fees collected in LINK by oracle nodes.

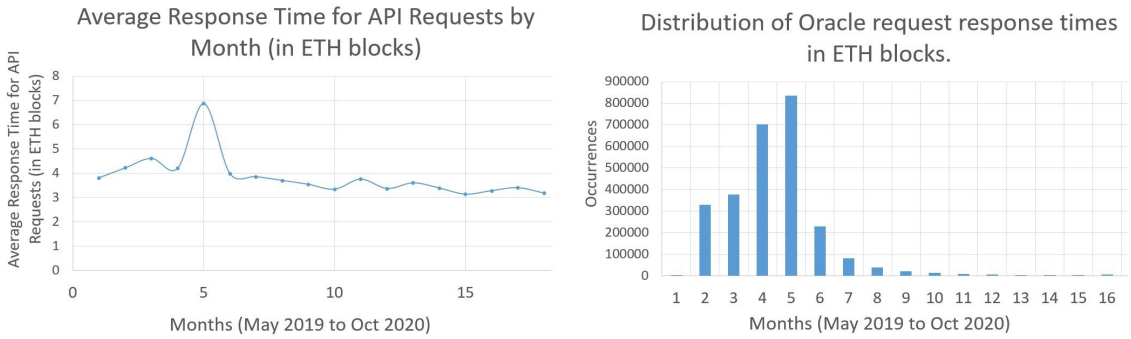

Figure 17: Average response time and response time distribution for API requests.

6 Analysis and Conclusion

Based on our analysis of the collected data, we obtained the following important insights regarding Chainlink usage on Ethereum in particular and the trends for Oracle systems in general:

- The number of individual users of the ChainLink platform is not very high. Currently, it is mostly being used by DeFi(Decentralized Finance) projects and applications to provide market prices to its contracts. This is perhaps indicative of a trend in the smart contract ecosystem in general.

- Currently, a single DeFi project, Synthetix has been responsible for almost 75% of the historic price-feed traffic in the ChainLink network (given that we ignore ChainLink's self-generated traffic). Synthetix uses various commodity and currency ratio feeds on ChainLink which are among the feeds that have serviced the most traffic. This dominance of Synthetix related traffic might fade with ChainLink increasingly integrating with new projects.

- The data shows that there is currently not a big market of people wanting to use oracles to connect smart contracts to the external world for trivial use cases. Whether it is the genuine lack of market demand for these applications or whether high Ethereum gas prices and ChainLink API fees discourage people from doing so will require further investigation.

- While ChainLink's API feature has not seen increased use with the rise of DeFi, ChainLink's price feeds have seen increasing usage since the project's launch. ChainLink has also managed

to provide an increasing variety of price feeds to cater to the demands of new DeFi projects integrating with ChainLink.

- The rising average API cost seen on the network might be attributed to the increased LINK token price which forces people to only make Oracle API usage for non-trivial cases.
- The average response time of ChainLink's API feature is seen to remain steady between 4 and 5 blocks which might not be good enough for time-sensitive applications.

In conclusion, at the time of this study, the ChainLink ecosystem on the Ethereum network appears to be driven purely by DeFi's demand for decentralized market price feeds [35]. In the coming future, it would be interesting to see if Oracle platforms like ChainLink take initiatives to attract other segments of users or tailor themselves more towards fulfilling the needs of the growing DeFi market.

Acknowledgements We warmly thank the ChainLink team for sharing historical price feed addresses with us for cross-verification.

Bibliography

[1] Etherscan: The ethereum block explorer. `https://etherscan.io`, 2017.

[2] Synthetix. `https://www.synthetix.io`, 2019.

[3] Al-Bassam, M., Sonnino, A., Bano, S., Hrycyszyn, D., and Danezis, G. Chainspace: A sharded smart contracts platform. *arXiv preprint arXiv:1708.03778* (2017).

[4] Al-Breiki, H., Rehman, M. H. U., Salah, K., and Svetinovic, D. Trustworthy blockchain oracles: Review, comparison, and open research challenges. *IEEE Access 8* (2020), 85675–85685.

[5] Aldweesh, A., Alharby, M., Solaiman, E., and van Moorsel, A. Performance benchmarking of smart contracts to assess miner incentives in Ethereum. In *2018 14th European Dependable Computing Conference (EDCC)* (2018), pp. 144–149.

[6] Alharby, M., and van Moorsel, A. Blocksim: An extensible simulation tool for blockchain systems. *Frontiers in Blockchain 3* (Jun 2020).

[7] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference* (2018), pp. 1–15.

[8] Baird, K., Jeong, S., Kim, Y., Burgstaller, B., and Scholz, B. The economics of smart contracts. *arXiv preprint arXiv:1910.11143* (2019).

[9] Baizel, E. Building an event-based application with amazon managed blockchain. `https://aws.amazon.com/blogs/database/building-an-event-based-application-with-amazon-managed-blockchain/`, 2020.

[10] Buterin, V., et al. Ethereum: A next-generation smart contract and decentralized application platform. *URL https://github. com/ethereum/wiki/wiki/% 5BEnglish% 5D-White-Paper 7* (2014).

[11] Cachin, C., et al. Architecture of the hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers* (2016), vol. 310.

[12] ChainLink. Chainlink developer documentation. `https://docs.chain.link`.

[13] ChainLink. Ethereum price feeds. `https://docs.chain.link/docs/ethereum-addresses`.

[14] ChainLink. Developer communications. https://docs.chain.link/docs/developer-communications, August 2020.

[15] ChainLink. Ecosystem. https://chainlinkecosystem.com/ecosystem, March 2021.

[16] Chen, H., Pendleton, M., Njilla, L., and Xu, S. A survey on Ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys 53*, 3 (2020), 1–43.

[17] Coinmarketcap. Defi category. https://coinmarketcap.com/defi, March 2021.

[18] Daian, P., Goldfeder, S., Kell, T., Li, Y., Zhao, X., Bentov, I., Breidenbach, L., and Juels, A. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *arXiv preprint arXiv:1904.05234* (2019).

[19] Egberts, A. The oracle problem-an analysis of how blockchain oracles undermine the advantages of decentralized ledger systems. *Master Thesis, EBS Universität für Wirtschaft und Recht* (2017).

[20] Ellis, S., Juels, A., and Nazarov, S. Chainlink: A decentralized oracle network. https://link.smartcontract.com/whitepaper, 2017.

[21] EOS.IO. Eos.io technical white paper v2. https://github.com/eosio/documentation/blob/master/TechnicalWhitePaper.md, 2018.

[22] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. The many faces of publish/subscribe. *ACM Computing Surveys*.

[23] Hinze, A., Sachs, K., and Buchmann, A. Event-based applications and enabling technologies. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems* (2009), pp. 1–15.

[24] Hull, R. Blockchain: distributed event-based processing in a data-centric world. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems* (2017), pp. 2–4.

[25] Jennings, T. Listen to events from a distributed blockchain network. https://developer.ibm.com/technologies/java/tutorials/listen-to-events-from-a-distributed-blockchain-network/, 2020.

[26] Kaleem, M., Kasichainula, K., Karanjai, R., Xu, L., Gao, Z., Chen, L., and Shi, W. An event driven framework for smart contract execution. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems* (2021), pp. 78–89.

[27] Kaleem, M., Mavridou, A., and Laszka, A. Vyper: A security comparison with solidity based on common vulnerabilities. *arXiv preprint arXiv:2003.07435* (2020).

[28] Kaleem, M., and Shi, W. Demystifying pythia: A survey of chainlink oracles usage on Ethereum. *arXiv preprint arXiv:2101.06781* (2021).

[29] Kehrli, J. Blockchain 2.0-from Bitcoin transactions to smart contract applications. *Niceideas, November. Available at: https://www. niceideas. ch/roller2/badtrash/entry/blockchain-2-0-frombitcoin* (2016).

[30] Kiayias, A., Russell, A., David, B., and Oliynykov, R. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference* (2017), Springer, pp. 357–388.

[31] Lerner, S. D. Rootstock: Bitcoin powered smart contracts v11. `https://www.rsk.co/Whitepapers/RSK-White-Paper-Updated.pdf`, 2019.

[32] Li, C., and Palanisamy, B. Decentralized privacy-preserving timed execution in blockchain-based smart contract platforms. *2018 IEEE 25th International Conference on High Performance Computing (HiPC)* (2018), 265–274.

[33] Li, C., and Palanisamy, B. Eventwarden: A decentralized event-driven proxy service for outsourcing arbitrary transactions in ethereum-like blockchains. *ArXiv abs/2004.12793* (2020).

[34] Liu, B., and Szalachowski, P. A first look into defi oracles. *arXiv preprint arXiv:2005.04377* (2020).

[35] Liu, B., Szalachowski, P., and Zhou, J. A first look into defi oracles. *arXiv preprint arXiv:2005.04377* (2020).

[36] Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), pp. 254–269.

[37] Mazieres, D. The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation 32* (2015).

[38] McIntosh, R. Ethereum struggles under the weight of defi growth: Is eth doomed to fail? Finance Magnates, `https://www.financemagnates.com/cryptocurrency/news/ethereum-struggles-under-the-weight-of-defi-growth-is-eth-doomed-to-fail`, September 2020.

[39] Merriam, P. Ethereum alarm clock. *Online document. URL http://docs. ethereum-alarm-clock. com/en/latest* (2015).

[40] Nakamoto, S. Bitcoin: a peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2008.

[41] Parzyjegla, H. *Engineering Publish/Subscribe Systems and Event-Driven Applications*. PhD thesis, University of Rostock, Germany, 2012.

[42] Peterson, J., Krug, J., Zoltu, M., Williams, A. K., and Alexander, S. Augur: a decentralized oracle and prediction market platform (v2. 0). *Whitepaper, https://augur. net/whitepaper. pdf* (2019).

[43] Provable. `https://provable.xyz`. Accessed on Sep. 10, 2020.

[44] Richards, M. *Software Architecture Patterns*. O'Reilly Media, Inc, 2015.

[45] Szabo, N. Formalizing and securing relationships on public networks. *First Monday 2*, 9 (1997).

[46] Team, T. Z. The zilliqa technical whitepaper. https://docs.zilliqa.com/whitepaper.pdf, 2017.

[47] Wayback Machine. The internet archive. https://archive.org/web/.

[48] Zamani, M., Movahedi, M., and Raykova, M. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), pp. 931–948.

[49] Zhang, F., Cecchetti, E., Croman, K., Juels, A., and Shi, E. Town crier: An authenticated data feed for smart contracts. Cryptology ePrint Archive, Report 2016/168, 2016. https://eprint.iacr.org/2016/168.

[50] Zhang, F., Maram, S. K. D., Malvai, H., Goldfeder, S., and Juels, A. Deco: Liberating web data using decentralized oracles for tls. *arXiv preprint arXiv:1909.00938* (2019).