This article has been accepted for publication, but has not been fully edited. Content may change prior to final publication.

Towards Eidetic Blockchain Systems with Enhanced Provenance

Shlomi Linoy
Faculty of Computer Science
University of New Brunswick
Fredericton, Canada
Email: slinoy@unb.ca

Suprio Ray
Faculty of Computer Science
University of New Brunswick
Fredericton, Canada
Email: sray@unb.ca

Natalia Stakhanova
Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
Email: natalia@cs.usask.ca

***Abstract*—Modern Blockchain systems with smart contract support are continuing to be adopted rapidly across various industry sectors and are increasingly used to manage valuable assets. As the size and complexity of smart contract applications increases so are the coding errors, exploit potential, and regulation requirements. For these reasons it has become necessary to efficiently manage the system's historic execution information, or *provenance*, to enable efficient analysis. Existing research enable to efficiently access historic data from within a smart contract, however it does not track what initiated the changes. We propose a system that enables to efficiently manage historic information of smart contracts calls, their parameters, and the Blockchain state before and after a call. We further explore how querying this historic data in different granularity levels can facilitate the debugging of a use case example that is comprised of multiple smart contracts calls across different entities.**

Blockchain, provenance, call graph, defect analysis, debugging, regulatory requirements, audit

I. Introduction

Blockchain systems increasingly capture the attention of academia as well as industry at a rapid rate. The adoption of Blockchain technologies is expanding in various industry sectors such as healthcare [3], IoT [4], and securities trading [5]. With a projected market size of $4.19 billion in 2020 and $162.84 billion in 2027 [13]. Modern Blockchains such as Ethereum [1] and Hyperledger [2] make use of smart contracts (abbreviated as 'contracts'), which are programs that operate on the Blockchain's current global state and produce a new global state. Each contract can be executed by a user, a contract, or can periodically run using a scheduler.

As the technology mature, companies are increasingly using Blockchain applications to manage valuable assets, including crypto-currencies, securities, real estate and valuable tangible assets. Hence, it is important that contracts are free of defects. Issues such as coding errors, malicious code and non-compliance with regulations can result in hefty financial consequences. For instance, it has been reported [14] that a software bug involving the replacement of += operation with =+ enabled the loss of assets worth $800,000. Another incident [15] involved an attacker exploiting a defect in contract code to cause a $80 million loss. A Blockchain system that can efficiently manage the provenance of both data and contract execution flow can be used to facilitate root-cause analysis of anomalies, support auditing, and investigate defects or malicious activities in the contracts. However, existing Blockchain-based provenance solutions, such as [8], do not track how the contract states evolved, for instance, which contract executions mutated these states. Therefore, they are not suitable for forensic analysis of Blockchain transactions impacted by contract defects or anomalies.

Understanding the change history of data has been extensively studied in the database systems community and is referred to as *data provenance*. In data provenance each row in the output of a single query (which is possibly comprised of sub queries) is annotated with the input tuples that derived it. Cheney et al. [11] introduced three types of data provenance for a specific output tuple in a query result: *Why-provenance* - the set of minimal input tuples that contributed to the output tuple; *How-provenance* - specifies how the output tuple was generated from the minimal input tuples; and *Where-provenance* - maps the specific output tuple's fields to the input tuples' fields. Glavic et al. [6] present a system that supports all three types of provenance by using query rewrites to annotate the output tuples with the corresponding provenance. While data provenance in general is mostly concerned with content, *workflow provenance* looks at the flow of execution and as such is derived from multiple components, each of which has its own configuration parameters which receives, processes, and forwards data from/to other components. Miao et al. [12] propose a system that collects workflow provenance in a collaborative workflows environment. Due to its potential, interest in Blockchain-based data provenance has been steadily growing. Researchers explored using Blockchain as a tamper-proof, fault-tolerant, distributive, and decentralized database for storing the provenance information itself [16] [17]. Recently, Ruan et al. [8] propose a system that enables Blockchain contracts to efficiently access the historic versions of the current contract state, which can increase the contracts' computation possibilities. Devecsery et al. [10] discuss software systems that remember all operations, function calls, and states at any time and refer to them as *Eidetic software systems*. This historic information, or *provenance*, facilitates analysis of meta data, e.g., it enables queries that can answer questions about what states or calls affected other states or calls, and conversely what states or calls were affected by other states or calls. Inspired by this, we propose an *Eidetic Blockchain system* that supports provenance for both data (Blockchain states) and control flow (contracts). Our system captures the provenance

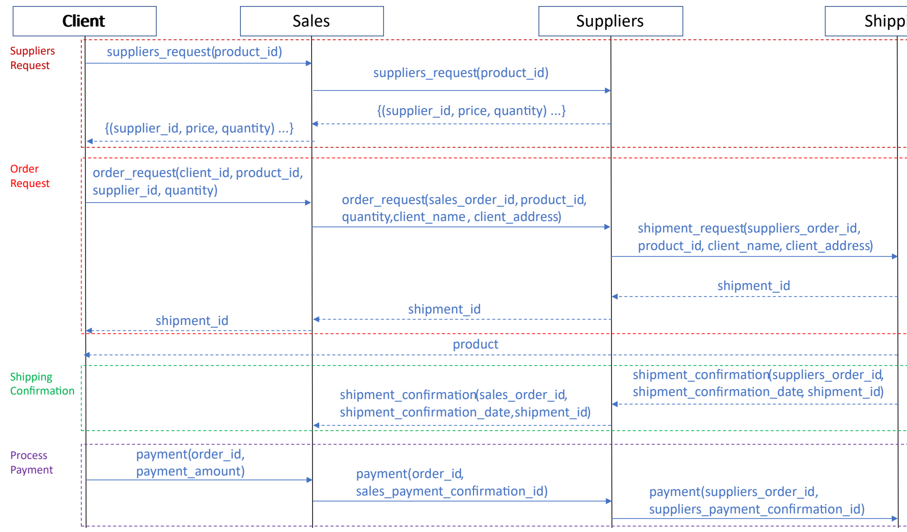

Fig. 1. Execution flow of a full order request session

of the contracts' execution flow (execution flow provenance), their parameters, and the relevant Blockchain states before and after each contract call (Why-provenance). This provenance information can then be used to query the execution flow across time in different granularity levels, facilitate a more complete understanding of the contracts' execution environment, and ultimately assist in better regulatory, quality, and maintenance management.

II. Smart Contract Execution Flow Provenance

Existing approaches to capturing data provenance on Blockchain [8] only gather information regarding the past history of Blockchain states. While, this is quite useful, for the purposes of finding bugs in the contract code, or analyzing malicious smart contacts, more detailed information is necessary regarding how these states where changed.

Accurately tracking information flow in the Blockchain, which includes its states along with the contract execution flow that operates on the data, can help identify the root cause of anomalies, debug defects in contract code, investigate malicious code, and enable efficient audit tracking for regulation purposes. To this end, in the next sections we provide a motivating example and then describe our proposed system.

A. Motivating Example

Alice, Bob, and Carol are friends and uPhone enthusiasts. Alice logs to an online retailer site to look for recent deals on the new phone model. She receives a list of suppliers that offer the phone alongside its price and quantity per supplier. Alice then picks the supplier that offers the best price and issues an order request. The sales department receives the order request, and issue an order request of their own to the suppliers department. The suppliers department issues a shipping request to the shipping department for the product from the specific supplier to Alice. The shipping department then issues a shipping identifier and returns it to the suppliers department that returns it to sales, that returns it to Alice. When the product is delivered, the shipping department sends the shipping confirmation to the suppliers, that send it to sales.

TABLE I
SUPPLIERS INITIAL DATA

supplier_id	product_id	product_quantity	product_price
1	100	3	135
2	100	2	110

Sales charge Alice using her recorded payment information and issue a payment confirmation for the suppliers, which issues a payment confirmation for shipping. The execution flow for Alice's example can be seen in Fig. 1. Alice then informs Bob about the supplier with the great deal she found. Bob logs to the site, issues a similar order request, and calls to update Carol, who logs to the site but can not find the deal. He calls the customer service to inquire why.

B. Implementation details

Each department's operational logic can be encapsulated by a smart contract, i.e. sales, suppliers, and shipping each has its own contract. Each department maintains its database state on the Blockchain. Each department issues a contract call using their corresponding client, e.g., the sales department uses a sales_client to call the sales contract. The human client that issues the order request uses its own client. A description of selected contract functions that participate in the order_request scenario can be seen in Table II.

Transactions that are related to a specific scenario share a unique scenario_id in addition to a session_id that identifies the client's session, which is comprised of different scenarios.

The initial suppliers database can be seen in Table I. In our example we use two user clients to issue an order request: Alice with client_id 1 is the first to issue the order request for product_id 100 from supplier_id 2, which has only 2 units of the product. Bob with client_id 2 follows with an order request of the same product from the same supplier and orders an additional unit.

III. The Proposed System

A. Collecting provenance data

Our proposed system collects contract execution flow and data provenance during run-time. The contract execution infrastructure is modified to include a non-intrusive provenance

TABLE II

CONTRACTS DESCRIPTION

Smart contract	API function	Description
sales	order_request	Records the order details in the client_orders_DB and calls the order_request function in the suppliers smart contract. The response should contain a shipment_id which is updated in the order details.
suppliers	order_request	Records the order details in the sales_orders_DB and calls the shipping_request function in the shipping smart contract. The response should contain a shipment_id which is updated in the order details and is returned to the calling smart contract.
shipping	shippping_request	The client physical address details are updated, a shipment_id is generated and is returned to the calling smart contract.

Fig. 2. A session's contracts call graph

collection mechanism that collects the contracts' execution information during run time and is comprised of:

- Execution parameters, such as the caller (user or contract) name/address, the callee (contract) name/address, and the API function and parameters to be invoked by the callee.
- The contracts call graph.
- The state before and after the caller calls the callee.

When a caller calls the callee contract only the states that are read or written by the caller and callee are collected. This is done by modifying the Blockchain context's get/set functions to record what addresses were read/written. The produced results provide a more concise view of the provenance and facilitates an easier analysis. The collected data are stored in a provenance database. A different process converts and exports the collected provenance data into the graph database such as neo4j [9] to enable visualizing and querying the provenance using the graph's query language.

Our system generates two types of provenance graphs: Contracts call graph - new nodes are generated and connected per contract call to reflect the provenance of the call graph. An example of this type of graph can be seen in Fig. 2. Contracts parameters and state change graph - for each contract call its parameters and the states before and after the call are captured and are connected to the caller node. An example of this type of graph can be seen in Fig. 3

B. Querying provenance data

The contracts call graph provides information on what contracts were called during each session and scenario and by who (client/contract). Fig. 2 shows the contracts call graph of Alice's session. From this we can see whether all contracts were called in the correct sequence. Each node contains an entity id and name, the session id, the scenario id, and the contract API function that was invoked on the callee contract. To get more detailed information on each entity call, further drill down is possible for each node by querying the contracts parameters and state change graph. Further, this graph can be used to answer queries regarding changes in different granularity levels. Following are query examples for different granularity levels:

a) Querying changes across multiple sessions: Returning to the motivating example, Alice and Carol both ordered the new uPhone from supplier id 2. Carol could not find the same supplier and called the customer service to inquire about it. A customer service representative queries the suppliers database and sees that the supplier with id 2 has no more products. To reveal the history that led to this current state, the representative can issue a query that show what order requests that involved supplier with id 2 were issued and their details. The result can be seen in table III. The table shows that customers with id 1 and 2 issued an order_request on the specific date and time for a product from supplier_id 2. The first line shows that after Alice issued the request, the product quantity changed into 1, and the second line shows that after Bob issued the request the product quantity changed into 0.

b) Querying changes in a session across multiple scenarios: Lets assume that Alice would like to know when was her account charged for the item. Table IV shows the query results for Alice's specific session in the sales department's client_orders_DB. The first line shows that after the order_request the sales department created a client order row with id 111. Since the shipping id is provided at the end of the order_request call it can be seen in the last column. The second line shows that the payment_confirmation_id was updated at the end of the process_payment call and its datetime.

c) Querying changes in a specific scenario and entity: Fig. 3 shows a drill down query result for the suppliers node in the order_request scenario (marked by a red square as in Fig. 2), where the suppliers department requests shipping for

TABLE III
SUPPLIER STATE CHANGE ACROSS MULTIPLE ORDER REQUEST SCENARIOS

Contract call parameters				API function parameters				Suppliers db row after contract call		
request datetime	API function	from	to	supplier id	client id	product id	product quantity	supplier id	product id	product quantity
2020-01-10 2:02:34	order_request	client	sales	2	1	100	1	2	100	1
2020-01-10 3:16:15	order_request	client	sales	2	2	100	1	2	100	0

TABLE IV
CLIENT ORDER STATE CHANGES ACROSS SESSION

Contract call parameters				API function parameters			Client orders db row after contract call		
request datetime	API function	from	to	sales order id	supplier id	payment confirmation id	sales order id	payment confirmation id	shipment id
2020-01-10 2:02:34	order_request	sales	suppliers	111	2		111		555
2020-01-13 1:13:11	process_payment	sales	suppliers	111		123	111	123	555

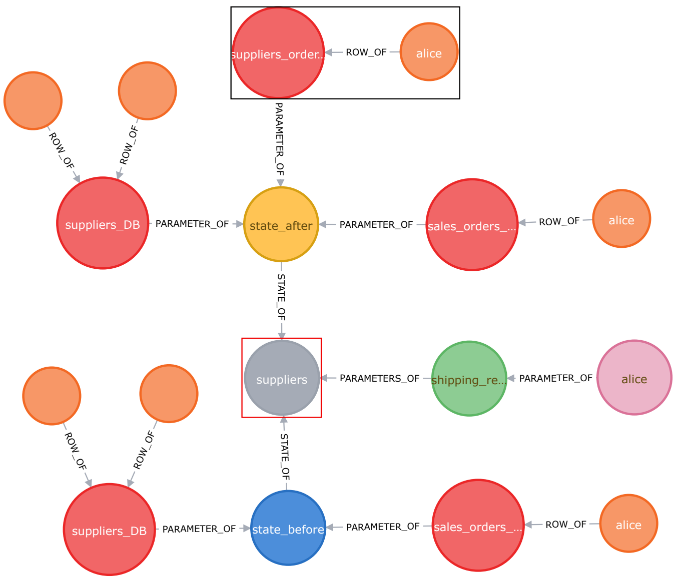

Fig. 3. Suppliers contract calls Shipping_request on Shipping contract

the product ordered by Alice. The middle node represents the suppliers department entity. The attached green node represents the parameters used in the contract call such as the contract and the API function names, and the request datetime; the attached pink node contains the function specific parameters such as Alice's name and address, and the suppliers order id; the blue node represents the relevant Blockchain state before the shipping contract call; and the yellow node the relevant Blockchain state after the call. As can bee seen, the top most nodes (marked by a black rectangle) are present only in the state after the call and represent the suppliers' shipping order for Alice that was recorded by the shipping contract.

IV. Conclusion

The adoption of smart contract applications in managing valuable assets rapidly increases alongside the applications' size and complexity. As a result coding errors, exploit potential, and regulation requirements with impactful financial repercussions increase as well. In order to facilitate root-cause analysis of anomalies and investigate defects or malicious activities in the smart contract, the Blockchain system needs to efficiently manage both data and execution flow. In this paper we propose a Blockchain provenance collection and analysis system and explore how the system can non-intrusively capture relevant provenance information of the contracts' execution flow, their parameters, and the Blockchain states before and after each contract call. We further explore how this information can be queried at different levels of granularity levels to facilitate better regulatory, quality, and maintenance management.

References

[1] https://ethereum.org/
[2] https://www.hyperledger.org/
[3] Blockchain Applications For The Modern Nation, https://cryptodaily.co.uk/2018/03/blockchain-applications/
[4] Blockchain has grabbed the attention of investors, https://www.cnbc.com/2018/04/02/blockchain-has-grabbed-the-attention-of-investors.html
[5] Three Near-term Applications For Blockchain Technology, https://www.forbes.com/sites/forbesfinancecouncil/2018/03/28/three-near-term-applications-for-blockchain-technology/2/#6c9f49c6310d
[6] Glavic, B. and Alonso, G., 2009, March. Perm: Processing provenance and data on the same data model through query rewriting. In 2009 IEEE 25th International Conference on Data Engineering (pp. 174-185). IEEE.
[7] Psallidas, F. and Wu, E., 2018. Smoke: Fine-grained lineage at interactive speed. Proceedings of the VLDB Endowment, 11(6), pp.719-732.
[8] Ruan, P., Chen, G., Dinh, T.T.A., Lin, Q., Ooi, B.C. and Zhang, M., 2019. Fine-grained, secure and efficient data provenance on blockchain systems. Proceedings of the VLDB Endowment, 12(9), pp.975-988.
[9] https://neo4j.com/
[10] Devecsery, D., Chow, M., Dou, X., Flinn, J. and Chen, P.M., 2014. Eidetic systems. In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14) (pp. 525-540).
[11] Cheney, J., Chiticariu, L. and Tan, W.C., 2009. Provenance in databases: Why, how, and where. Foundations and Trends® in Databases, 1(4), pp.379-474.
[12] Miao, H., Chavan, A. and Deshpande, A., 2017, May. Provdb: Lifecycle management of collaborative analysis workflows. In Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics (p. 7). ACM.
[13] https://www.statista.com/statistics/1015362/worldwide-blockchain-technology-market-size/
[14] ETHNews: Hkg token has a bug and needs to be reissued (2017). ethnews.com/ethercamps-hkg-token-has-a-bug-and-needs-to-be-reissued
[15] Simonite, T.: $80 million hack shows the dangers of programmable money, June 2016. https://www.technologyreview.com
[16] Liang, X., Shetty, S., Tosh, D., Kamhoua, C., Kwiat, K. and Njilla, L., 2017, May. Provchain: A blockchain-based data provenance architecture in cloud environment with enhanced privacy and availability. In Proceedings of the 17th IEEE/ACM international symposium on cluster, cloud and grid computing (pp. 468-477). IEEE Press.
[17] Ramachandran, A. and Kantarcioglu, M., 2018, March. SmartProvenance: a distributed, blockchain based dataprovenance system. In Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy (pp. 35-42). ACM.