

Transactional Smart Contracts in Blockchain Systems

Victor Zakhary
UC Santa Barbara
Santa Barbara, California
USA, 93106
victorzakhary@ucsb.edu

Divyakant Agrawal
UC Santa Barbara
Santa Barbara, California
USA, 93106
divyagrawal@ucsb.edu

Amr El Abbadi
UC Santa Barbara
Santa Barbara, California
USA, 93106
elabbadi@ucsb.edu

ABSTRACT

This paper presents TXSC, a framework that provides smart contract developers with transaction primitives. These primitives allow developers to write smart contracts without the need to reason about the anomalies that can arise due to concurrent smart contract function executions.

1. INTRODUCTION

Executing concurrent operations has been a long-term challenge in the design of large software systems. Without careful usage of synchronization primitives [8], the concurrent execution of multiple procedures that access shared variables can easily result in anomalous executions. Instead of using synchronization primitives, that a programmer must carefully program, database systems introduced the elegant declarative notion of *transactions* [9]. Programs that may be executed concurrently are each executed as a transaction, and the database management system ensures that transaction execution is isolated from each other and that the concurrent and interleaved execution of multiple transactions is serializable, i.e., equivalent to a serial execution [4].

Recent interest in blockchains has resulted in its rapid usage in diverse applications, and its evolution to support complex concurrent executions. The original blockchain, as proposed in Bitcoin [17], involved simple *transactions*, that transfer some bitcoins from one end-user (typically Alice) to another end-user (typically Bob). The original bitcoin blockchain can be easily modelled as an abstract data type representing a linked list of blocks of transactions. The accessed data is the cryptocurrency, bitcoins, and transactions transfer part of the remaining, unused assets of Alice to Bob, while keeping the rest with Alice (hence the term Unspent Transaction Output, UTXO to refer to the assets belonging to a client in Bitcoin). A *miner* adds a transaction to a block if the assets consumed in the transaction are not double spent in the same block and if the miner can validate that the end-user does actually have these assets, i.e., the UTXO actually belongs to the end-user issuing the transaction. Finally, a miner adds a block to the blockchain if it solves the Proof of Work (PoW) puzzle [17].

Ethereum [22] reintroduced the notion of *smart contracts* [21] to blockchains. Smart contracts extend the simple abstract data type notion of blockchain transactions to include complex data type classes with end-user defined variables and functions. When an end-user deploys a smart contract in a blockchain, this deployment results in instantiating an object instance of the smart contract class in the blockchain [7, 11]. The object state is initially stored in the block

where the object is instantiated. End-users can issue a smart contract function call by sending function call requests to the miners of a blockchain. These function calls are transactions that are sent to the *address* of the smart contract object. Miners execute these transactions and record object state changes in their currently mined block. Therefore, the state of a smart contract object could span one or more blocks of a blockchain.

Smart contracts now have their own variables and multiple functions that may be executed by different end-users results in transactions which might be incorporated in different blocks by different miners. This clearly results in complex concurrency challenges which need to be handled by smart contract developers. Distributed database literature [6, 20] has shown that putting the burden of implementing transaction logic in the application layer is problematic. This is no simple task and serious smart contract concurrency bugs have been highlighted in the blockchain literature [7, 13, 15, 19]. In fact, from a financial point-of-view, two such famous anomalies in the context of blockchains, TheDAO [1, 5] and the BlockKing [2] have resulted in a loss of tens of millions of investors' dollars [15].

In this paper, we advocate leveraging the traditional transactional approach to address the concurrency violations in the context of smart contract executions in large scale blockchain systems. In particular, we propose Transactional Smart Contracts (TXSC) as a framework that allows developers to write smart contracts with correct transaction isolation semantics. Unlike previous works [13, 15, 19] that propose smart contract analysis tools to detect concurrency bugs in smart contracts, TXSC aims to free smart contract developers from the burden of implementing correct concurrency control semantics for each smart contract. Instead, developers can focus on the smart contract application semantics and leave the concurrency semantics to TXSC.

Concurrency control problems arise in two general contexts during smart contract function execution depending on whether the application semantic functionality is implemented by a single or multiple functions. In a **single function**, each function in a smart contract is executed correctly (and in isolation) as a miner validates its execution. However, the state of the data in the blockchain is visible and can be read all the time by any end-user. An end-user might take action based on a value read, but due to the concurrent execution of smart contract functions, such a read value might be stale when the function is executed. TXSC needs to ensure that the attribute values observed by an end-user, where these attributes are in the read set of a function,

are still valid when the function is executed. Alternatively, the semantic functionality might be executed by **multiple functions** in the same or even different smart contracts on potentially different blockchains. These functions might invoke each other in an asynchronous manner. In particular, a function, before termination may call another function to perform a specific task, which in turn calls a third function, and so on. This arises due to smart contracts in a single blockchain like the puzzle example in [15] or across multiple chains [2, 5] that requires atomic execution across blockchains [10, 18, 24]. In this case, different invocations of the function might be interleaved resulting in incorrect executions due to the lack of isolation.

In this paper, we propose the Transactional Smart Contracts paradigm to solve these concurrency problems. In particular,

1. This paper models smart contract concurrency anomalies as transaction isolation problems. Examples illustrate how different smart contract concurrency anomalies can be mapped to the problem of transaction isolation of either single domain or distributed cross-domain transactions.
2. TXSC is the first framework to provide smart contract developers with transactional primitives *start transaction* and *end transaction*. TXSC takes a smart contract that contains these primitives as an input and translates it to a transactionally correct smart contract using the smart contract native language.

The rest of the paper is organized as follows. We start with two examples to illustrate the types of concurrency anomalies that can arise in the context of smart contracts in Section 2. Data and transaction models are presented in Section 3. Section 4 explains our solution and presents TXSC and the paper is concluded in Section 5.

2. CONCURRENCY ANOMALIES IN SMART CONTRACTS

Most of the smart contract anomalies identified in prior work [7, 13, 15, 19] are rooted to faulty transaction isolation semantics implemented by the smart contract developers. These anomalies can be classified into *two* categories: 1) faulty transaction isolation semantics among transactions that span a single administrative domain (or one blockchain) and 2) faulty transaction isolation semantics among distributed transactions that span several administrative domains (more than one blockchain or one blockchain and services outside the domain of this blockchain). We explain the two categories using the following two examples from [15] and [19]. For consistency with the original blockchain terminology, in this section, we refer to a function call request as a transaction (later we will change this).

The puzzle example. This example illustrates the first category of smart contract concurrency anomalies. In this example, an end-user, *the challenger*, deploys a smart contract that pays another end-user, *the solver*, a reward if the solver’s submitted puzzle solution is correct. Algorithm 1 shows the puzzle smart contract pseudocode. As shown, the smart contract has three functions: a **Constructor** (Line 6), **UpdateReward** (Line 12), and **SubmitSolution** (Line 19) functions. The **Constructor** is executed by the contract owner,

the challenger, to initialize the smart contract object. **UpdateReward** can be executed only by the challenger to update the reward value of the puzzle. Furthermore, **UpdateReward** can only be executed if the puzzle has not been solved yet (Line 14) and **UpdateReward** sends the old reward value to the challenger and updates the reward value with the new value sent by the challenger (Line 16). **SubmitSolution** (Line 19) allows any solver to submit a solution to the puzzle only if the puzzle has not been solved yet. If the submitted solution is correct (Line 21), the reward goes to the solver, the puzzle’s solution is updated, and the puzzle is marked as solved.

Now, assume Alice is a challenger who posts a puzzle that follows the smart contract description in Algorithm 1 in the Ethereum network and she sets the reward value r to $r = 2$ ethers, the currency of the Ethereum network. Bob, a solver, reads the reward value $r = 2$ ethers, solves the puzzles, and submits the solution to the smart contract through a transaction TX_1 . Bob assumes to receive a puzzle reward of 2 ethers if his solution is correct. Concurrently, Alice might, benignly or maliciously, schedule a transaction TX_2 that updates the reward of the puzzle to a smaller value than the current reward e.g., $r = 0$. If TX_2 is executed first, r would be updated to its new value 0. While updating the reward value should result in aborting TX_1 as the value of r read by TX_1 is stale, the smart contract code in Algorithm 1 would allow TX_1 to execute. This results in Alice receiving a solution to her puzzle while Bob gets a reward of 0 ethers. As both TX_1 and TX_2 access an object that spans only one blockchain, the Ethereum network, this concurrency anomaly falls into the first category of the two aforementioned categories.

The BlockKing [2, 19] example. This example demonstrates the second category of smart contract concurrency anomalies where end-user distributed transactions span several administrative domains (objects of one or more blockchains in addition to asynchronous calls to external services). Algorithm 2 shows code snippets from the original 366 lines of code of the BlockKing smart contract [2] where concurrency anomalies occur. The BlockKing smart contract works as follows. At any moment in time, there exists one block king, initially, the contract owner. Users send money to the contract via the **Enter** function (Line 4) as bids to become the next block king. The **Enter** function stores the address of the caller, the current block number, and the caller’s bid value in the attributes **warrior**, **warriorBlock**, and **warriorGold** respectively. Then, the **Enter** function calls an external random number generator to generate a random number between 1-9 and if the returned number equals to the first digit of the block number stored in the **warriorBlock** attribute, the caller of the **Enter** function becomes the new block king. A block king gets a percentage of the bid money of every call to the **Enter** function and the contract owner gets the remaining percentage of this bid money. Notice that the random number generator triggers an asynchronous callback function (Line 10) where the returned random number is checked against the block number in the **warriorBlock** attribute. If the returned random number matches the first digit of the block number in the **warriorBlock**, the current warrior becomes the new block king.

If calls to the **Enter** function are blocking; meaning that at most one call to the **Enter** function is allowed until its call-

Algorithm 1 Puzzle smart contract example in [15]

```
class Puzzle {
1: address public owner          ▷ contract owner
2: bool public solved    ▷ true if the puzzle is solved
3: uint public reward    ▷ puzzle solving reward
4: bytes32 public diff    ▷ puzzle difficulty
5: byte32 public solution ▷ puzzle solution if found
6: procedure CONSTRUCTOR
7:   this.owner = msg.sender
8:   this.reward = msg.value
9:   this.solved = false
10:  this.diff = bytes32(msg.data)    ▷ set difficulty
11: end procedure
12: procedure UPDATEREWARD
13:   requires(msg.sender == this.owner)
14:   if ! solved then
15:     transfer reward to owner
16:     reward = msg.value
17:   end if
18: end procedure
19: procedure SUBMITSOLUTION
20:   if ! solved then
21:     if sha256(msg.data) < diff then
22:       transfer reward to msg.sender
23:       solution = msg.data
24:       solved = true
25:     end if
26:   end if
27: end procedure
}
```

back is completed, the smart contract in Algorithm 2 would not have any concurrency anomalies. However, the smart contract in Algorithm 2 is non-blocking. This non-blocking behavior allows many concurrent calls to the `Enter` function to take place. If multiple transactions are concurrently sent to the `Enter` function, each transaction would replace the values of the `warrior`, the `warriorBlock`, and the `warriorGold` attributes of all the previous incomplete transactions. This leads to an advantage to the latest caller who sends a transactions to the `Enter` function before all previous callbacks occur. Every trigger to the callback function gives the latest caller a chance to become the new block king while previous callers have no chance to become the new block king. We illustrate this transaction isolation anomaly using the following example. Assume Alice, Bob, and Carol concurrently want to become the next block king. They send three transactions (corresponding to three `Enter` function calls) TX_1 , TX_2 , and TX_3 accompanied by their bids to the `enter` function respectively. TX_1 updates the warrior attributes to Alice's attributes sent along with TX_1 then, calls the external random number generator. Before TX_1 's callback is triggered, TX_2 replaces the warrior attributes with Bob's attributes sent with TX_2 and similarly, TX_3 replaces the warrior attributes with Carol's attributes. When the callbacks of TX_1 , TX_2 , and TX_3 are triggered, which possibly could take place in another block in the BlockKing blockchain, the three callbacks use the warrior attribute values of Carol to decide if she could be the next block king or not. Carol gets 3 chances to become the block king while Alice and Bob have no chance.

Algorithm 2 Snippets from the BlockKing contract [2]

```
class BlockKing {
1: address public king, warrior
2: uint public kingBlock, warriorBlock
3: uint public warriorGold, randomNumber
4: procedure ENTER
5:   ...          ▷ check if minimum bet is sent
6:   warrior = msg.sender, warriorGold = msg.value
7:   warriorBlock = block.number
8:   byte32 myid = oraclize_query(0, "WolframAlpha",
   "random number between 1 and 9")
9: end procedure
10: procedure _CALLBACK(byte32 myid, string result)
11:   requires(msg.sender == oraclize_cbAddress())
12:   randomNumber = uint(bytes(result)[0]) - 48;
13:   if singleDigitBlock == randomNumber then
14:     ...          ▷ update reward
15:     king = warrior, kingBlock = warriorBlock
16:   end if
17: end procedure
}
```

Transactions in the first category can be atomically executed in one shot within one block of its smart contract blockchain. On the other hand, distributed transactions could span multiple blocks in one or more blockchains and hence ensuring their atomicity while executing them in isolation is significantly more complicated than executing transactions in the first category in isolation.

3. DATA AND TRANSACTION MODELS

An open permissionless blockchain [16] comprises an application layer and a storage layer. Clients in the application layer have public identities represented by their public keys and private signatures generated using their private keys. Clients send signed transactions to the storage layer in order to transfer assets from one client to another. The storage layer consists of mining or computing nodes, *miners*, and each miner manages a copy of the blockchain. Transactions, in the storage layer, are grouped into blocks and each block is hash chained to the previous block; hence the name *blockchain*. When a mining node receives a transaction, it verifies the transaction and adds it to its current block, *only if the transaction is valid*. Mining nodes run a consensus algorithm or in a permissionless blockchain Proof of Work (PoW) to reach consensus on the next block to be added to the blockchain.

Smart contracts are analogous to classes [7, 11, 25] in Object Oriented Programming Languages (OOP) and are used by clients to implement complex data types. Clients deploy smart contracts to a blockchain by sending a deployment message to miners of this blockchain. As a result, a miner instantiates an object of the smart contract class and stores this object in the current block in the blockchain. Smart contract objects have attributes that capture their state. Once a smart contract object is instantiated in a blockchain, the state of this object, as part of the blockchain, is made public and can be **externally read by any client at any moment**. In addition, smart contract objects have functions that define the possible state transitions of these

objects. Since an object state is public, smart contract read-only functions are pointless. Therefore, it is safe to assume that any smart contract function call has to update at least one attribute of the smart contract object [23]. A smart contract object has an address in the blockchain. When a client wants to issue a smart contract function call, the client sends a function call request to the miners of the blockchain where the smart contract is deployed. This function call request is directed to the address of the smart contract object. Miners use the smart contract address to locate the smart contract object (state and code). This function call is accompanied by some implicit parameters like *msg.sender*, the address of the client who sent the transaction, *msg.val*, the value of the money sent along with the transaction, and *msg.data*, any data that needs to be sent along with the transaction. In addition, function calls could be accompanied by some function explicit parameters.

We follow the Ethereum [22] smart contract execution model. Each function call is accompanied by some *gas* value. The *gas* value represents the amount of money a client is willing to pay to incentivize miners to execute the function call. Miners charge some *gas* for every executed line of code in the called function. A miner stores any intermediate results of a function call in their local storage. If the function call completes before the function call runs out of *gas*, the intermediate results are finalized and included in the miner's current block. However, if a function call runs out of *gas* before the function call is completed, intermediate results are deleted and the smart contract object state does not change. Either way, the miner includes a transaction that pays the miner the amount of *gas* spent during the execution of the function call in its current block. Smart contract function calls are atomic meaning that each function call either terminates after it successfully updates the object state in the blockchain or rolls back to the object state before the call occurs. Concurrent function calls are sequentially executed one after the other without any interruption [19]. In blockchain terminology, a function call request is usually referred to as a transaction. Yet, a function call might not ensure the ACID [4] properties of transactions in traditional databases.

In traditional DBMS, a client transaction starts when a client calls the *start (begin) transaction* command. Afterwards, a transaction reads and updates some data values followed by an *end (commit) transaction* command. The role of the DBMS is to ensure the ACID properties of a client transaction from the moment the transaction begins till the moment the transaction ends (whether the transaction commits or aborts).

In permissionless blockchains, miners have no way to learn the details of all client activities before calling the smart contract functions, e.g., when the client activities start and what values were read before a function call request is sent to the miners. Even when each function call is executed in isolation from concurrent function calls, transaction isolation concurrency violation still occur as shown in Algorithms 1 and 2 as a result of poor client transaction isolation, network asynchrony, and smart contract asynchronous callbacks. We consider a *client transaction span* to include all the read operations that took place before the client sends a function call, the function execution caused by the function call, and any callbacks that are triggered as a result of this function call. The goal of this paper is to ensure the ACID properties of client transactions from the time a client starts a trans-

action till the end of the function call that terminates this transaction.

4. TRANSACTIONAL SMART CONTRACTS

Algorithm 3 A smart contract example that uses TXSC

```
class SmartContract
1: procedure F1
2:   start transaction
3:   f1's logic
4:   end transaction
5: end procedure
6: procedure F2
7:   start transaction
8:   f2's logic
9:   end transaction
10: end procedure
```

This section presents TXSC, a framework that allows smart contract developers to write smart contracts with correct client transaction isolation semantics. The goal of TXSC is to provide developers with the primitives *start transaction* and *end transaction*. We call each function surrounded by these primitives, a **transactional** function. TXSC ensures that calls to transactional functions are executed in isolation from any concurrent function calls to the same function or any other function in the smart contract even in the presence of network asynchrony. Algorithm 3 illustrates an example smart contract written using TXSC. This smart contract has two functions F1 and F2 and both functions are transactional functions.

The ACID execution of a client transaction requires *atomic*, *consistent*, *isolated*, and *durable* execution of this client transaction. If the semantics of every smart contract function is correct, function calls should transfer the smart contract object from one consist state to another. Therefore, *consistency* is the responsibility of the smart contract developer. *Durability* of a function call is guaranteed through the blockchain protocol. Function calls that complete execution and are included in a mined block are durable assuming this block gets enough confirmations [3]. Since confirmed blocks are replicated to most of the mining nodes, these blocks are durable even in the presence of failures of many mining nodes. This leaves the responsibility of ensuring atomicity and isolation of client transactions on TXSC.

Isolation: Since smart contract developers have no way to detect which attribute values have been read by the client before a function call request is sent to miners, a smart contract developer has to insert checks at the beginning of every smart contract function call (similar to optimistic concurrency control [14]) to ensure that any data attribute value read by the client and is in the read set of the function call matches its current value in the blockchain. The read set of a smart contract function is the set of attributes that a function reads during its execution. We assume that the outcome of each function is **invariant** to any attribute outside the read set of this function. To ensure serializability [4] of client transactions, the client has to send her observed attribute values of the read set of the function along with the function call. The smart contract has to ensure that the received attribute values are up-to-data and they match the current values of all the attributes in the function read set before

executing the function call. Otherwise, the function call has to abort. A function call and all its asynchronous callbacks must be executed in isolation from concurrent function calls and callbacks.

Atomicity: The smart contract code has to guarantee that a function call and all its asynchronous callbacks are atomic. This means that updates that result from a function call and all its asynchronous callbacks should either all take place or none of them do.

TXSC automatically adds transaction isolation checks at the beginning of every transactional function to ensure an isolated execution of every call to any transactional function. TXSC handles the atomicity of single domain transactional functions differently from cross-domain distributed transactional functions as follows.

4.1 Single Domain Transactional Functions

A Single Domain Transactional Function (SDTF for short) is a function that reads and updates one or more smart contract objects stored under a single administrative domain or a single blockchain. SDTFs do not access external services or objects outside the domain of their blockchain. As a result, SDTF calls do not trigger any asynchronous callbacks. Any transactional function that accesses external services, blockchains, or trigger callbacks is classified as cross-domain distributed transactional function.

Since all the objects accessed by SDTF calls are stored in a miner’s copy of the blockchain and since SDTFs do not trigger asynchronous callbacks, a SDTF call can atomically be executed in one shot. Therefore, the atomicity of a client transaction that calls a SDTF is guaranteed by the smart contract execution model. To ensure a serializable execution of a SDTF, the function code has to only ensure the freshness of the read set of this function. TXSC scans every SDTF in a smart contract to determine the object’s attributes in the read set of this SDTF. Then, TXSC adds checks at the beginning of the SDTF to ensure that the attribute values observed by the client at the time when the transaction started are equivalent to attribute values when the function call is received by miners.

Recall the puzzle example in Algorithm 1. Both `UpdateReward` and `SubmitSolution` are single domain function calls. To convert `UpdateReward` to a SDTF, TXSC adds a requirement that every function call to the `UpdateReward` function must be accompanied by the client observed value of the attribute `solved`, in the function read set, in its implicit parameter `msg.data`. Then, TXSC adds a requirement check `solved == msg.data.solved`. If the `solved` attribute value in a client’s `UpdateReward` function call is stale, the call must abort and the smart contract object state remains unchanged. However, if the `solved` attribute is up-to-date and the function call is also accompanied by sufficient `gas`, the call can be atomically executed and as a result, the reward value is updated.

For the `SubmitSolution` function call, TXSC adds the requirement checks `solved == msg.data.solved` and `reward == msg.data.reward`. Recall the concurrency violation of the puzzle smart contract in Section 2. When Bob sends his solution to the `SubmitSolution` function, Bob would send the attribute values `solved = false` and `reward = 2 ethers` in the `msg.data` parameter of his function call. When Bob’s request is received by a miner, there are two possible outcomes: 1) the function call gets executed only if the current

reward value equals to 2 ethers and the puzzle is not solved and 2) the function call aborts if the reward value has been updated in between the time when Bob’s transaction started and the time when his function call is received by a miner. Both outcomes do not violate the serializability guarantee.

4.2 Cross-Domain Transactional Functions

A Cross-Domain Distributed Transactional Function (CDTF for short) is a function that reads and updates one or more smart contract objects stored under multiple administrative domains or multiple blockchains. In addition, CDTFs can access external services or objects outside the domain of their blockchain. Also, CDTFs may trigger asynchronous callbacks. As a result, updates made by a CDTF can span more than one block of the blockchain. Recall the `BlockKing` smart contract in Algorithm 2. Each function call first updates the `warrior`, `warriorBlock`, and `warriorGold` in some block and might update the `BlockKing` in another block when the callback function is trigger. Allowing a CDTF call to update the state of a smart contract object in several blockchain blocks is problematic. If the updates in the first block gets committed in a mined block, committed updates cannot be rolled back even if updates in the following blocks fail due to an exception or that the call runs out of gas. We first explain isolation and atomicity challenges of CDTFs. Afterwards, we explain how TXSC handles CDTFs.

Isolation: A CDTF has an entry point that comprises a function call in an object in some blockchain. This function call might trigger other function calls of objects stored in different blockchains. Since a CDTF can span multiple blockchain objects, sending the read set of all the accessed objects at the entry point (the first function call) is not sufficient to guarantee transaction isolation as in SDFT. Since all subsequent function calls to other objects are trigger over an asynchronous network, the state of these subsequent objects might change in the time between the entry point and the point when the subsequent call is received by the miners of the blockchain where these objects are stored. Even if the read set is carried on with every subsequent call, a stale attribute in the read set might result in aborting a subsequent call. However, the first call might have been committed leading to a violation to atomicity.

Atomicity: Guaranteeing the atomicity of CDTF calls is significantly more complicated than SDFTs. First, atomicity could be violated if one of the subsequent calls to functions in other blockchains runs out of `gas`. Second, if an external service (e.g., the random number generator in the `BlockKing` example) crashes for a long time or if the message from this external services that triggers the callback function is lost, atomicity can be violated resulting in an inconsistent state (some updates occur in one block but the callback is never triggered to complete the execution of the function call).

Due to space limitation, we only present a high level solution that guarantees both the isolation and atomicity of CDTFs. The atomic and isolated execution of a CDTF that spans multiple blockchains can be mapped to the problem of atomic cross-chain transaction processing. Atomic cross-chain commitment protocols have been introduced in [10, 12, 18, 24]. First, the solution requires to lock all the object attributes in both the read set and the write set of *all* the functions in a CDTF before calling the entry point. This locking guarantees the isolation of a CDTF from all

concurrent function calls to any of the functions that can update either the read set or the write set of a CDTF. However, as shown in [24], using timelocks as proposed in [10,18] can lead to atomicity violations. The AC^3WN [24] and the CBC [12] protocols that use an additional blockchain as a lock manager are possible solutions to manage the locking of object attributes across blockchains. After all the object attributes are locked, a caller can send a function call request to the entry point accompanied by evidence that all the object attributes in both the read set and the write set of this function call and all subsequent function calls are locked. Object attributes are unlocked only when the function call that accesses them and its corresponding callbacks, if any, terminate. Recall the BlockKing concurrency anomaly in Section 2. Alice’s call locks the accessed attributes before calling the `Enter` function. This prevents other callers, Bob and Carol, from issuing concurrent function calls to the `Enter` function. Second, economic incentives should be used to enforce callers to accompany function calls with enough *gas*. At the entry point, a caller locks some money in the contract that gets refunded to the caller only if all her function calls terminate. If any function call runs out of gas, the caller loses her locked money to the contract owner who can complete the call and gets the locked objects unlocked. Finally, redo logs can be used to overcome the atomicity violations in the presence of external service crashes. In the BlockKing example, the smart contract object should have an “after-image” attribute corresponding to every attribute in the object. The `Enter` function should update the after-images of `warrior`, `warriorBlock`, and `warriorGold` attributes. When the `callback` is triggered, only then, the after-image attributes can be copied to the actual attributes of the object. This guarantees that even if the external service crashes or the callback trigger is lost, the object is in consistent state.

5. CONCLUSION

In this paper, we presented TXSC, a framework that allows developers to write smart contracts with correct transactional semantics. We showed that TXSC can help developers solve isolation anomalies of both single domain and cross-domain distributed transactional functions.

6. REFERENCES

- [1] The dao (organization). [https://en.wikipedia.org/wiki/The_DAO_\(organization\)](https://en.wikipedia.org/wiki/The_DAO_(organization)).
- [2] Blockking contract. <https://etherscan.io/address/0x3ad14db4e5a658d8d20f8836deabe9d5286f79e1>, 2016.
- [3] Bitcoin confirmations. <https://www.buybitcoinworldwide.com/confirmations/>, 2018.
- [4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency control and recovery in database systems. 1987.
- [5] V. Buterin. Critical update re: Dao vulnerability. <https://ethereum.github.io/blog/2016/06/17/critical-update-re-dao-vulnerability/>, 2016.
- [6] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google’s globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.
- [7] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 303–312. ACM, 2017.
- [8] E. W. Dijkstra. Cooperating sequential processes. In *The origin of concurrent programming*, pages 65–138. Springer, 1968.
- [9] J. Gray et al. The transaction concept: Virtues and limitations. In *VLDB*, volume 81, pages 144–154. Citeseer, 1981.
- [10] M. Herlihy. Atomic cross-chain swaps. *arXiv preprint arXiv:1801.09515*, 2018.
- [11] M. Herlihy. Blockchains from a distributed computing perspective. *Communications of the ACM*, 62(2):78–85, 2019.
- [12] M. Herlihy, B. Liskov, and L. Shrira. Cross-chain deals and adversarial commerce. *arXiv preprint arXiv:1905.09743*, 2019.
- [13] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena. Exploiting the laws of order in smart contracts. *arXiv preprint arXiv:1810.11605*, 2018.
- [14] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.
- [15] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269. ACM, 2016.
- [16] S. Maiyya, V. Zakhary, D. Agrawal, and A. E. Abbadi. Database and distributed computing fundamentals for scalable, fault-tolerant, and consistent maintenance of blockchains. *Proceedings of the VLDB Endowment*, 11(12):2098–2101, 2018.
- [17] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.
- [18] T. Nolan. Alt chains and atomic transfers. <https://bitcointalk.org/index.php?topic=193281.msg2224949#msg2224949>, 2013.
- [19] I. Sergey and A. Hobor. A concurrent perspective on smart contracts. In *International Conference on Financial Cryptography and Data Security*, pages 478–493. Springer, 2017.
- [20] J. Shute, M. Oancea, S. Ellner, B. Handy, E. Rollins, B. Samwel, R. Vingralek, C. Whipkey, X. Chen, B. Jegerlehner, et al. F1-the fault-tolerant distributed rdbms supporting google’s ad business. 2012.
- [21] N. Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.
- [22] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.
- [23] K. Wüst, S. Matetic, S. Egli, K. Kostianen, and S. Capkun. Ace: Asynchronous and concurrent execution of complex smart contracts.
- [24] V. Zakhary, D. Agrawal, and A. E. Abbadi. Atomic commitment across blockchains. *arXiv preprint arXiv:1905.02847*, 2019.
- [25] V. Zakhary, M. J. Amiri, S. Maiyya, D. Agrawal, and A. E. Abbadi. Towards global asset management in blockchain systems. *arXiv preprint arXiv:1905.09359*, 2019.