Emerging Design Patterns for Blockchain Applications

Vijay Rajasekar[1], Shiv Sondhi[1], Sherif Saad[1] and Shady Mohammed[2]
[1]*School of Computer Science, University of Windsor, Canada*
[2]*Faculty of Computer Science & IT, Ahram Canadian University, Egypt*

Keywords: Software Engineering, Software Patterns, Empirical Analysis, Anti-patterns, Blockchain.

Abstract: Blockchain or Distributed Ledger Technology (DLT) introduces a new computing paradigm that is viewed by experts as a disruptive and revolutionary technology. While bitcoin is the most well-known successful application of blockchain technology, many other applications and sectors could successfully utilize the power of blockchain. The potential applications of blockchain beyond finance and banking encouraged many organizations to integrate and adopt blockchain into existing or new software systems. Integrating and using any new computing paradigm is expected to affect the best practice and design principles of building software systems. This paper summarizes our ongoing research on collecting, categorizing and understanding, existing software design patterns when building blockchain-based software systems. It collects and categorizes the existing software (design and architectural) patterns that are commonly linked to blockchain and distributed ledger technology. We provide an informal analysis of the identified patterns to highlight their maturity. Finally, we discuss the current research gap in software engineering for blockchain-based applications and propose potential research directions.

1 INTRODUCTION

A blockchain is a distributed store or ledger of records that is usually used to store transactional data. However, there is no restriction on the type of data that can be stored health records for health applications, ownership records for property management and even executable code in the form of smart-contracts. This indicates how flexible blockchains can be. Blockchain technology came into existence when a white paper titled Bitcoin: A peer-to-peer electronic cash system was announced by Satoshi Nakamoto in 2008. However, the word blockchain was never used in the paper specifically. The paper discusses a peer-to-peer version of electronic cash that overcome a well-known limitation in electronic cash (cryptocurrency) known as double-spending.

One of the major milestones in blockchain technology was the introduction of a blockchain framework named Ethereum (Buterin, 2014). It provides a powerful application stack embedded within each node of the Ethereum network, known as the Ethereum Virtual Machine. It also provides Solidity, a programming language for writing smart contracts, inspired by many object-oriented programming languages, to simplify and standardize the development of Decentralized Applications or DApps. The development of DApps in Ethereum is achieved through writing smart contracts using Solidity and implementing other parts of the DApp using Web3 (Buterin, 2014).

Today, there are many blockchain/distributed ledger platforms that enable the development of blockchain-based applications or integrating blockchain into existing software systems. For instance, we have Hyperledger Fabric, Quorum, Corda, Ripple, Tezos, Sawtooth, BigChainDB, OpenChain, etc. All of these platforms provide technology stacks that enable the developers to build and blockchain-based applications. The recent developments of blockchain platforms and the widespread of blockchain-based applications introduced new challenges to the software development communityblockchain developers encountering several challenges when working with blockchain. To solve these challenges, blockchain developers introduced new software design patterns or adapted existing patterns.

Software design patterns played a significant role in software engineering practice and education since the term was coined (Gamma et al., 1993). Design patterns are generally considered as the accepted

Rajasekar, V., Sondhi, S., Saad, S. and Mohammed, S.
Emerging Design Patterns for Blockchain Applications.
DOI: 10.5220/0009892702420249
In *Proceedings of the 15th International Conference on Software Technologies (ICSOFT 2020)*, pages 242-249
ISBN: 978-989-758-443-5

practices applied to software system design, which help in making the system portable, clean and efficient. It is not entirely clear how design patterns affect the performance of a system; although a few traditional patterns do optimize memory consumption (Flyweight Pattern) and speed optimization (Proxy Pattern).

These patterns however, are not a one-size-fits-all utility. Different systems benefit from different patterns; so selecting the correct ones for your application is important. Similarly, when new technologies crop up, they create the need for new design patterns. Examples include technologies like cloud computing, the internet of things and blockchain. Since these technologies differ significantly from traditional software systems, they require design patterns that are suited to their unique attributes. Here, we attempt to find and validate the effect of design patterns on the efficiency, security and reusability of blockchain applications.

This short paper is organized as follows. In section 2, we identity and discuss common blockchain design patterns. In section 3 we provide brief empirical analysis and discuss how the patterns have matured with time. Finally, in section 4 we provide the conclusion for our findings and our stance on the matter.

2 BLOCKCHAIN DESIGN PATTERNS

Several studies have been proposed in the literature to introduce some new patterns or to extend existing patterns to fit blockchain-application requirements. This section compiles the patterns found and presents each one with a description, the need for the pattern and the consequences of using it. Wherever applicable, real-world context is also provided.

Checks-Effects-Interactions Description - The contract first **checks** whether the necessary conditions are met, then makes necessary changes to the contract state (**effects**) and finally passes control to another contract (**interactions**) (Wohrer and Zdun, 2018).

Need - If the "interactions" occur before the "effects" are recorded and the invoked contract makes a callback to the original one, the system state can be exploited. And it was, in the hack of The DAO (Atzei et al., 2017). The attacker made a recursive call to a function in such a way that the contract's funds were transferred (interaction) before the program could ever record the transfer of funds (effects).

Consequences - The advantage of this software pattern is that it ensures all debits and credits are recorded before any further transactions can be made. A disadvantage is that the pattern is counter-intuitive in the sense that it is generally good practice to wait for a function to execute successfully, before making any state changes.

Secure Ether Transfer Description - This is infact an anti-pattern. The low-level Solidity function, *address.call()* should not be used to invoke a contract. Instead, the high-level *address.send()* or *address.transfer()* should be preferred because they set a limit on the amount of gas[1] the contract can forward to any invoked contract. (Wohrer and Zdun, 2018; Whrer and Zdun, 2018)

Need - Since the low-level function will forward all remaining gas to the invoked contract, using it allows further invocations to be made, provided the gas sent in the original call is sufficiently larger than the required minimum.

Consequences - The advantage is that if the amount of gas transferred is capped, the extra invocations will end eventually due to insufficient funds.

Oracle Description - A contract or blockchain-based application may sometimes need to access information from the external world. The oracle pattern uses a third-party verifier to verify that information stored off-chain is valid, before it can be relayed to the blockchain. (Xu et al., 2018; Whrer and Zdun, 2018; Zhang et al., 2018)

Need - There are several scenarios where a blockchain may need to access external data. This depends on the application's use-case but a few examples are: stock market data, weather-related data and bank account information. Since such information is stored off-chain, it's validity may be compromised and the trust generated by the blockchain can be compromised without even attacking it directly.

Consequences - Some considerations when using an oracle are to be sure you can trust the third-party. It can also be argued that this introduces an element of centralisation in the blockchain-based application. But that is something that can be controlled while developing the oracle, as shown by (Griffith, 2017) and (Costa, 2019).

Off-chain Datastore Description - If the raw data to be stored on the blockchain takes too much memory, calculate a hash for the entire raw data and store this on the blockchain instead.(Lu et al., 2018)

Need - Since the blockchain is a distributed store of data, many nodes in the network must store the entire blockchain, which becomes infeasible if too much data is stored in every block. Additionally, on public blockchains, storing data requires money and can get

[1]Gas is a monetary token or the "cost" for any transaction in Ethereum

expensive quickly.
Consequences - The hash will let you know if the data has been tampered with or not. This is called hash integrity. However, tampering cannot be reversed and if the data is deleted or lost, only its hash remains permanently.

State Channel Description - Micro-transactions should not be stored on the blockchain; they should be stored off-chain and only the final settled amounts should be stored on-chain (Xu et al., 2018). There are some use-cases of this pattern, the most common being the Lightning Network in Bitcoin (Poon and Dryja, 2015) and Plasma in Ethereum.
Need - As mentioned, storing large amounts of data on the blockchain is not sustainable for scalability and sometimes cost issues. In the case of micro-transactions, the amount being transacted is significantly lesser than the processing fees and will take up as much space as any other transaction on the chain.
Consequences - Using the state channel pattern can thus save storage space and costs. A drawback could be the trustworthiness of the state-channel protocol which is different from the blockchain's.

Contract Registry Description - Every contract and its address are stored off-chain as key-value pairs. This mapping is mutable since it is off-chain, which means that the address of a contract can be updated in the registry. Since calls to any contract will now go through the registry, this leaves all dependencies intact. (Xu et al., 2018; Whrer and Zdun, 2018)
Need - Code on a smart contract may need to be modified or updated to deal with bugs and changing requirements. Since data on a blockchain cannot be modified, the registry pattern offers an abstraction over contract names and their code.
Consequences - The advantage is that this allows us to update our contract logic and can be used to rectify bugs and errors. The registry being stored off-chain seems like a drawback but Ethereum itself stores a lot of important data off-chain as we'll see in Section 3.

Data Contract Description - Store data and code on separate contracts to decouple them.(Xu et al., 2018; Whrer and Zdun, 2018)
Need - If a contract stores its data within itself, modifying the contract using the registry or proxy pattern will mean rewriting all of the data to the updated version as well. Since the outdated contracts still remain on the blockchain, and are only indirectly replaced by the updated versions, the data will be stored redundantly with every new version of the contract (Volland, 2018). Conversely, storing the data on a data contract would reduce redundant data storage and save space as well as money.
Consequences - The consequences of this separation are the obvious cost and storage efficiency. Some negative consequences would be that the separation of logic from data increases complexity of code due to external calls. Moreover, external calls increase chances of unintended behaviour.

Embedded Permission Description - Contracts must have an embedded permission field for certain critical functions, which allows only authorized users to run them. This is comparable to an end-user not having access to all the back-end code.(Xu et al., 2018; Lu et al., 2018)
Need - A good example of a critical or sensitive function is *self_destruct()* in Solidity[2]. The absence of embedded permission to execute this particular function was leveraged in the Parity Wallet hack (Destefanis et al., 2018).
Consequences - To use this design pattern one must first decide on the method to check for permissions. For instance, you could perform authorization using the user's private key, although this may pose problems when permissions change or a private key is lost.

Factory Contract Description - This pattern is similar to the idea of a class in programming languages. It entails storing a template contract on the blockchain to instantiate other contracts having a similar structure and flow.(Xu et al., 2018; Zhang et al., 2018)
Need - Using a factory contract helps make a DApp's code modular and reduces the amount of updates required when new functionality is added to the DApp. It also ensures that certain design principles are consistent throughout the application.
Consequences - The factory contract makes it easier for beginners to create contracts which follow predefined best-practices. It can reduce the use of inefficient design but may however, imply extra costs. This is specifically applicable if the application is running on a public blockchain as it most likely will be.

Emergency Stop Description - Include the ability for an authorized party to stop the execution of a contract.(Wohrer and Zdun, 2018)
Need - If a contract is executing malicious functions recursively or for several iterations, without the ability to interrupt, the malicious function can run indefinitely and the nodes can do nothing till execution completes. While the attacker hacked The DAO, the participants on the Ethereum network could only watch as their money was slowly siphoned out of their contracts and into the attacker's.
Consequences - Having an authorized emergency stop functionality will allow an authorized party to prevent further damage in cases of malicious or erroneous executions. Choice of the authorized party is an impor-

[2]It is technically an opcode for the EVM - like the ones in low-level assembly languages

Table 1: Design Patterns Discussed.

Name	Domain	Usefulness	Similar Traditional Patterns
Checks-Effects-Interactions	Security	Avoid unexpected errors	Chain of Responsibility
Secure Ether Transfer	Security	Minimize risk	—
Oracle	Data	Maintain integrity	—
Off-chain Datastore	Data	Storage and Integrity	—
State Channel	Structural	Storage	—
Contract Registry	Structural	Flexibility	Proxy Pattern
Data Contract	Structural	Avoid unexpected errors	—
Embedded Permission	Security	Authentication and authorization	Facade
Factory Contract	Creational	Convenience	Factory
Emergency Stop	Security	Minimize Risk	Memento
Mutex	Security	Synchronisation	Mutex
Create Balance Limit	Security	Minimize Risk	—
Reverse Verifier	Data	Availability and Authentication	Observer
Incentive Execution	Behavioral	Good practice	Composite
Commit and Reveal	Security	Authentication and Authorization	Iterator
Proxy Contract	Structural	Flexibility	Proxy Pattern
Dynamic Binding	Structural	Authentication and Authorization	Chain of Responsibility
Flyweight	Structural	Storage	Flyweight
Tight Variable Packing	Behavioral	Storage	—

tant consideration. Moreover, having a single authority may lead to centralisation - to tackle this multi-party authorization can also be used.

Mutex Description - This patern is similar to mutexes in traditional programming. It prevents a contract from executing code in its parent contract, until the parent has executed completely. (Wohrer and Zdun, 2018) The parent contract here is the contract that invokes another contract.

Need - Having a recursive call (or a callback) that modifies the state of the parent, before it has finished its execution, can cause serious issues as described several times above. Using a mutex is just another way to assure that such attacks (like re-entrancy) cannot occur.

Consequences - There aren't many negative consequences of using a mutex - generally if you find the mutex hindering your task, there's probably a better way to accomplish the task.

Contract Balance-limit Description - This pattern states that a contract should not hold any more funds than a predefined balance-limit. It must reject any further transactions made to it except for forced payments [3]. (Wohrer and Zdun, 2018)

Need - This design pattern will reduce the risk if any individual contract is compromised. Having a contract with too much capital reduces the target area for a potential attack to take place.

Consequences - The consequences of this design pattern are that once the balance-limit has been reached, the contract will not accept any more gas, therefore nobody can invoke the contract again unless there is some provision to send the funds elsewhere.

Reverse Verifier Description - Sometimes an application may need to access data from the blockchain. Like the oracle pattern, reverse verifier is used to verify data being sent to an external source from the blockchain.(Xu et al., 2018)

Need - It is common to have application data in a traditional database and store only a hash on the blockchain. One shortcoming of this is that we cannot prevent tampering of data in the database. The tampering will be detected however, thanks to the hash value. The reverse verifier checks for hash integrity when blockchain data is requested by external components.

Consequences - One possible consequence of using a reverse verifier is that communication between external components and the blockchain may take some time. Additionally, the reverse verifier must also be a trusted party.

Incentive Execution Description - Make seldom-run utility functions piggyback onto contracts that execute more often.

Need - Some functions perform tasks like cleaning up expired records, making dividend payouts and destroying deprecated contracts. These utility functions also require gas to be executed, and the payout is usually not enough to offset the execution cost. Therefore reimbursing callers or adding these functions to other contracts are possible solutions. (Xu et al., 2018)

Consequences - There are some things to consider if this pattern is used. For starters, the type of contract that the utilities are appended to; if it contains sensitive functions, experiences a lot of traffic or is called

[3]Functions like *self_destruct()* and also mining rewards sent to a contract, cannot be declined. Therefore they are "forced" payments

more often than the utilities must be, it is not a good fit. Also, the amount of gas required to invoke the contract may increase due to the utility functions.

Commit and Reveal Description - This pattern works by hiding certain secret variables in a contract's function, and only displaying the final values. Authorized users may have access to the secret.(Whrer and Zdun, 2018)
Need - If a contract's internal state is visible to the network, it is possible for a malicious user to take advantage, and invoke the contract with deliberately selected variables that change the state to one they desire. So, this pattern hides some variables from the network, reducing the probability of such injection attacks.
Consequences - Implementing this pattern requires overhead in terms of code. Deciding access roles will also need consideration to prevent centralisation of the system.

Proxy Contract Description - Create a proxy for each contract, which will accept, and then forward the parameters to the current version of the contract.(Whrer and Zdun, 2018)
Need - This pattern is similar in purpose to the contract registry. They both help in modification and updating of contracts.
Consequences - A contract registry has two obvious advantages over a proxy contract: there is only one contract registry but there are several proxies, and using a proxy contract does not allow anything to be returned to the invoking contract. One advantage, is that the proxy contracts unlike the registry, will be stored on-chain.

Dynamic Binding Description Create a dynamic association between a contract and its authorized users' addresses, in such a way that the addresses are not defined in the contract. Instead, the users send a secret key to another, designated contract, which forwards the request to the the main one if the key is valid.(Lu et al., 2018)
Need - This pattern can be used in cases where an extra layer of security is required. Since the authorized user's address is not listed on the contract, it is not visible to the public.
Consequences - This design pattern ensures privacy and also respects the authorization and authentication protocols of the network. It may however, increase gas overhead due to the extra contract invocations and can raise problems if the secret key is lost or compromised.

Flyweight Description - Inspired in part by the data contract design pattern, the flyweight contract stores data that is shared by a group of clients in one common place.(Zhang et al., 2018)
Need - Since this pattern is essentially a data contract itself, the need is the same - to decouple data from programmatic logic. It also helps conserve space by storing a single copy of data that is shared between a group of contracts.
Consequences - The consequences of using the flyweight are that lots of storage space is saved, which translates to money saved. It also means that updating a contract will not affect the shared data. Some considerations while using this pattern are: accessing the data may require extra gas and, some provisions are required for when the data needs to be updated.
Tight Variable Packing Description - Store static variables in smart contracts as the smallest possible data type that they can fit in. For example don't store a value as an *int* if it could be stored as a *byte*. (Volland, 2018)
Need - This is because storage costs money on Ethereum and other public blockchains.
Consequences - There are no negative consequences of using the pattern - it should be followed in traditional programming as well, but has greater consequences in blockchain-oriented software.

In Table 1 each design pattern is assigned a domain that describes what aspect of the blockchain the pattern affects. The domains are listed below with a brief note on each.

- **Security -** Includes design patterns that increase security of the system.
- **Data -** Includes design patterns that deal with data on and off the blockchain.
- **Creational -** Include design patterns that deal with creation of contracts on the blockchain.
- **Structural -** Include design patterns that define structural properties that lead to new functionality.
- **Behavioral -** Include design patterns dealing mainly with good practices and specific behaviors of contracts on the blockchain.

3 PATTERNS MATURITY

We have discussed common design patterns in blockchain-oriented software and how they may be useful to developers. Some of these patterns emerged as a means to tackle malicious users on the public internet and others to make an application more cost-effective or scalable. In this section, we discuss how the design patterns have matured over years and whether they hold any value in blockchain-oriented software engineering at all.

The **oracle** or external data **verifier**, has emerged as one of the most prominent design patterns in the

industry today. Part of its popularity lies in the fact that companies and individuals began implementing oracles as a distributed application i.e. the oracle itself was built on a blockchain. This enables oracles to be used in other industries for data validation as well. ChainLink is one such company that has quickly risen to fame (Ellis et al., 2017). It offers a distributed solution to the external data access problem. The great thing about ChainLink is that it works as an oracle verifier, as well as a **reverse verifier**, i.e. it can validate data coming from a blockchain application too. There are many projects like ChainLink that create distributed oracles and such services are used by other popular DApps. Chainlink itself, connects DApps to external APIs and also provides support for blockchain transactions. It has partnered with the likes of Google, Binance and other major players in the DLT/Blockchain industry.

One DApp that uses Chainlink's oracle services is Loopring - an open protocol for building decentralized exchanges. Despite the P2P and decentralized nature of blockchains, most cryptographic assets today are exchanged at centralised exchanges (like Binance, Bitmex, etc). When trading through these centralised exchanges, traders must transfer their assets to a wallet provided by the exchange, and then the exchange will facilitate the trade. One problem here is that after transferring your assets, you are no longer in control of them - the exchange does not provide the wallet's private key to traders. During this intermediary period, you can't control what happens to your assets and Loopring identifies this as a single point of failure. Their website lists around 40 hacks of exchanges and online marketplaces since 2013, each resulting in a loss of tens of millions of dollars on average. All of this money ultimately belongs to traders using the exchange. Loopring's aim is to create non-custodial, decentralised exchanges. To do this, they use zero-knowledge proofs (ZKP). ZKP aims to allow a user to prove they know a secret value, without disclosing the secret itself. In other words, a trader can prove that he knows his private key without actually disclosing it. This is an application of the **commit-and-reveal** design pattern, where secret values are hidden from unauthorized users and only the final solutions are visible. Using this pattern, Loopring is trying to make blockchains scalable and increase transaction throughput. Apart from commit-and-reveal, Loopring uses an **off-chain data storage** in addition to the Ethereum blockchain.

Another pattern that receives some attention is the **flyweight** pattern. The DApp for Smart Health (DASH) is a distributed application that uses blockchain technology in the healthcare sector (Zhang et al., 2018). Patient records and information are stored on the blockchain. However, storing insurance policies, coverage details and other patient information requires a lot of storage which causes scalability issues. Using the flyweight pattern, DASH stores all policies separately and creates references to appropriate policies for each patient. Since the number of insurance policies is a much smaller subset compared to the number of patients, the flyweight pattern helps to save a lot of storage space.

Coming to the giants in the blockchain space; the Bitcoin community has been working hard on the lightning network since 2016 with the first version going live in early 2018. The Lightning Network is defined as a Layer 2 protocol that functions on top of a blockchain-enabled cryptocurrency like Bitcoin (Poon and Dryja, 2015). The protocol is in its early stages and is only available for public tests now. It uses the **state channel** design pattern by enabling micro-transaction networks to work at a higher layer of abstraction than the payment protocol. Once a certain number of micro-transactions have been recorded and settlement occurs, the final amounts are added onto the blockchain. However, recent research suggests that the micro-transaction network is highly centralised (Lin et al., 2020). It has found "hubs" on the network that handle a majority of the micro-transactions. Further, taking out one of these hubs results in the network collapsing into several components. This is undesirable as it can lead to splits in the network, where two or more groups of validators (validating micro-transactions) start working on separate chains, leading to two or more competing chains. This could destroy the network since each group is likely playing by its own rules. It is worth noting here that the main Bitcoin network too is controlled majorly by only 5 mining pools (companies).

Ethereum, on the other hand, quite successfully uses some of the discussed patterns. Ethereum's environment is designed as a state machine. The state of the network includes the current state of each contract as well as the amount of funds available at each address. The state of a contract is defined by its internal variables and the funds available to it. This state information is stored on an external database (LevelDB and RocksDB) as a trie data structure [4]. The root is hashed and stored with every new block on the blockchain for hash integrity. Additionally, the external database stores current as well as past state-tries of the network, which enables the network to roll-back the state if needed.

The **off-chain data storage** design pattern is wo-

[4]A trie is like a binary tree except that each tree node may have upto n child nodes

ven right into Ethereum. The external database is stored at all validator nodes (called full-nodes). Ethereum also allows ownership of contracts which gives owners certain privileges, defined by the owners themselves. These privileges usually translate to permissions to run certain functions in the contract (like contract termination), thus creating a sort of **embedded permission**. Additionally, patterns like **tight variable packing** are genenrally good practice, especially when dealing with public blockchains where storage costs money and may cause scalability issues.

After a few hacks on Ethereum's platform, the developers started working to prevent further malicious activity on the network. **Checks-effects-interactions** and **mutexes** are two design patterns that were suggested to safeguard against future attacks. They were mostly relevant to hacks like The DAO's which was a re-entrancy attack. Re-entrancy is an attack where contract execution passes recursively from an invoked contract to itself in such a way that the "effects" of the calling contract occur after its "interactions". This means that the state is never updated since the recursive call occurs beforehand and continues indefinitely. Using mutexes and the checks-effects-interactions flow-control can help against such attacks but suppose that a hacker does succeed somehow, the **emergency stop** pattern and **balance limits** can minimize the risk associated with the malicious activity.

Similar to how the above design patterns emerged, the **secure ether transfer** anti-pattern also came about for a similar purpose. The high-level *transfer()* and *send()* functions were created in Solidity to help against re-entrancy attacks. These functions add a cap of 2300 units of gas to be passed to a contract on invocation; limiting the number of times a recursive call can continue (any interaction with a contract in Ethereum requires gas). However, after the Istanbul hard fork[5], the secure ether transfer anti-pattern was scrapped. These changes directly affected the *transfer()* and *send()* functions. Now gas amounts required for different contract invocations increased in many cases. Therefore limiting the gas was now detrimental, as the limit was not enough to see the desired number of invocations through. In other words, the high-level *transfer* and *send* functions now constrain the ability to execute necessary functions in Ethereum - leading the developer community to advise against the use of these two functions (Marx, 2019). In retrospect, they called it a quick-fix for the issues revealed by the hacks. The advise now is to use the low-level *call()* function along with the checks-effects-interactions and mutex patterns.

[5]Istanbul hard fork was an update to Ethereum that made fundamental changes to the existing protocols

4 CONCLUSION

In this paper, we identified common software design patterns and investigated their maturity using an empirical analysis approach. We only consider a proposed pattern (design solution) as a mature pattern if it has been adapted/used by either a blockchain platform or was applied in a well-known blockchain-based application. In general, the identified patterns could be categorized into five main categories. These categories are patterns for data management, patterns for security assurance, patterns for smart contracts, patterns for communication with external entities, and finally patterns to reduce computational cost. Moreover, some patterns are designed for specific blockchain platforms or commonly used in public or private blockchain networks. In the literature, we did not find any quantitative or qualitative assessments for the proposed patterns. Therefore, for some patterns, it is not clear exactly how they affect the quality of the produced system or application. Moreover, extracting or identifying the patterns is a time consuming and intensive task, that could only be achieved using code review and white-box analysis. These are some of the limitations that we plan to address in our future research.

ACKNOWLEDGEMENT

This research was supported by the ***Scotiabank Global Trade Transactions Initiative administered by the University of Windsor's Cross-Border Institute and Mitacs***. We thank our colleagues from Scotiabank and Cross-Border Institute, who provided insight and expertise that greatly assisted the research. However, they may not agree with all of the interpretations/conclusions of this paper.

We are also immensely grateful to ***Dr. William Anderson*** from the University of Windsors Cross-Border Institute for his comments on an earlier version of the manuscript, although any errors are our own and should not tarnish the reputations of any other person.

REFERENCES

Atzei, N., Bartoletti, M., and Cimoli, T. (2017). A survey of attacks on ethereum smart contracts (sok). pages 164–186.

Buterin, V. (2014). Ethereum: A next-generation smart contract and decentralized application platform. Accessed: 2016-08-22.

Costa, P. (2019). Github repository: Blockchain oracle.

Destefanis, G., Marchesi, M., Ortu, M., Tonelli, R., Bracciali, A., and Hierons, R. (2018). Smart contracts vulnerabilities: a call for blockchain software engineering? In *2018 International Workshop on Blockchain Oriented Software Engineering*, pages 19–25.

Ellis, S., Juels, A., and Nazarov, S. (2017). Chainlink: A decentralized oracle network.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1993). Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of 7th European Conference on Object Oriented Programming*, pages 406–431.

Griffith, A. (2017). Github repository: Concurrence.io.

Lin, J.-H., Primicerio, K., Squartini, T., Decker, C., and Tessone, C. J. (2020). Lightning network: a second path towards centralisation of the bitcoin economy.

Lu, Q., Xu, X., Liu, Y., and Zhang, W. (2018). Design pattern as a service for blockchain applications. pages 128–135.

Marx, S. (2019). Blog: Stop using solidity's transfer() now.

Poon, J. and Dryja, T. (2015). The bitcoin lightning network: Scalable off-chain instant payments.

Volland, F. (2018). Github blog: Solidity patterns.

Whrer, M. and Zdun, U. (2018). Design patterns for smart contracts in the ethereum ecosystem. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1513–1520.

Wohrer, M. and Zdun, U. (2018). Smart contracts: security patterns in the ethereum ecosystem and solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 2–8.

Xu, X., Pautasso, C., Zhu, L., Lu, Q., and Weber, I. (2018). A pattern collection for blockchain-based applications.

Zhang, P., Schmidt, D., White, J., and Lenz, G. (2018). *Blockchain Technology Use Cases in Healthcare*.