A Pattern Collection for Blockchain-based Applications

Xiwei Xu*
Data61, CSIRO
Eveleigh, NSW, Australia
CSE, UNSW
Sydney, NSW, Australia
xiwei.xu@data61.csiro.au

Cesare Pautasso
Software Institute, Faculty of Informatics, USI
Lugano, Switzerland
c.pautasso@ieee.org

Liming Zhu
Data61, CSIRO
Eveleigh, NSW, Australia
CSE, UNSW
Sydney, NSW, Australia
Liming.Zhu@data61.csiro.au

Qinghua Lu
Data61, CSIRO
Eveleigh, NSW, Australia
CSE, UNSW
Sydney, NSW, Australia
qinghualu@upc.edu.cn

Ingo Weber
Data61, CSIRO
Eveleigh, NSW, Australia
CSE, UNSW
Sydney, NSW, Australia
Ingo.Weber@data61.csiro.au

ABSTRACT

Blockchain is an emerging technology that enables new forms of decentralized software architectures, where distributed components can reach agreements on shared system states without trusting a central integration point. Blockchain provides a shared infrastructure to execute programs, called smart contracts, and to store data. Since blockchain technologies are at an early stage, there is a lack of a systematic and holistic view on designing software systems that use blockchain. We view blockchain as part of a bigger system, which requires patterns for using blockchain in the design of their software architecture. In this paper, we collect a list of patterns for blockchain-based applications. The pattern collection is categorized into four types, including interaction with external world patterns, data management patterns, security patterns and contract structural patterns. Some patterns are designed considering the nature of blockchain and how it can be specifically introduced within real-world applications . Others are variants of existing design patterns applied in the context of blockchain-based applications and smart contracts.

CCS CONCEPTS

• **Software and its engineering → Patterns**; *Designing software*;

KEYWORDS

Blockchain, Smart contract, Patterns

ACM Reference Format:
Xiwei Xu, Cesare Pautasso, Liming Zhu, Qinghua Lu, and Ingo Weber. 2018. A Pattern Collection for Blockchain-based Applications. In *23rd European Conference on Pattern Languages of Programs (EuroPLoP '18), July 4–8, 2018, Irsee, Germany*. ACM, New York, NY, USA, **??** pages. https://doi.org/10.1145/3282308.3282312

*This is the corresponding author

EuroPLoP '18, July 4–8, 2018, Irsee, Germany

ACM ISBN 978-1-4503-6387-7/18/07…$15.00
https://doi.org/10.1145/3282308.3282312

1 INTRODUCTION

Blockchain is the technology behind Bitcoin [**?**], which is a digital currency based on a peer-to-peer network and cryptographic techniques. The blockchain provides immutable, append-only, shared data storage, which only allows inserting transactions without updating or deleting any existing ones, thus preventing any tampering or revision of previously stored data on blockchain as long as the majority of the network peers do not agree to allow such revision. The blockchain enables decentralization as new forms of distributed software architectures, where components can reach agreements on the historical log of shared states for decentralized and transactional data sharing, across a large network of untrusted participants without relying on a central integration point.

Financial transactions are the first, but far from the only use case being investigated for blockchain. Many start-ups, enterprises, and governments [**?**] are exploring blockchain-based applications in areas as diverse as supply chain, electronic health records, voting, energy supply, ownership management, and protecting critical civil infrastructure. Despite of the wide array of interest in blockchain technology, there is a lack of a systematic and holistic view when applying blockchain in the design of software applications.

Previous work has characterized blockchain from a software architecture perspective as a software connector [**?**] that provides a shared infrastructure for storing data and running programs (known as *smart contracts*). Blockchain has unique properties including immutability, non-repudiation, data integrity, transparency, and equal rights. It also has two main limitations, namely, lack of privacy and poor performance [**?**]. The taxonomy presented in [**?**] discusses such properties for different types and configurations of blockchain technology. To better leverage the positive properties of blockchain and avoid limitations, more architectural guidance on blockchain-based applications is needed.

In this paper, we propose a set of patterns for the design of blockchain-based applications. In software engineering, a design pattern is a reusable solution to a problem that commonly occurs

within a given context during software design [**?**]. A design pattern defines constraints that restrict the roles of architectural elements (processing, connectors and data) and the interaction among those elements. Adopting a design pattern causes trade-offs among quality attributes. Our pattern collection includes three patterns about interaction between blockchain and the external world, four data management patterns, three security patterns and five structural patterns of smart contract. The pattern collection provides an architectural guidance for developers to build applications on blockchain.

The remainder of the paper is organized as follows. Section **??** presents a background of blockchain and smart contracts. Section **??** gives an overview of the pattern collection, followed by detailed patterns discussed from Section **??** to Section **??**. Related work on blockchain-based applications and design patterns is discussed in Section **??**. Section **??** concludes the paper and outlines future work.

2 BACKGROUND

2.1 Blockchain

Blockchain is a data structure of an ordered list of blocks, which "chained" back to the previous block through containing a hash of a presentation of the previous block. Every block on blockchain contains a list of transactions (possibly empty). Due to the security properties of hash function, the historical transactions on blockchain can not be deleted or altered without invalidating the chain of hashes. In addition to the design of the data structure, there are computational constraints and consensus protocols applied to the creation of blocks. All together, blockchain can in practice prevent revision and tampering of the information on blockchain. In blockchain network, public key cryptography and digital signatures are used to identify accounts and authorize transactions submitted to a blockchain. Fig. **??** gives an overview of blockchain data structure, blockchain network and transaction life cycle.

A transaction is a data package that stores information for money transfer, like sender, receiver, and monetary value, or the (compiled) code of smart contracts, or parameters of function calls of smart contracts. Section **??** discusses more on the concept of smart contract. The life-cycle of a transaction starts from the transaction being signed by its initiator using the private key, to authorize the expenditure of the monetary value or the function call associated with the transaction. Some transactions require authorization from a set of private keys corresponding to multiple addresses. The signed transaction is sent to a node within the blockchain network. The transaction is validated by the receiver. Algorithmic rules and cryptographic techniques are used to check the integrity of the transaction. If the transaction is valid and previously unknown to the node, it is propagated to other nodes. These receivers validate the transaction before further propagating it to their peers until the transaction reaches all nodes within the network.

The process of appending new blocks to the blockchain data structure is called *Mining*. *Miners* in a blockchain network are responsible for aggregating valid new transactions into blocks, adding the blocks to the blockchain data structure, and propagating the blocks to the blockchain network. Every new block is broadcast across the blockchain network, where each node stores a replica of the whole blockchain. For every new block with the latest set of transactions, the whole network needs to reach a consensus about whether to include the set of transactions into the blockchain. A distributed consensus mechanism is used to govern the addition of new blocks, which consists of the rules for validating and broadcasting transactions and blocks, resolving conflicts, and the incentive scheme. The consensus mechanism ensures all the new transactions are valid, and that each valid transaction is added only once. There are different consensus mechanisms, *e.g.* , *Proof-of-work* or *Proof-of-stake* (see *e.g.* [**?**] for details). *Proof-of-work* is used by prominent blockchain systems, like Bitcoin and Ethereum, which can only offer probabilistic guarantees to their clients in terms of the immutability of recorded transactions [**?**]. There is always a chance that the most recent few blocks are replaced by a competing chain fork.

When using a blockchain, one design decision is the deployment, *i.e.* , whether to use a public blockchain, consortium/community blockchain or private blockchain [**?**]. Most cryptocurrencies use public blockchains, which can be accessed by anyone on the Internet. In a cryptocurrency ecosystem, the users typically interact with the blockchain by using a *wallet*. Wallet is a software program to support money transfer and basic smart contract transfer (*e.g.* , on Ethereum). Normally, a wallet runs a light node of the cryptocurrency, which only downloads the block header rather than the complete blockchain to validate the authenticity of new transactions. Light nodes are easy to maintain and run. However, a light node may leave the user vulnerable because it skips several security steps. Using a public blockchain results in better information transparency and auditability, but sacrifices performance and has a different cost model compared with a conventional data storage. It costs monetary value to store data or execute code on a public blockchain. In a public blockchain, data privacy relies on encryption or cryptographic hashes.

A consortium blockchain is used across multiple organizations. The consensus process in a consortium blockchain is controlled by pre-authorised nodes. The right to read the blockchain may be public or may be restricted to specific participants. In a private blockchain network, write permission is kept within one organization, although this may include multiple divisions of a single organization. Consortium and private blockchain can be an instantiation of a public blockchain with a permission management component that authorises participants within the network.

Properties of Blockchain. The data contained in a committed transaction on blockckchain is seen as *immutable* in practice. The chain of immutable cryptographically-signed historical transactions provides *non-repudiation* of the stored data. Cryptographic techniques used by blockchain support data *integrity*, the public access provides data *transparency*, and *equal rights* allows every participant to have the same ability to manipulate the data on blockchain. Such rights can be weighted by the computational power (*Proof-of-work*) or stake (*Proof-of-stake*) owned by a node. *Trust* of the blockchain is built based on the interactions between nodes within the blockchain network. The participants of a blockchain network rely on the design of blockchain, the cryptographic techniques used by blockchain and the blockchain network itself rather than relying on trusted third-party to facilitate transactions. If a user interacts

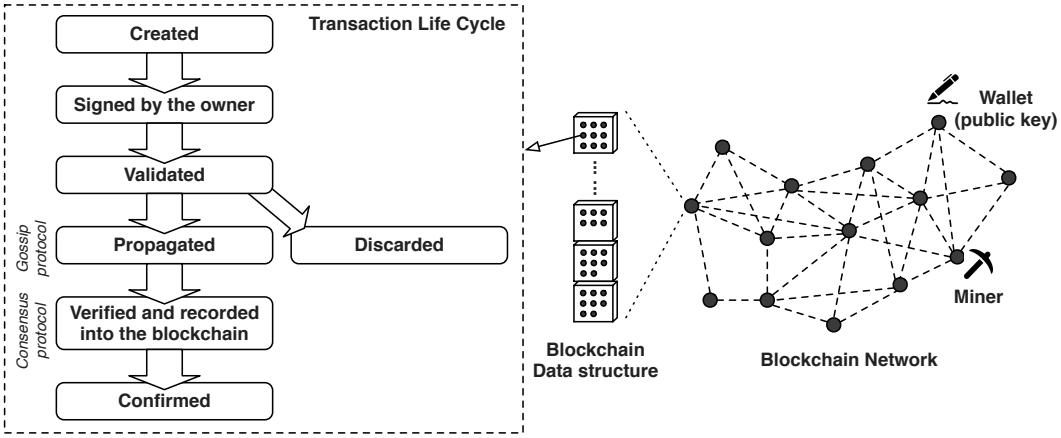

Figure 1: Blockchain overview

with blockchain through using a wallet service, the service provider is a third party trusted by the user.

Limitations. Data privacy and *scalability* are the main two limitations of public blockchains. Data privacy on public blockchain is limited because there is no privileged user, and every participant can join the network to access all the information on blockchain and validate new transactions. There are scalability limits on (i) the size of the data on blockchain, (ii) the transaction processing rate, and (iii) the latency of data transmission and commits. Latency between submitting a transaction and it being committed on a blockchain is affected by the consensus protocol. This is around 1 hour (10-minute block interval with time for inclusion and 5-block confirmation) on Bitcoin, and around 3 minutes (14-second block interval with 11 confirmation blocks) on Ethereum[1]. Times in practice can be even longer [**?**]. The number of transactions included in each block is also limited by the bandwidth of nodes participating in the network (for Bitcoin the current bandwidth per block is 1MB) [**?**]. Ethereum applies a so-called *gas* limit to blocks (*gas* is the internal pricing unit for executing a transaction or storing data), which limits the number and complexity of transactions that can fit into a block. Note that all of these limitations are subject to large efforts of ongoing work.

2.2 Smart Contract

The first generation of blockchains, like Bitcoin, provides a public ledger to store cryptographically-signed financial transactions [**?**]. The tokens on Bitcoin blockchain are known as BTC. There is very limited capability to support programmable transactions, and only very small pieces of auxiliary data could be embedded in the transactions (or attached to the tokens) to serve other purposes, such as representing other digital assets or physical assets.

The second generation of blockchains provides a general-purpose programmable infrastructure with a public ledger that records the computational results. Programs, known as *smart contracts* [**?**], can be deployed and run on a blockchain. Smart contracts can express triggers, conditions and business logic [**?**] to enable more complex programmable transactions. The signature of the transaction initiator authorizes the data payload of a transaction or the creation or execution of a smart contract. A common simple example of a smart contract-enabled service is escrow, which can hold funds until the obligations defined in the smart contract have been fulfilled. Smart contracts are pure functions by design, which cannot access the state of external systems directly.

Smart Contract Languages. Script used by Bitcoin is a simple stack-based scripting language[2], which is intentionally designed not to be Turing-complete. Script provides the flexibility to define conditions required to spend the Bitcoin associated with the transactions, for example, requiring multiple private keys to authorize the payment. Ethereum is currently the most widely-used blockchain that supports general-purpose (Turing-complete) smart contracts. The primary smart contract language used on Ethereum blockchain is *Solidity*[3]. DigitalAsset[4] proposed DAML[5]as a domain specific smart contract language for financial institutes. Smart contracts running on Hyperledger Fabric[6] are called Chaincode, which can be written in any programming language and executed in containers inside the fabric context layer.

2.3 Blockchain as a Component of Application System

The architecture of a software system, where blockchain is one of the components, is shown in Fig. **??**. In this system, blockchain is responsible for storing and sharing data, and executing smart contracts. The blockchain component might also have tokens as digital currencies or representing other assets. Due to the limitations of privacy and performance, there are off-chain auxiliary databases used in the system. First, private data is stored in an internal database. Second, data with large size is stored in a separate data storage, which could be a cloud service. There is a API layer between the

[1]https://www.ethereum.org/

[2]https://en.bitcoin.it/wiki/Script
[3]https://solidity.readthedocs.io/
[4]http://www.digitalasset.com/
[5]http://hub.digitalasset.com/blog/introducing-the-digital-asset-modeling-language-a-powerful-alternative-to-smart-contracts-for-financial-institutions
[6]https://hyperledger.org/projects/fabric

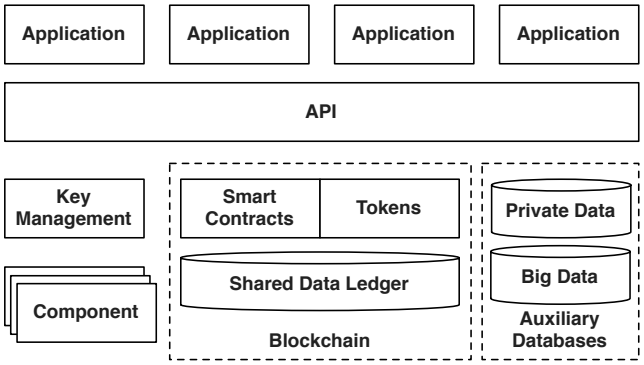

Figure 2: Blockchain as a component within a software architecture

data storage layer and the applications using the blockchain, which is same as with conventional technology. Key management is an essential component working with blockchain. Every participant in a blockchain network has one or more private keys, which are used by the participant to digitally sign the transactions relating to the addresses of the participant. The security of these private keys is very important. If the private key of a user is stolen, any other user in the system can forge transactions from that user to spend the assets belonging to the user, invoke functions of smart contracts in their name. Blockchain also interact with other off-chain components.

3 BLOCKCHAIN-BASED APPLICATION PATTERN COLLECTION

In this section, we discuss the blockchain-based application pattern collection, which currently includes fifteen design patterns that shape the architectural elements and their interactions in blockchain-based applications. Table. **??** gives an overview of these patterns. Applying the patterns to an application can better align it with the unique properties provided by blockchain, avoid its limitations, and achieve other quality attributes.

The patterns about interaction between blockchain and the external world describe different ways for blockchain to communicate data with the external world, including *Oracle* (Section **??**), *Reverse oracle* (Section **??**) and *Legal and smart contract pair* (Section **??**). The four data management patterns are about managing data on and off blockchain, including *Encrypting on-chain data* (Section **??**), *Tokenisation* (Section **??**), *Off-chain data storage* (Section **??**) and *State channel* (Section **??**). The three security patterns concern the security aspect of the blockchain-based applications. *Multiple authorization* (Section **??**) and *Off-chain secret enabled dynamic authorization* (Section **??**) are aimed at adding dynamism to authorization of transactions and smart contracts. *X-confirmation* (Section **??**) is a pattern that further increases the security of transactions. The five contract structural patterns define the dependencies among smart contracts and behaviour of smart contract. Smart contracts on blockchain are immutable. Upgrading a smart contract to a new version is a challenge which hinders the evolution of blockchain-based applications. *Contract registry* (Section **??**) and *Data contract* (Section **??**) are two patterns that aim to improve upgradability of smart contracts. Two patterns aim to improve security of smart contracts: *Embedded permission* (Section **??**) and *Factory contract* (Section **??**). Finally, *Incentive execution* (Section **??**) concerns the maintenance of smart contracts.

In this paper we follow the extended pattern form from [**?**], which includes the name of the pattern, a short summary, the context, the problem statement, an explicit discussion of the forces which make the problem difficult, the solution, its consequences, and some examples of real-world known uses of the pattern. Forces are identified with the corresponding quality attribute, as sometimes the solution will propose a trade-off between them. Regarding the consequences, we distinguish the benefits and drawbacks. Finally, we discuss features only applicable to a certain deployment of blockchain, such as monetary cost of data storage and code execution.

4 INTERACTION WITH EXTERNAL WORLD PATTERNS

Due to the unique properties and limitations of blockchain, the main architectural consideration for a blockchain-based software application is to decide what data and executable code (smart contract) should be kept on-chain, and what should be kept off-chain. Two factors need particular attention, namely performance and privacy. Performance highly depends on the type of deployment of the blockchain. For example, a consortium blockchain [**?**] can be configured to achieve much better performance than a public blockchain. As a component of a big software system, blockchain needs to communicate data with other components within the software system (Fig. **??**).

4.1 Pattern 1: Oracle

Summary: Introduce the state of external systems into the closed blockchain execution environment through the oracle. Fig. **??** is a graphical representation of the pattern with the external oracle solution approach.

Context: From the software architecture perspective, blockchain can be viewed as a component or connector within a large software system [**?**]. In the case the blockchain is used as a distributed database for more general purposes other than financial services, the applications built on blockchain might need to interact with other external systems. Thus, the validation of transactions on blockchain might depend on states of external systems.

Problem: The execution environment of a blockchain is self-contained. It can only access information present in the data and transactions on the blockchain. Smart contracts running on blockchain are pure functions by design. The state of external systems are not directly accessible to smart contracts. Yet, function calls in smart contracts sometimes need to access state of the external world.

How can function calls in smart contracts be enabled to access the state of the external world from within smart contracts?

Forces: The problem requires to balance the following forces:

- *Closed environment.* Blockchain is a secure, self-contained environment, which is isolated from external systems. Smart

Table 1: Overview of the Blockchain-based Application Pattern Collection.

Category	Name	Summary
Interaction with External World	Oracle	Introducing the state of external systems into the closed blockchain execution environment.
	Reverse oracle	The off-chain components of an existing system rely on smart contracts running on a blockchain to supply requested data and check required conditions.
	Legal and smart contract pair	A bidirectional binding is established between a legal agreement and a corresponding smart contract.
Data Management	Encrypting on-chain data	Ensure confidentiality of the data stored on blockchain by encrypting it.
	Tokenisation	Using tokens on blockchain to represent transferable digital or physical assets or services.
	Off-chain data storage	Use hashing to ensure the integrity of arbitrarily large datasets which may not fit directly on the blockchain.
	State channel	Transactions that are too small in value relative to a blockchain transaction fee or that require much shorter latency than can be provided by a blockchain, are performed off-chain with periodic recording of net transaction settlements on- chain.
Security	Multiple authorization	A set of blockchain addresses which can authorise a transaction is predefined. Only a subset of the addresses is required to authorize transactions.
	Off-chain secret enabled dynamic authorization	Using a hash created off-chain to dynamically bind authority for a trans- action.
	X-confirmation	Waiting for enough number of blocks as confirmations to ensure that a transaction added into blockchain is immutable with high probability.
Structural Patterns of Contract	Contract registry	Before invoking a smart contract, the address of the latest version of the smart contract is located by looking up its name on a contract registry.
	Embedded permission	Smart contracts use embedded permission control to restrict access to the invocation of the functions defined in the smart contracts.
	Data contract	Store data in a separate smart contract.
	Factory contract	An on-chain template contract is used as a factory that generates con- tract instances from the template.
	Incentive execution	A reward is provided to the caller of a contract function for invoking it.

contracts on blockchain cannot read the states of the external systems.

- *Connectivity.* In addition to the data found on the blockchain, general-purpose applications might require information from external systems, for example, context information like geo-location information, or weather data from a Web API[7].
- *Long-term availability and validity.* While transactions on blockchain are immutable, the external state used to validate a transaction may change or even disappear after the transactions were originally appended to the blockchain.

Solution: To connect the closed execution environment of blockchain with the external world, a oracle is introduced to evaluate conditions that cannot be expressed in a smart contract running within the blockchain environment. A oracle is a trusted third party that provides the smart contracts with information about the external world. When validation of a transaction depends on external state, the oracle is requested to check the external state and to provide the result to the validator (*miner*), which then takes the result provided by the oracle into account when validating the transaction. The oracle can be implemented inside a blockchain network as a smart contract with external state being injected into the oracle periodically by an off-chain *injector*. Later, other smart contracts can access the data from the oracle smart contract. A oracle can be also implemented as a server outside the blockchain. Such an external oracle needs the permission to sign transactions using its own key pair on-demand. Extra mechanisms might be needed to improve trustworthiness of the oracle, for example, a distributed oracle based on multiple servers and M-of-N multiple signature. Through using oracle, the validation of transactions is based on the authentication of the oracle, rather than the external state.

Consequences:

Benefits:

[7] https://openweathermap.org/api

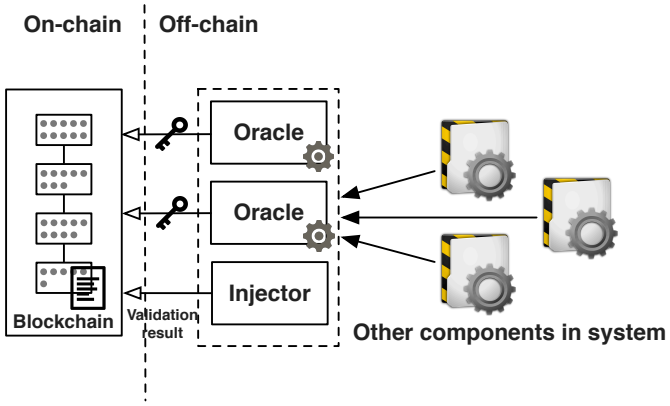

Figure 3: Oracle Pattern

- *Connectivity.* The closed execution environment of blockchain is âĂIJconnectedâĂİ with external world through the oracle. The applications based on blockchain can access external states through the oracle and use the external states to validate transactions.

Drawbacks:

- *Trust.* Using oracle introduces a trusted third party into the system. The oracle selected to verify the external state needs to be trusted by all the participants involved in relevant transactions.
- *Validity.* The external states injected into the transactions can not be fully validated by miners. Thus, when miners validate the transaction including external state, they rely on the oracle to check the validity of the information from external world. *Long-term availability and validity.* It could happen that while transactions are immutable, the external state used to validate them may change after the transactions were originally appended to the blockchain.

Related patterns: *Reverse Oracle* (Section **??**)

Known uses:

- *Oracle* in Bitcoin is an instance of this pattern [8]. Oracle is a server outside the Bitcoin blockchain network, which can evaluate user-defined expressions based on the external state.
- A central oracle becomes a potential single point of failure for the transactions relying on the oracle. To improve the trustworthiness of the oracle, a *distributed oracle* can be introduced. A distributed oracle contains several oracles that provide the same functionality to check the external state. All the oracles need to be trusted by the whole network. In this case, a transaction that relies on external state can use a multi-signature (M-of-N) schema that requires keys from M out of N oracles to authorize a transaction. Orisi[9] on Bitcoin maintains a set of independent oracles. Orisi allows the participants involved in a transaction to select a set of oracles and define the value of M before initiating a conditional transaction.
- Participants who wish to transact with each other on a blockchain could rely on an ad hoc arbitrator trusted by all the participants to resolve disputes or check external state. An arbitrator may be a human with a blockchain account who is able to sign transactions. Alternatively, an arbitrator may be automated and validate transactions based on state taken from the blockchain and the external world. For example, Gnosis[10] is a decentralized prediction market that allows users to choose any oracle they trust, such as another user or a web service, *e.g.* , for weather forecasts.

4.2 Pattern 2: Reverse Oracle

Summary: The reverse oracle of an existing system relies on smart contracts running on blockchain to validate requested data and check required status. Fig. **??** is a graphical representation of the pattern.

Context: In a software system, where blockchain is one of the components, the off-chain components might need to use the data stored on the blockchain and the smart contracts running on the blockchain to check certain conditions.

Problem: Some domains use very large and mature (or even legacy) systems, which comply with existing standards. For such domains, an non-intrusive approach is desired to leverage the existing complex systems with blockchain without changing the core of the existing systems.

How to integrate the blockchain within existing systems?

Forces: The problem requires to balance the following forces:

- *Connectivity.* Integrating blockchain into an existing system to leverage the unique properties of blockchain.
- *Simplicity.* Introduce minimal changes to the existing system.

Solution: The unique ID of the transactions or blocks on blockchain is a piece of data that can be easily integrated into the existing systems. Validation of the data can be implemented by smart contracts running on blockchain. An off-chain component is required to query the blockchain through using the ID of the data.

Consequences:

Benefits:

- *Connectivity.* The blockchain is integrated into an existing system through adding the ID of the transaction as a piece of data into the system, and using smart contracts to do data validation.

Drawbacks:

- *Non-intrusive.* It's not always possible to use blockchain in a non-intrusive way depending on the extensibility of the existing systems. Writing and reading blockchain might need changes to the existing system.

Related patterns: *Oracle* (Section **??**)

Known uses:

[8] https://en.bitcoin.it/wiki/Contract#Example_4:_Using_external_state
[9] http://orisi.org/
[10] https://gnosis.pm/

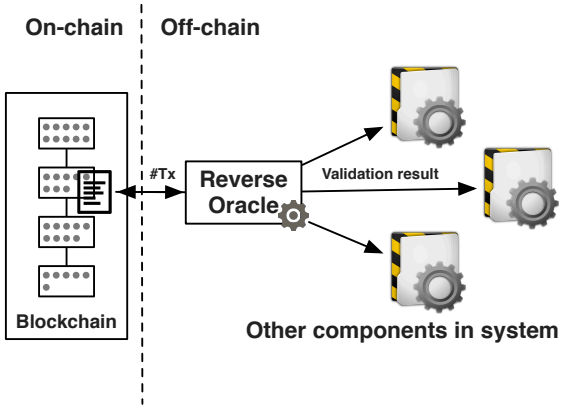

Figure 4: Reverse Oracle Pattern

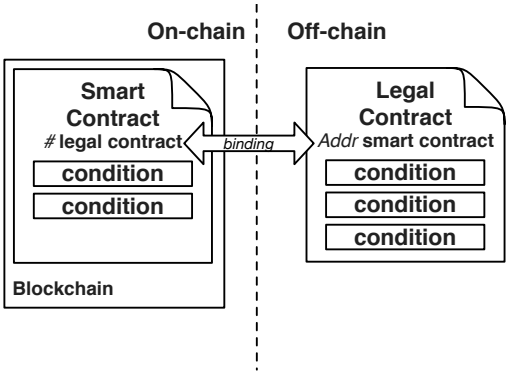

Figure 5: Legal and Smart Contract Pair Pattern

- *Identitii*[11] provides a solution to enrich the payments in banking systems with documents and attributes through using blockchain. Identitii invents the concept of identity token stored on a blockchain. Every payment is associated with an identity token, which is used to exchange enriched information about a payment. The identity token is exchanged between the banks through being embedded into the SWIFT protocol.
- *Slock.it*[12] is aimed to build autonomous objects and an universal sharing network through using blockchain and IoT devices, where the devices can sell or rent themselves, and also pay for services provided by others. In terms of renting a device, the availability information is stored on blockchain, thus, validity checking can be done on blockchain.

4.3 Pattern 3: Legal and Smart Contract Pair

Summary: A bidirectional binding is established between a legal agreement and the corresponding smart contract. Fig. **??** is a graphical representation of the pattern.

Context: The legal industry is becoming digitized, for example, using digital signatures has become a valid way to sign legal agreements. The Ricardian contract [**?**] was developed in the mid 1990s to interpret legal contracts digitally without losing the value of the legal prose. Digital legal agreements need to be executed and enforced.

Problem: An independent trustworthy execution platform trusted by all the involved participants is needed to execute the digital legal agreement.

How to bind a legal agreement to the corresponding smart contract on a trusted execution environment to ensure a 1-to-1 mapping?

Forces: The problem requires to balance the following forces:

- *Authoritative source.* A 1-to-1 mapping is required between a legal contract and its corresponding smart contract to make the smart contract as the authoritative source of the legal contract.
- *Secure storage.* Blockchain provides a trustworthy data storage to keep the legal agreement.
- *Secure execution.* Blockchain also provides a trustworthy computational platform that can execute digital agreements to enforce certain conditions as defined in a legal contract.

Solution:

Blockchain can be an ideal trusted platform to run digital legal agreements, which are bound with corresponding on-chain smart contracts. The smart contract implements conditions defined in the legal agreement. When deployed, there is a variable to store the hash value of the legal agreement, but is initially a blank value. The address of the smart contract is included in the legal agreement, and then the hash of the legal agreement is calculated and added to the contract variable. By binding a physical agreement with a smart contract, the bridge between the off-chain physical agreement and the on-chain smart contract is established. The two directional binding makes sure that the legal agreement and smart contract have a 1-to-1 mapping.

The smart contract digitizes the conditions defined within the legal agreement. Thus, these conditions can be checked and enforced automatically by the smart contract. However, not all the legal terms can be easily digitalized. The smart contract can also enable automated regulatory compliance checking in terms of the required information and process. However, the capability of compliance checking might be limited due to the constraints of smart contract programming language.

Consequences:

Benefits:

- *Automation.* Some of the conditions defined in the legal contract, for example, a conditional payment, can be automatically enforced by blockchain.
- *Audit trail.* Blockchain permanently records all historical transactions related to the legal contract and the contract itself. This immutable data enables auditing at anytime in future.

[11] https://identitii.com/
[12] https://slock.it/

- *Clarification.* Encoding legal terms expressed in natural language into smart contracts will require to give them a clear interpretation.

Drawbacks:

- *Expressiveness.* Smart contracts are written in programming languages. The smart contract languages might have limited expressiveness to express contractual terms of arbitrary complexity. The capability of regulatory compliance checking also depends on the expressiveness of the smart contracts. A regulation may regulate the process, for example, what should or should not be done by whom at what stage.
- *Enforceability.* If a public blockchain is used, there is no central administering authority to decide a dispute, or perform the enforcement of a court judgment.
- *Interpretation.* There might be different ways to interpret a certain legal term and to encode them in the smart contract. Ambiguity of natural language makes it a challenge to accurately implement a certain legal term in a way that is agreed upon by all the involved participants.

Related patterns: N/A

Known uses:

- *OpenLaw*[13] is a platform that allows lawyers to make legally binding and self-executable agreements on the Ethereum blockchain. The legal agreement templates are stored on a decentralized data storage, IPFS[14]. Users can create customized contracts for specific uses.
- *Smart Contract Template* proposed by Barclays[15] uses legal document templates to facilitate smart contracts running on Corda[16] blockchain platform [**? ?**].
- Specific proposals for the representation of machine-interpretable legal terms have been explored in KWM's project on digital and analog (*DnA*) contracts[17] and in the *Accord Project*[18].
- An academic work [**?**] uses a logic-based language to define smart contracts on blockchain.

5 DATA MANAGEMENT PATTERNS

This section discusses three data management patterns that manage data on and off blockchain.

5.1 Pattern 4: Encrypting On-Chain Data

Summary: Ensure confidentiality of the data stored on blockchain by encrypting it. Fig. **??** is a graphical representation of the pattern.

Context: For some applications on blockchain, there might be commercially critical data that should be only accessible to the involved participants. An example would be a special discount price offered by a service provider to a subset of its users. Such information should not be accessible to the other users who do not get the discount.

[13]http://openlaw.io/
[14]https://ipfs.io/
[15]https://www.barclays.co.uk/
[16]https://www.corda.net/
[17]https://github.com/KingandWoodMallesonsAU/Project-DnA
[18]https://www.accordproject.org/

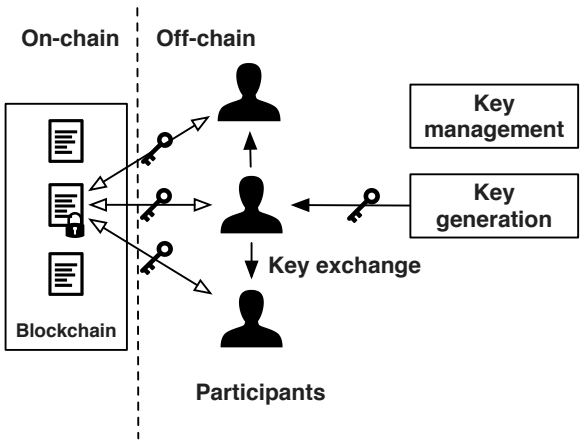

Figure 6: Encrypting On-chain Data Pattern

Problem: The lack of data privacy is one of the main limitations of blockchain. All the information on blockchain is publicly available to the participants of the blockchain. There is no privileged user within the blockchain network, no matter the blockchain is public, consortium or private. On a public blockchain, new participants can join the blockchain network freely and access all the information recorded on blockchain. Any confidential data on public blockchain is exposed to the public.

Forces: The problem requires to balance the following forces:

- *Transparency.* Every participant within a blockchain network is able to access all the historical transactions on blockchain, which is required to enable them to validate previous transactions. The transactions on a public blockchain are also accessible to everyone with access to the internet, simply using tools like a blockchain explorer such as Etherscan[19].
- *Lack of confidentiality.* Since all the information on blockchain is publicly available to everyone in the network, commercially sensitive data meant to be kept confidential should not be stored on blockchain, at least not in plain form.

Solution: To preserve the privacy of the involved participants, symmetric or asymmetric encryption can be used to encrypt data before inserting the data into blockchain. One possible design for sharing encrypted data among multiple participants is as follows. First, one of the involved participants creates a secret key for encrypting data and distributes it during an initial key exchange. When one of the participants needs to add a new data item to the blockchain, they first symmetrically encrypt it using the secret key. Only the participants allowed to access the transaction have the secret key and can decrypt the information.

Consequences:

Benefits:

- *Confidentiality.* Using encryption, the publicly accessible information on blockchain is encrypted, so that is useless to anyone who does not hold the secret key.

[19]http://etherscan.io

Drawbacks:

- *Compromised key.* Both symmetric and asymmetric encryption require off-chain key management. If key management is not done properly, it can lead to compromise and disclosure of private or secret keys. If the required private key or secret key is compromised, the encryption mechanism does not guarantee the confidentiality nor the integrity of the data.
- *Access revocation.* Revoking read access is a challenge after the encrypted data has been published to the blockchain. The encrypted data on blockchain is immutable. Thus, as long as the participant keeps the secret key used to encrypt the data, it has access to the encrypted data forever.
- *Immutable data.* Even if stored in encrypted form, the critical data will remain in the blockchain forever. In addition to the risk of key compromise, the encrypted data may be subject to brute force decryption attacks at any time in the future, or breakthroughs in technology like quantum computing might render current encryption technologies ineffective. So even if the data is considered to be secure with a given key size when it is stored in the blockchain, this may no longer be the case in the future.
- *Key sharing.* The encryption key needs to be shared off-chain before submitting any relevant transaction to the blockchain secretly. Although blockchain can be used as a software connector [**?**] to communicate data, secret keys can not be shared through blockchain because the shared key would be publicly accessible if being communicated through blockchain.

Related patterns: *Off-Chain Data Storage* (Section **??**)

Known uses:

- Encrypted queries from *Oraclize*[20]. Oraclize is a smart contract running on Ethereum public blockchain, which provides a service to access state from external world. Oraclize allows smart contract developers to encrypt the parameters of their queries locally by using a public key before passing them to a smart contract. The only one who can decrypt the call parameters is Oraclize with the paired private key.
- *Crypto digital signature* suggested by *MLGBlockchain*[21] to encrypt data and share the data between the parties who interact and transmit data through blockchain.
- Hawk [**?**] is a smart contract system that stores transactions as encrypted data on blockchain to retain the privacy of the transactions. The compiler of Hawk can automatically generate a cryptographic protocol for a smart contract. The involved participants interact with the blockchain following the cryptographic protocol.

5.2 Pattern 5: Tokenisation

Summary: Using tokens to represent fungible goods for easier distribution.

Context: The concept of tokenisation has emerged centuries ago with the first currency systems. Tokenisation is a means to reduce risk in handling high value financial instruments by replacing them with equivalents, for example, the tokens used in casino. Tokens can represent a wide range of goods which are transferable and fungible, like shares, or tickets.

Problem: Tokens representing assets should be the authoritative source of the corresponding assets.

Forces: The problem requires to balance the following forces:

- *Risk.* Handling fungible financial instruments with high value is risky, e.g., lost tokens cannot be replaced.
- *Authority.* Tokens should be the authoritative source of the assets.

Solution: Blockchain provides a trustworthy platform to realise tokenisation. There are different ways to implement tokenisation using blockchain. Naive tokens on a blockchain (*e.g.*, BTC on Bitcoin, ETC on Ethereum) can be used to formulate a system where the tokens represent monetary value or other physical assets. The token is generally used to track title over the physical assets. Transactions on blockchain record the verifiable title transfer from one user to another. However, using the native token on blockchain for tokenisation is limited because it can only implement the title transfer of the physical assets, with limited conditions checking.

A more flexible way is to define a data structure in a smart contract to represent physical assets. Tokenisation is a process starting from an asset (*e.g.*, money) is locked under a custody (*e.g.*, a bank), and gets represented in the cryptographic world through a token. The ownership of the digital token matches the ownership of the corresponding asset. The reverse process can take place by which the user redeems the token to recover the value which is sitting within the bank. A token on blockchain is the authoritative source of the physical asset. By using smart contracts, some conditions can be implemented and associated with the ownership transfer.

Consequences:

Benefits:

- *Risk.* Tokenisation reduces risk in handling high value financial instruments by replacing them with equivalents.
- *Authority.* Blockchain and smart contracts provide a trustworthy infrastructure to provide authorised tokens for the corresponding assets.

Drawbacks:

- *Integrity.* Integrity of the tokens is guaranteed by the blockchain infrastructure. But the authenticity of the corresponding physical/digital asset is not guaranteed automatically.
- *Standardisation.* 24% of the existing financial smart contracts on Ethereum uses this tokenisation pattern . Given the popularity of this pattern, ERC20[22] (and ERC777[23] as an advanced version) has been proposed as a fungible token standard that describes the functions and events that a token smart contract has to implement. The new proposed fungible tokens should follow the standard.
- *Legal processes for ownership.* A token on a blockchain is not necessarily the authoritative source of information about the ownership of a physical asset. The owner of an asset may

[20] https://blog.oraclize.it/encrypted-queries-private-data-on-a-public-blockchain-71d893fac2bf
[21] https://mlgblockchain.com/crypto-signature.html
[22] https://theethereum.wiki/w/index.php/ERC20_Token_Standard
[23] https://eips.ethereum.org/EIPS/eip-777

be entitled to sell the asset without being required to create a transaction on the blockchain. Also, legal processes such as court orders and bankruptcy proceedings can change the ownership of physical assets without any associated transaction being recorded on the blockchain.

Related patterns: N/A

Known uses:

- *Coloredcoin*[24] is an open source protocol for tokenizing digital assets on Bitcoin blockchain.
- *Digix*[25] uses tokens to track the ownership of gold as a physical property.

5.3 Pattern 6: Off-Chain Data Storage

Summary: Use hashing to ensure the integrity of arbitrarily large datasets which may not fit directly on the blockchain. Fig. **??** is a graphical representation of the pattern solution.

Context: Some applications consider using the blockchain to guarantee the integrity of large amounts of data.

Problem: The blockchain, due to its full replication across all participants of the blockchain network, has limited storage capacity. Storing large amounts of data within a transaction may be impossible due to the limited size of the blocks of the blockchain (for example, Ethereum has a block gas limit to determine the number, computational complexity, and data size of the transactions included in the block). Data cannot take advantage of the immutability or integrity guarantees without being stored on the blockchain.

How to store data of arbitrary size and take advantage of the immutability and integrity guarantees provided by the blockchain?

Forces: The problem requires to balance the following forces:

- *Scalability.* Blockchain provides limited scalability because every bit of data is replicated across all nodes, where it is kept permanently.
- *Cost.* If a public blockchain is used, storing data on blockchain costs real money, although the cost is a one-time cost to write the data. This is in contrast to traditional distributed data storage, like cloud, which charge based on the amount of allocated storage space over time. A piece of data can be stored on blockchain through being embedded into a transaction, or as a variable of smart contract or as a log event. Embedding data into a transaction is the cheapest way, while storing data in a contract is more efficient to enable manipulation, but can be less flexible due to the potential constraints of the smart contract languages on the value types and length [**?**]. Different blockchain has different cost model for storing data.
- *Size.* There are limits of transaction size or block size. For example, on Bitcoin blockchain, The default Bitcoin client only relayed *OP_RETURN* transactions up to 80 bytes, which was reduced to 40 bytes in February 2014[26]. Ethereum has a block gas limit that restricts the amount of gas which all transactions in a block are allowed to use.

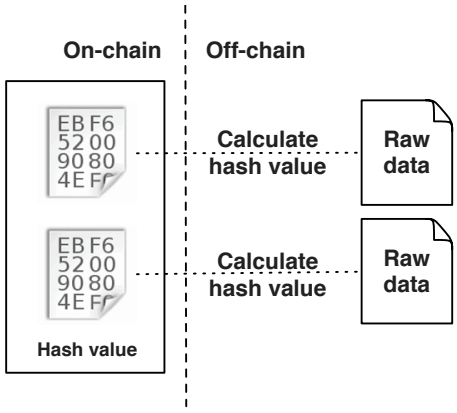

Figure 7: Off-chain Data Storage Pattern

Solution: The blockchain can be used as a general-purpose replicated database, as transactions logged in the blockchain can include arbitrary data on some blockchain platforms. For data of big size (essentially data that is bigger than its hash value), rather than storing the raw data directly on blockchain, a representation of the data with smaller size can be stored on blockchain with other small sized metadata about the data (*e.g.*, a URI pointing to it). The solution is to store a hash value (also called digest) of the raw data on chain. The value is generated by a hash function, *e.g.* one from the SHA-2 [**?**] family, which maps data of arbitrary size to data of fixed size. Hash function is a one-way function which is easy to compute, but hard to invert given the output of a random input. If even one bit of the data changes, its corresponding hash value would change as well. The hash value is used for ensuring the integrity of the raw data stored off-chain, and the transaction on blockchain that includes the hash value guarantees the integrity of the hash value as well as the original raw data from which the hash was derived.

Consequences:

Benefits:

- *Integrity.* Blockchain guarantees the integrity of the hash value that represents the raw data. The integrity of the raw data can be checked using the on-chain hash value.
- *Cost.* If a public blockchain is used, blockchain is utilized at a lower cost (fixed cost as the size of the hash value is fixed) for integrity of data with arbitrary size.

Drawbacks:

- *Integrity.* The raw data is stored off-chain, where the off-chain data store might not be as secure as blockchain. The raw data may be changed without authorization. This change will be detected thanks to the hash of the original data stored on the blockchain. However, without additional measures, it will neither be possible to recover the original data nor to prevent the change from happening in the first place.

[24] http://coloredcoins.org/
[25] https://digix.global/
[26] https://github.com/bitcoin/bitcoin/pull/3737

- *Data loss.* Since the raw data is stored off-chain, it may be deleted or lost. Only its hash value remains permanently on the blockchain.
- *Data sharing.* The on-chain data can be shared through using blockchain platforms. Extra communication mechanisms and storage platforms are required for data sharing off-chain.

Related patterns: *Proxy* from [**?**]

Known uses:

- *Proof-of-Existence (POEX.IO*[27]*).* This service allows entering an SHA-256 cryptographic hash of a document into the Bitcoin blockchain as a "proof-of-existence" of the document at a certain time. The hash value guarantees the data integrity of the document.
- *Chainy*[28]. This is a smart contract running on Ethereum blockchain. Chainy stores a short link to an off-chain file and its corresponding hash value in one place.

5.4 Pattern 7: State channel

Summary: Micro-payments transactions are too expensive to be performed on-chain because the required transaction fee might be higher than the monetary value associated with the transaction assuming a public blockchain is used. Thus, micro-payments should be exchanged off-chain while periodically recording settlements for larger amounts on chain. Such a payment channel can be generalized for arbitrary state updates for more general purposes other than monetary value. Fig. **??** is a graphical representation of the pattern.

Context: Micro-payments are payments that can be as small as a few cents, *e.g.*, payment of a very small amount of money to a WiFi hot-spot for every 10 kilobytes of data usage. Blockchain has potential to be used for such financial transactions with tiny monetary value. The question is if it is necessary and cost effective to store all the micro-payment transactions on blockchain.

Problem: The decentralized design of blockchain has limited performance. Transactions can take several minutes or even one hour (for Bitcoin blockchain) to be *committed* on the blockchain [**?**]. Due to the long commit time and high transaction fees on a public blockchain (where fees are largely independent of the transacted amount), it is often infeasible to store every micro-payment transaction on the blockchain network. During a recent peak in demand, the average fee per transaction has risen to the equivalent of US$55[29] on Bitcoin. On-chain transactions are suitable for transactions with medium to large monetary value, relative to the transaction fee.

Forces: The problem requires to balance the following forces:

- *Latency.* Blockchain transactions may take a long time to be committed while users expect micro-payments to happen instantaneously.
- *Scalability.* Blockchain has limited scalability because every bit of data is replicated across all nodes, and kept permanently.
- *Cost.* Storing data on a public blockchain costs real money. The transaction fee of individual micro-payment transaction might be higher than the monetary value associated with the micro-payment transaction.

Solution: Storing every micro-payment transaction on blockchain is infeasible in certain contexts due to the small monetary value associated with it. Thus, a solution is to establish a payment channel between two participants, with a deposit from one or both sides of the participants locked up as security in a smart contract for the lifetime of the payment channel. The payment channel keeps the intermediate states of the micro-payment off-chain, and only stores the finalized payment on chain. The frequency of transaction settlement depends on the use case, and agreement between the two sides. For example, in scenarios around utilities, internet service providers or electricity companies can establish payment channel with their consumers for an agreed billing period, for example, a month. As the consumer uses data or energy daily, the intermediate state is stored in the channel until the end of the month, when the channel is closed to finalize the payment of that month. A network of micro-payment channels can be built where the transactions transferring small values occur off-chain. The individual transactions take place entirely off the blockchain and exclusively between the participants, across multiple hops where needed. Only the final transaction that settles the payment for a given channel or set of channels is submitted to the blockchain. The technologies used to implement state channel are specific to blockchain platform. For example, Lightning network[30] on the Bitcoin blockchain is a proposed implementation of Hashed Timelock Contracts (HTLCs)[31] with bi-directional payment channels which allows secure payments across multiple peer-to-peer channels. A HTLC is a type of payments that use the features of Script, like *hashlocks* and *timelocks*, to require that the receiver of a payment acknowledges receiving the payment prior to a deadline by generating cryptographic proof.

Consequences:

Benefits:

- *Speed.* Without involving blockchain for every transfer, the off-chain transactions can be settled without waiting for the blockchain network to process the transaction, generate a new block with the transaction and reach consensus, and the desired number of confirmation blocks.
- *Throughput.* The number of off-chain transactions that can be processed is not limited by the configuration of blockchain, such as the block size, block interval, gas limit, etc., and thus a much higher throughput can be achieved than for on-chain transactions.
- *Privacy.* Other than the final settlement transaction, the individual off-chain transactions do not show up in the public ledger, thus, the detail of these intermediate off-chain transactions is not publicly visible.

[27] https://poex.io/
[28] https://chainy.info/
[29] Recorded by https://bitinfocharts.com/comparison/bitcoin-transactionfees.html for 22 Dec 2017; accessed on 1/2/2018.
[30] https://lightning.network/
[31] https://en.bitcoin.it/wiki/Hashed_Timelock_Contracts

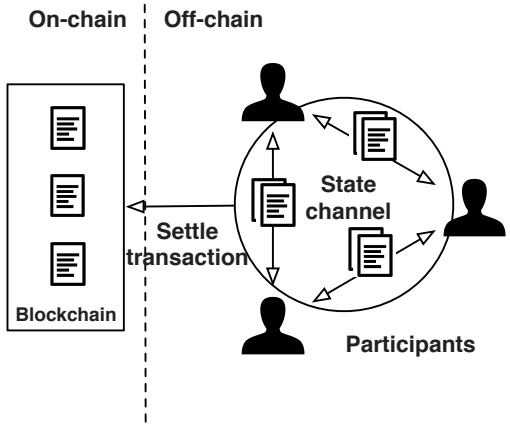

Figure 8: State Channel Pattern

- *Cost.* If a public blockchain is used, only the final settlement transaction costs transaction fee to be included in the blockchain. Direct individual off-chain transactions do not cost any money. Multi-hop transactions may cost small transaction fees, which are typically charged as a percentage of the transacted amount.

Drawbacks:

- *Trustworthiness.* The individual off-chain micro-payment transactions might not be as trustworthy as the on-chain transactions because the micro-payment transactions are not stored in an immutable data store. The intermediate state of payment channels might be lost after the payment channels are closed.
- *Reduced liquidity.* To establish a payment channel, money from one or both sides of the channel needs to be locked up in a smart contract for the lifetime of the payment channel. The liquidity of the channel participants is thereby reduced.
- *Wallet.* A new wallet or extension to the existing wallet is needed to support the micro-payment protocol.

Related patterns: *Off-chain signatures* from [**?**]

Known uses:

- The *Lightning network* uses an off-chain protocol to enable micro-payments of Bitcoin and several other cryptocurrencies. Micro-payments are enabled by establishing a bidirectional payment channel through committing a funding transaction to the blockchain. This can be followed by a number of micro-payment transactions that update the distribution of the funds within the channel without broadcasting transactions to the blockchain network. The payment channel can be closed by broadcasting the final version of the funding transaction to settle the payment.
- The *Raiden network*[32] on the Ethereum blockchain is a similar solution as lightning network. The basic idea is to avoid the consensus bottleneck by leveraging a network of off-chain payment channels that allow to securely transfer monetary value. Smart contracts are used to deposit value into the payment channels.
- *Orinoco*[33] is another payment channel solution built on Ethereum blockchain. Other than the payment channels, Orinoco also provides a payment hub for payment channel management. However, the payment hub introduces an extra party that needs to be trusted by both the sender and the recipient of the payment channel.
- *State channel* on Ethereum[34] and *Gnosis Go*[35] offer a more generalized form of state channels that support exchanging state for general-purpose applications.

6 SECURITY PATTERNS

This section discusses three security patterns that mainly concern the security aspect of the blockchain-based applications.

6.1 Pattern 8: Multiple Authorization

Summary: A set of blockchain addresses which can authorise a transaction is pre-defined. Only a subset of the addresses is required to authorize transactions. Fig. **??** is a graphical representation of the pattern.

Context: In blockchain-based applications, activities might need to be authorized by multiple blockchain addresses. For example, a monetary transaction may require authorization from multiple blockchain addresses.

Problem:

- The actual addresses that authorize an activity might not be able to be decided due to the availability of the authorities.

Forces: The problem requires to balance the following forces:

- *Flexibility.* The actual authorities who authorize the transaction can be from a set of pre-defined authorities.
- *Tolerance of compromised or lost private key* Authentication on blockchain uses digital signature. However, blockchain does not offer any mechanism to recover a lost or a compromised private key. Losing a key results in permanent loss of control over an account, and potentially smart contracts that refer to it.

Solution: It would enable more dynamism if the set of blockchain addresses for authorization are not decided before the corresponding transaction being submited into the blockchain network, or the corresponding smart contract being deployed on blockchain. On the Bitcoin blockchain, a multi-signature mechanism can be used to require more than one private key to authorize a Bitcoin transaction. In Ethereum, smart contract can mimic multi-signature mechanism. More flexibly, an M-of-N multi-signature can be used to define that M out of N private keys are required to authorize the transaction. M is the threshold of authorization. This on-chain mechanism enables more flexible binding of authorities.

[32] https://raiden.network/

[33] www.orinocopay.com/

[34] http://www.jeffcoleman.ca/state-channels/

[35] https://forum.gnosis.pm/t/how-offchain-trading-will-work/63

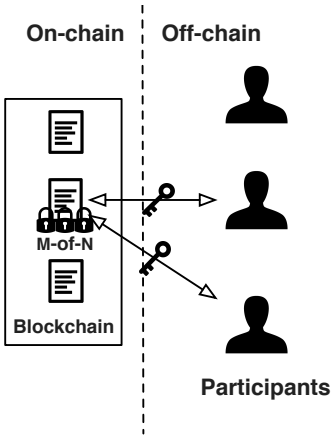

Figure 9: Multiple Authorization Pattern

Consequences:

Benefits:

- *Flexibility.* This pattern enables flexible binding of authorities, but depends on the availability of authorities when the activity is proceeded.
- *Lost key tolerant.* One participant can own more than one blockchain address to reduce the risk of losing control over their smart contracts due to a lost private key. There could be a function that can update the list of allowed authorities, and the threshold of the authorization. In the case that the update function also requires threshold-based authorization, the list of the update addresses can be also updated through authorization from at least the minimum number of addresses.

Drawbacks:

- *Pre-defined authorities.* Although the pattern enables flexible binding, all the possible authorities still need to be known in advance of any decision or update.
- *Lost key.* At least M private keys among the N private keys should be safely kept to avoid losing control.
- *Cost of dynamism.* If a public blockchain is used, updating the list of authorities costs money, as does deploying the logic for multiple authorities. Besides, it costs more to store multiple addresses as the possible authorities than storing only one.

Related patterns: *Off-chain Secret Enabled Dynamic Authorization* (Section **??**). An off-chain secret enabled dynamic authorization pattern is used when the possible authorities are unknown beforehand.

Known uses:

- MultiSignature mechanism provided by Bitcoin[36].
- Multisignature wallet, written in Solidity, running on Ethereum blochchain and is available in the Ethereum DApp browser Mist[37].

[36] https://en.bitcoin.it/wiki/Multisignature
[37] https://github.com/ethereum/mist

6.2 Pattern 9: Off-Chain Secret Enabled Dynamic Authorization

Summary: Using a hash created off-chain to dynamically bind authority for a transaction. Fig. **??** is a graphical representation of the pattern. This solution is also referred to as *Hashlock*.

Context: In blockchain-based applications, some activities need to be authorized by one or more participants that are unknown when a first transaction is submitted to blockchain.

Problem: Sometimes, the authority who can authorize a given activity is unknown when the corresponding smart contract is deployed, or the corresponding transaction is submitted to the blockchain. Blockchain uses digital signature for authentication and transaction authorization. Blockchain does not support dynamic binding with an address of a participant which is not defined in the respective transaction or smart contract. All accounts that can authorize a second transaction have to be defined in the first transaction before that transaction is added to the blockchain.

Forces: The problem requires to balance the following forces:

- *Dynamism.* Dynamic binding one or more unknown authorities with a second transaction representing an activity after the first transaction submitted to blockchain.
- *Pre-defined authorities.* Using only on-chain mechanisms, all the possible authorities are required to be defined beforehand.

Solution: An off-chain secret can be used to enable a dynamic authorization when the participant authorizing a transaction is unknown beforehand. In the context of payment, for example, a smart contract can be used as an escrow. When the sender deposits the money to an escrow smart contract, a hash of a secret (*e.g.* a random string, called pre-image) is submitted with the money as well. Whoever receives the secret off-chain can claim the money from the escrow smart contract by revealing the secret. With this solution, the receiver of the money does not need to be defined beforehand in the escrow contract. This can be generalized to any transaction that needs authorization from a dynamically bound participant. Note that since the secret is revealed, it cannot be reused. One variant is to lock multiple transactions with the same secret – by unlocking one, all of them are unlocked.

Consequences:

Benefits:

- *Dynamism.* This pattern enables dynamic binding of unknown authorities after the transaction is added into the blockchain.
- *Lost key tolerant.* No specific private key is required to authorize transactions.
- *Routability.* This pattern has the useful property that once the secret is revealed, any other transactions secured using the same secret can also be opened. This makes it possible to create multiple transactions that are all locked by the same secret. This property is used by micro-payment channels to enable multi-hop transfers where the money hosted by every hop and secured by a same secret can be released after the end receiver claims the money with the secret (*i.e.* the

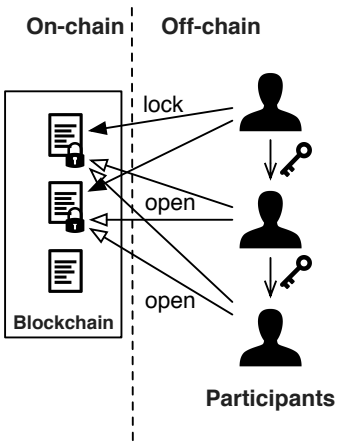

Figure 10: Off-chain Secret Enabled Dynamic Authorization Pattern

secret is revealed). The secret can be exchanged through an off-chain channel to every hop.
- *Interoperability.* There is no need for a special protocol to exchange the secret. The secret can be exchanged in any ways off-chain. It provides a mechanism for other systems to trigger events on blockchain.

Drawbacks:

- *One-off secret.* The secret used in this pattern is a one-off secret. Verification of the secret is on-chain. Thus, once a secret is embedded in a transaction submitted to the blockchain, the secret is revealed.
- *Combination of signature and secret.* Because this pattern has the property that once the secret is revealed, any other transactions secured using the same secret can also be opened, sometimes the transaction protected by the secret should also be associated with a public key so that both a correct secret and an appropriate signature with the respective private key are required to authorize the transaction. This is applicable to the situation where a large set of authorities are known beforehand, but not all of them are allowed to authorize a certain activity/transaction. Thus, a hash secret is used to dynamically bind one or multiple authorities from the larger pre-defined set of authorities.
- *Lost secret.* The sender/initiator of a transaction takes the risk of losing the off-chainsecret. If the secret is lost, the transaction cannot be authorized and being proceeded anymore. In the case of money transfer, the money associated with the transaction would be locked forever if the transaction cannot be authorized properly.

Related patterns: *Multiple authorization* (Section **??**). The multiple authorization pattern is used when all the possible authorities are known beforehand. Multiple authorization pattern is an on-chain mechanism.

Known uses:

- *Raiden network*[38] is a network of off-chain payment channels on top of Ethereum blockchain network, which enables secure value transfer. The multi-hop transfer mechanism in Raiden Network uses *hashlocked* transactions to securely router payment through a middleman.
- In the Bitcoin ecosystem, *atomic cross-chain trading*[39] allows one crytocurrency (for example, Bitcoin) to be traded for another cryptocurrency (for example, tokens on a Bitcoin sidechain) using a off-chain hash secret.

6.3 Pattern 10: X-Confirmation

Summary: Waiting for enough number of blocks as confirmations to ensure that a transaction added into blockchain is immutable with high probability. Fig. **??** is a graphical representation of the pattern.

Context: Immutability of a blockchain using Proof-of-work (Nakamoto) consensus is probabilistic immutability. There is always a chance that the most recent few blocks are replaced by a competing chain fork.

Problem: At the time a fork occurs, there is usually no certainty as to which branch will be permanently kept in the blockchain and which branches will be discarded. The transactions that were included in the branches being discarded eventually go back to the transaction pool and being added into a later block.

Forces: The problem requires to balance the following forces:

- *Chain fork.* Chain fork may occur on a blockchain using proof-of-work consensus, like Bitcoin and Ethereum.
- *Frequency of chain fork.* Transaction handling and inter-block time differs significantly from one blockchain to another. A shorter inter-block time would lead to an increased frequency of forks .

Solution: From the application perspective, one security strategy is to wait for a certain number (X) of blocks to be generated after the transaction is included into one block. After X blocks, the transaction is taken to be *committed* and thus perceived as immutable [**?**]. The value of X can be decided by the developers of the blockchain-based applications.

Consequences:

Benefits:

- *Immutability.* The more blocks being generated after the block including the transaction, the higher probability of the immutability of the transaction.

Drawbacks:

- *Latency.* Latency between submission and confirmation of a transaction has been included on a blockchain is affected by the consensus protocol and the X value of X-confirmation. For example, this is around 1 hour (10 minute block interval with 6-confirmation) on Bitcoin. The larger value of the X, the longer the latency.

Related patterns: N/A

[38] https://raiden.network/
[39] https://en.bitcoin.it/wiki/Atomic_cross-chain_trading

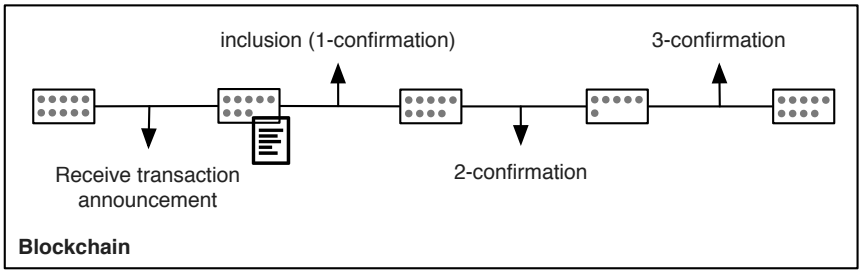

Figure 11: X-Confirmation Pattern

Known uses:

- *Bitcoin* uses 6-confirmation. The value 6 of Bitcoin blockchain was chosen based on the assumption that an attacker is unlikely to amass more than 10% of the total amount of computing power within Bitcoin network (measured by *hash rate*[40]), and that a negligible risk of less than 0.1% is acceptable[41].
- *Ethereum* recommends to wait for 11 confirmations after block inclusion before assuming that a transaction is committed permanently with high probability[42].

7 CONTRACT STRUCTURAL PATTERNS

This section discusses five smart contracts patterns. Essentially, smart contracts are programs running on blockchain, thus some of the existing design patterns and programming principles for conventional software environments are also applicable to smart contracts. If a public blockchain is used, the structural design of the smart contract has large impact on its execution cost. The cost of deploying a smart contract depends on the size of the smart contract(s) because the code is stored on blockchain, resulting in a data storage fee that is proportional to the size of the smart contract. Thus, a structural design with more lines of compiled code costs more money. A consortium blockchain does not necessarily have tokens/currency; therefore monetary cost is typically not an issue for a consortium blockchain. However, blockchain size is still a design concern because the total size of the blockchain keeps growing as more blocks are appended to it and no block can ever be detached from it, and every participant stores a full replica of blockchain. Besides, different structural designs of smart contracts may affect performance because more or less transactions may be required.

7.1 Pattern 11: Contract Registry

Summary: Before invoking it, the address of the latest version of a smart contract is located by looking up its name on a contract registry. Fig. **??** is a graphical representation of the pattern.

Context: As any software application, blockchain-based applications need to be upgraded to new versions. To do so, the on-chain functions defined in smart contracts need to be updated to fix bugs as well as to fulfil new requirements.

Problem: Smart contracts deployed on blockchain cannot be upgraded because the code of the smart contracts as a type of data, stored on blockchain is immutable.

Forces: The problem requires to balance the following forces:

- *Immutability.* Every bit of data, including deployed smart contracts, stored on blockchain is immutable.
- *Upgradability.* There is a fundamental need to upgrade all but short-lived applications and their smart contracts over time.
- *Human-readable contract identifier.* The identifier of a smart contract on blockchain platforms, like Ethereum, is hexadecimal address, which is not human-readable.

Solution: An on-chain registry contract is used to maintain a mapping between user-defined symbolic names and the blockchain addresses of the registered contracts. The address of the registry contract needs to be advertised off-chain. The creator of a contract can register the name and the address of the new contract to the registry contract after the new contract being deployed. The invoker of a registered contract retrieves the latest version of the new smart contract from the registry contract. The corresponding functions provided by the registered contract can be upgraded by replacing the address of the old version contract in the registry contract with the address of a new version without breaking the dependency between the upgraded smart contract and other smart contracts that depend on its functions. The address of a contract is stored as a variable in the registry contract. The value of contract variables can be updated. The registry contract can have a permission control module to maintain the writing permission. Note that all the previous values of the variable are still stored on the blockchain.

Consequences:

Benefits:

- *Human-readable contract name.* The registry contract maintains a mapping between human-readable names and the hexadecimal addresses of the smart contracts. A human readable form of smart contract names is desired, for example, to be exposed to the user interface. A human readable name is also useful for developers.
- *Constant contract name.* The smart contract associated with a registered name can be updated without changing its name. This way dependencies relying on the name of the smart contract do not get broken.

[40] https://blockchain.info/charts/hash-rate
[41] https://en.bitcoin.it/wiki/Confirmation
[42] https://blog.ethereum.org/2015/09/14/on-slow-and-fast-block-times/

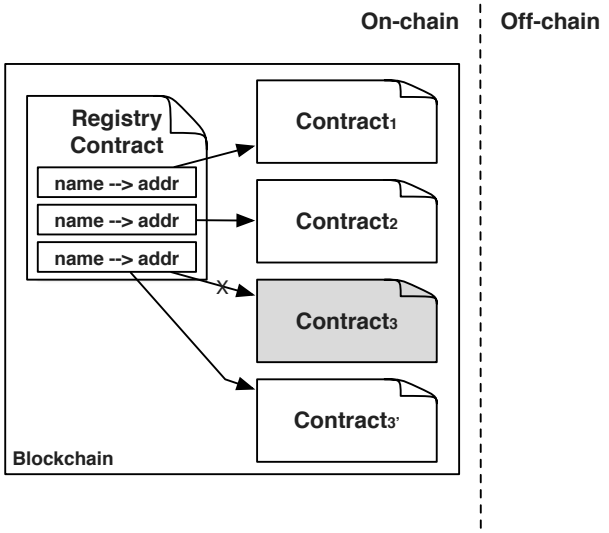

Figure 12: Contract Registry Pattern

- *Transparent upgradability.* The smart contract associated with a registered name could be replaced by a new version without breaking the dependencies based on the human-readable name.
- *Version control.* Version control can be integrated in the registry contract as well to allow a look-up based on the name and version of a smart contract. Old versions of a smart contract that are no longer needed should be terminated.

Drawbacks:

- *Limited upgradability.* Upgradability is still limited if the functions defined in the smart contract are called by other contracts. Although the implementation of the function can be upgraded, the interface (that is function signature) cannot be modified without breaking the link to dependent smart contracts. Similar methods as for API / service interface management need to be implemented, *e.g.* through versioning and depreciation flags.
- *Cost.* There is an additional cost to maintain a registry that contains the mapping between the contract names and their addresses. Furthermore, all the inter-contract function calls require a registry look-up to find the latest version of the smart contract to be invoked.

Related patterns: *Embedded permission* (Section refsec:permission) can be used to define writing permission. *Data contract* (Section **??**) and this pattern can work together to further improve upgradability of smart contracts.

Known uses:

- ENS[43] is a name service on Ethereum blockchain, which is implemented as smart contracts. ENS maintains a mapping between both smart contracts on-chain and resources off-chain and simple, human-readable names. ENS can be viewed as a contract registry built in a blockchain platform, which is accessible to everyone. A blockchain-based application can also maintain a separate registry contract for the application.
- Regis[44] is an in-browser application that makes it easy to build, deploy and manage registries as smart contracts on Ethereum blockchain. It allows user-defined key-value pairs. It can be used to create a contract registry.

7.2 Pattern 12: Data Contract

Summary: Store data in a separate smart contract. Fig. **??** is a graphical representation of the pattern.

Context: The need to upgrade a blockchain-based application over time is ultimately necessary, so as the smart contracts used by the application. In general, logic and data change at different times and with different frequencies. There are different ways to store a data on blockchain, as discussed in *Hash Integrity* pattern (Section **??**).

Problem: Storing data on blockchain is expensive and there is a limitation on the amount of data and amount of computation a transaction can contain. In the context of upgrading smart contracts, the upgrading transactions might contain a large data storage for copying the data from the old version of the smart contract to the new version of the smart contract. Porting data to a new version might even require multiple transactions, *e.g.* when the block gas limit on Ethereum prevents an overly complex data migration transaction.

Forces: The problem requires to balance the following forces:

- *Coupling.* Smart contracts can live forever on blockchain if not being explicitly terminated. If a smart contract is deactivated in this way, the data stored in the smart contract cannot be accessed through the smart contract functions any more – although it can still be accessed with some effort, *e.g.* for provenance or audit purposes.
- *Upgradability.* The need to upgrade the application and the smart contracts supporting the application over time is ultimately necessary for many applications.
- *Cost.* If a public blockchain is used, storing data on blockchain costs money. Thus copying data from an old version of a smart contract to a new version should be avoid or minimized.

Solution: To avoid moving data during upgrades of smart contracts, the data store is isolated from the rest of the code. In the context of blockchain, data could be separately stored in different smart contracts to enable isolation. Depending on the circumstances of the application, how large of a data store it needs and whether the data structure is expected to change often, the data store could use a strict definition or a loosely typed flat store. The more generic and flexible data structure can be used by all the other logic smart contracts and is unlikely to require changes. One example of a generic data structure is a mapping to store *SHA3* key and value pairs.

Consequences:

Benefits:

[43] https://ens.domains

[44] https://regis.nu/

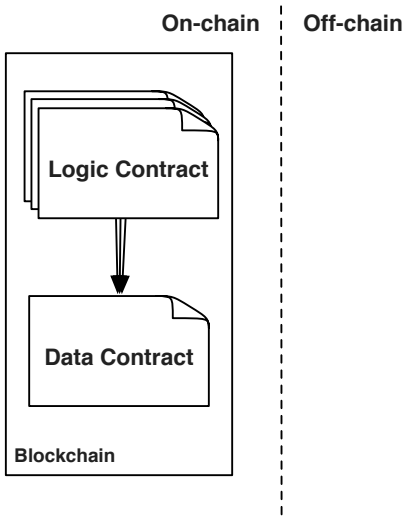

Figure 13: Data Contract Pattern

- *Upgradability.* By separating data from the rest of the code, the logic of the application is able to be upgraded without affecting the data contract.
- *Cost.* Since the data is separated from the rest of the code, there is no cost for migrating data when the application is upgraded.
- *Generality.* If the data can be cleanly separated and generalized, there would be an additional benefit: the generic data contract can be used by all related logic smart contracts.

Drawbacks:

- *Cost.* If a public blockchain is used, storing a piece of data in a generic data structure costs more money than a strictly defined data structure. For example, as mentioned earlier, a generic data structure maintains a mapping between *SHA3* key and value pairs, but a more strictly defined data structure can be of smaller size, *e.g.* not requiring the key to be stored. Querying the data is also less straightforward. This is the cost of a generalized solution.

Related patterns: *Contract registry* (Section **??**) and this pattern can work together to further improve upgradability of smart contracts.

Known uses:

- *Chronobank*[45] is a blockchain project that tokenizes labour and provides a market for professionals to trade their labour time with businesses. It uses a smart contract with a generic data structure as the data store used by all the other logic smart contracts.
- *Colony*[46], a platform for open organizations running on Ethereum. Similar to Chronobank, Colony has a data contract with a generic data structure.

[45] https://chronobank.io/
[46] https://colony.io/

7.3 Pattern 13: Embedded Permission

Summary: Smart contracts use an embedded permission control to restrict access to the invocation of the functions defined in the smart contracts. Fig. **??** is a graphical representation of the pattern.

Context: All the smart contracts running on blockchain can be accessed and called by all the blockchain participants and other smart contracts by default, because there are no privileged users and, in the case of public blockchain, every participant can join the network to access all the information and code stored and running on blockchain.

Problem: A smart contract by default has no owner, meaning that once deployed the author of the smart contract has no special privilege on the smart contract. A permission-less function can be triggered by unauthorized users accidentally. Such a permission-less function becomes vulnerability of blockchain-based application. For example, a permission-less function which is discovered in a smart contract library used by the Parity multi-sig wallet, caused the freezing of about 500K Ethers[47]. 7% smart contract on public Ethereum can be terminated without authority [**?**].

Forces: The problem requires to balance the following forces:

- *Security.* The functions defined in the smart contracts should be called only by the authorized participants. Due to the transparency of public blockchains, all the smart contracts are also publicly available to everyone connecting to the Internet. In contrast, in a conventional software system, the internal logic is normally not visible to the end uses. Interaction with the software system is either through a user interface or API, where it is possible to enforce access control policies.

Solution: Adding permission control to every smart contract function to check permissions for every caller that triggers the functions defined in the smart contract based on the blockchain addresses of the caller. This can be done by checking the authorization of the caller before executing the logic of the function: unauthorized calls are rejected and the execution of the function terminated before reaching the core logic of the function.

Consequences:

Benefits:

- *Security.* Only the participants and smart contracts that are authorized by the smart contract can call the corresponding functions successfully.
- *Secure authorization.* Authorization is implemented in smart contracts running on blockchain, which leverages the properties provided by blockchain.

Drawbacks:

- *Cost.* On a public blockchain, extra code that implements the permission control mechanism also has additional deployment and run-time cost.
- *Lack of flexibility.* Such permissions are defined in the smart contract before its deployment, therefore they are difficult

[47] https://paritytech.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/

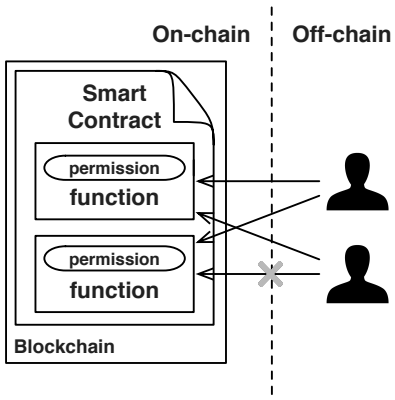

Figure 14: Embedded Permission Pattern

to change. However, permissions may be required to be dynamic. A mechanism is needed to support dynamic granting and removal of permissions.

Related patterns: *Multiple authorization* (Section **??**) and *Off-chain secret enabled dynamic authorization* (Section **??**) are different ways to design authorization. And *Authorization* from [**?**]

Known uses:

- The Mortal contract discussed in the Solidity tutorial[48] restricts the permission of invoking the *selfdestruct* function to the "owner" of the contract – where "owner" is a variable defined in the contract code itself.
- The *Restrict access* pattern suggested in the Solidity tutorial[49] uses *modifier* to restrict who can make modifications to the state of the contract or call the functions of the contract. *Modifier* is a mechanism to add a piece of code before the function to check certain conditions. *Modifier* can make such restrictions highly readable.

7.4 Pattern 14: Factory Contract

Summary: An on-chain template contract is used as a factory that generates contract instances from the template. Fig. **??** is a graphical representation of the pattern.

Context: Applications based on blockchain might need to use multiple instances of a standard contract with customization. Each contract instance is created by instantiating a contract template. For example, in a business process management system, each of the business process instances might be represented by a smart contract being generated from a contract template representing the business process model [**?**]. The template can be stored off-chain in a code repository, or on-chain, within its own smart contract.

Problem: Keeping the contract template off-chain cannot guarantee consistency between different smart contract instances created from the same template because the source code of the template can be independently modified.

[48] http://solidity.readthedocs.io/en/develop/contracts.html
[49] http://solidity.readthedocs.io/en/develop/common-patterns.html

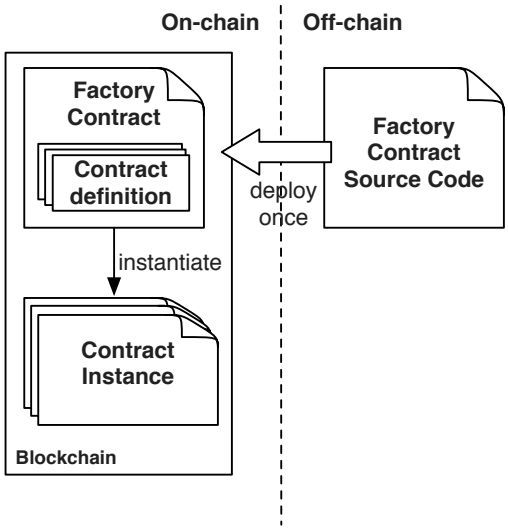

Figure 15: Factory Contract Pattern

Forces: The problem requires to balance the following forces:

- *Dependency management.* Storing the source code of smart contract off-chain in a code repository introduces the issue of integrating more systems into the blockchain-based application.
- *Secure code sharing.* Blockchain can be used as a secure platform to share code of smart contracts. As opposed to a traditional code repository, changes of code deployed on a smart contract can be strictly limited or prohibited.
- *Deployment.* If a public code repository, like Github, is used to store the source code of a smart contract, a component is needed to implement the function of deploying smart contract on blockchain, otherwise, the end users need to understand how to deploy smart contracts by sending transactions with the customized source code of the contract definition.

Solution: Smart contracts are created from a contract factory deployed on blockchain. The factory contract is deployed once from the off-chain source code. The factory may contain the definition of multiple smart contracts. Smart contract instances are generated by passing parameters to the contract factory to instantiate customized smart contract instances. Factory contract is analogous to a *Class* in an object-oriented programming language. Every transaction that generates a smart contract instance essentially instantiates an object of the factory contract class. This contract instance (the object) will maintain its own properties independently of the other instances but with a structure consistent with its original template.

Consequences:

Benefits:

- *Security.* Keeping the factory contract on-chain guarantees the consistency of the contract definition.
- *Efficiency.* If the contract definition is kept on-chain in a factory contract, smart contract instances are generated by calling a function defined in the factory contract.

Drawbacks:

- *Deployment cost.* If a public blockchain is used, using factory contract requires extra cost to deploy the factory contract.
- *Function call cost.* If a public blockchain is used, creating a new smart contract instance requires extra cost to call a function defined in the factory contract.

Related patterns: *Contract registry* (Section **??**). A contract registry can be used to store the addresses of all the smart contract instances generated from a factory contract. The factory and instance registry can be implemented in the same contract, although that limits upgradability.

Known uses:

- A tutorial from Ethereum developers[50] about how to create a contract factory from which smart contract instances can be created.
- Factory pattern has been applied in a real-world blockchain-based health care application [**?**].
- The business process management system in an academic work [**?**] uses a contract factory to generate process instances.

7.5 Pattern 15: Incentive Execution

Summary: Reward is provided to the caller of the contract function for invoking the execution. Fig. **??** is a graphical representation of the pattern.

Context: Smart contracts are event-driven programs, which cannot execute autonomously. All the functions defined in a smart contract need to be triggered either by a transaction from external account or another smart contract to execute. Other than the functions that provide regular services to users, some functions need to run asynchronously from regular user interaction, for example, to clean up the expired records, or make dividend payouts etc. Such functions usually involve a time, after which the function should start.

Problem: Users of a smart contract have no direct benefit from calling the accessorial functions. If a public blockchain is used, executing these functions causes extra monetary cost. Some accessorial functions are expensive to execute.

Forces: The problem requires to balance the following forces:

- *Completeness.* The regular services provided by a smart contract are supported by some accessorial functions.
- *Cost.* Execution of accessorial functions causes extra costs from the users.

Solution: Reward the caller of a function defined in a smart contract for invoking the execution, for example, sending back a percentage of payout to the caller to reimburse the (gas) execution cost.

Consequences:

Benefits:

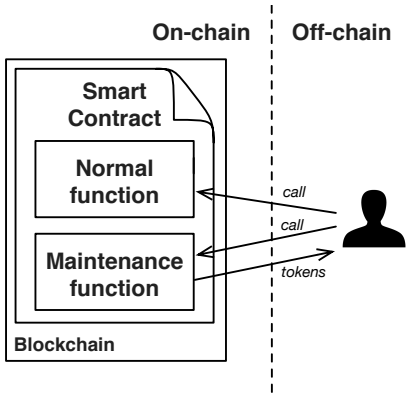

Figure 16: Incentive Execution Pattern

- *Completeness.* The execution of the accessorial functions helps to complete the regular services provided by the smart contract.
- *Cost.* The users, who spends extra to execute the accessorial functions, are compensated by the reward associated with the execution.

Drawbacks:

- *Unguaranteed execution.* Execution cannot be guaranteed even with incentive. Thus, another option is to embed the logic of accessorial functions into other regular functions that users have to call to use the services.

Related patterns: N/A

Known uses:

- *Regis*[51] is an in-browser tool for developers to create smart contracts representing registries on Ethereum. The functions that clean up the expired records provide incentive for users to execute them.
- *Ethereum alarm clock*[52] is a service provided by a smart contract running on Ethereum. It facilitates scheduling function calls for a specified block in the future and provides incentive for users to execute the scheduled function.

8 RELATED WORK

The following works collected a few design patterns of smart contracts or blockchain-based applications. [**?**] conducts an empirical analysis on smart contracts supported by different blockchain platforms. The paper focuses on the two most widespread platforms, Bitcoin and Ethereum. Nine common programming patterns are identified in Solidity-based smart contracts by manually inspecting the publicly available source code. The identified programming patterns include *Token*, *Authorization*, *Oracle*, *Randomness*, *Poll*, *Time constraint*, *Termination*, *Math* and *Fork check*. [**?**] applies four existing object-oriented software patterns to smart contract programming in the context of a blockchain-based health care application. The applied software patterns include *Abstract factory*,

[50] https://ethereumdev.io/manage-several-contracts-with-factories/

[51] https://regis.nu/

[52] http://www.ethereum-alarm-clock.com/

Flyweight, *Proxy*, and *Publisher-subscriber*. [**?**] proposes five patterns for blockchain-based applications focusing on what data and computation should be on-chain and what should be kept off-chain, which include *Challenge response pattern*, *Off-chain signatures pattern*, *Content-addressable storage pattern*, *Delegated computation pattern*, and *Low contract footprint pattern*.

Compared with the existing works, our paper covers system-level design patterns about interaction between blockchain and other components within a big software system, data management patterns, security patterns, and structural patterns for smart contracts. Some structural patterns are new and some are modifications of the existing design patterns. More importantly, we provide use cases from the real world with each of the patterns. There is some overlap between the existing works and our paper. For example the *Proxy pattern* from [**?**] is a more generic pattern compared with our *Off-chain data storage pattern*. The *Off-chain signatures pattern* from [**?**] is similar to our *State channel pattern*. The *Authorization pattern* from [**?**] is similar to our *Embedded permission pattern*.

9 CONCLUSIONS

We view the blockchain as a fundamental building block of large-scale decentralized software systems. For effective use of blockchain to this end, patterns are needed that show how to make good use of the blockchain in the design of systems and applications. In this paper, we propose a pattern collection for blockchain-based applications. Our pattern collection includes three patterns about interaction between blockchain and the external world, four data management patterns, three security patterns and six contract structural patterns. The pattern collection provides an architectural guidance for developers to build applications on blockchain. Some patterns are designed specifically for blockchain-based applications considering the unique properties of blockchain. Others are variants of existing software patterns applied to smart contracts. We plan to illustrate how to implement these patterns in the context of specific blockchain platforms and how to apply these patterns within a real world applications in our future work.

ACKNOWLEDGMENTS

We want to thank Tim Wellhausen, the shepherd for the design patterns paper, and the anonymous reviewers of our paper for their helpful comments.