Smart Contract Migration Across Heterogeneous Blockchains

Tiago Domingues
tiagofsdomingues@tecnico.ulisboa.pt

Instituto Superior Técnico

January 2022

Abstract. Since the emergence of Bitcoin over a decade ago, blockchain technology has been maturing at an unprecedented rate. The success of Ethereum allowed numerous possibilities and new blockchains with smart contract functionalities started to emerge. These new systems often present different trade-offs between decentralization, scalability and security. Having several blockchains in the ecosystem gives developers the liberty to choose which platform best suits their use-case and their application, depending on their requirements and the type of users they want to target.
Since there are plenty alternatives for users and developers to choose from, interoperability and migration have become outstanding subjects in blockchain technology. There is an increasing need to allow users and applications to easily move across chains and take advantage of what these platforms have to offer. Therefore, systems that enable interoperability between blockchains and allow smart contracts to be migrated are required to fulfill the needs of the industry and to keep it moving forward.
This document proposes a tool to migrate smart contracts between heterogeneous blockchains. More precisely, it proposes an extension to an existing tool called *Osprey*, that allows the translation of Solidity smart contracts to Hyperledger Fabric chaincode written in Typescript. The main goal is to add support to the translation of Typescript chaincode to Solidity, making *Osprey* a full two-way migrating tool.

Keywords: Blockchain Interoperability · Migration · Translation · Smart Contracts · Ethereum · Solidity · Hyperledger Fabric · Chaincode.

Table of Contents

1 Introduction

Since the emergence of Bitcoin [42] over a decade ago, blockchain technology has been maturing at an unprecedented rate. What was originally designed as a peer-to-peer electronic cash system that made radical developments in money and currency as we knew it, rapidly evolved into a more complex network of decentralized applications, with the launch of Ethereum [7].

Ethereum was the first blockchain to provide the ability to use a Turing-complete programming language to write what is called a smart contract. As per Buterin's description, these contracts "can be used to encode arbitrary state transition functions" [7]. Smart contracts drew attention from several fields (audits, education, finance, health care, insurance, etc.) and enabled a meaningful amount of systems and applications never imagined before (decentralized exchanges and liquidity providing, decentralized borrowing and lending, decentralized derivatives, DAOs, NFTs, etc.).

The success of Ethereum allowed numerous possibilities and, therefore, new blockchains with smart contract functionalities started to emerge. These new blockchains often present different trade-offs between decentralization, scalability and security, also know as The Blockchain Trilemma [12], termed by Vitalik Buterin. Having several blockchains gives developers the liberty to choose which platform best suits their use-case and their application, depending on their requirements and their target users.

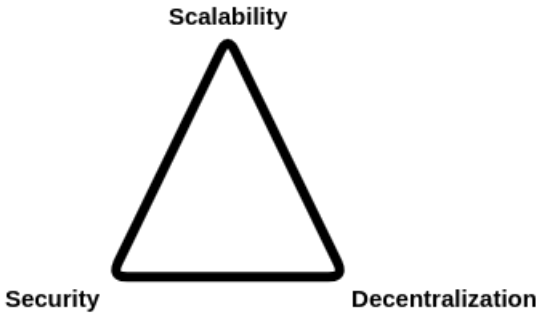


Figure 1. The Blockchain Trilemma [12]

Nowadays, Ethereum is by far the most widely used smart contract blockchain, but many others have been gaining a lot of traction and adoption by both developers and users. Some blockchains adopted the Ethereum Virtual Machine (EVM) [7] and opted to make more relevant changes on the consensus mechanism layer, such as Avalanche [47], Binance Smart Chain [6], Fantom [15], Harmony [54], among others. Alternatively, some blockchains have a whole new approach and decided not to reuse the EVM, such as Cardano [34], Cosmos [35], Polkadot [59] and Solana [60].

The first type of platforms is very close to Ethereum in the sense that any Solidity smart contract deployed on the Ethereum blockchain can be easily deployed on any of these blockchains with few to zero changes, since the EVM is being used. Therefore, the process of deploying a decentralized application (dApp) in several of these blockchains is not as hard as one would expect. On the other hand, the second type is not EVM-compatible and Solidity smart contracts cannot be natively deployed in these blockchains, given the fact that another kind of Virtual Machine is being used, which may require the use of other Turing-complete programming languages (for instance, Plutus and Rust).

More recently, there have been significant developments concerning the scalability of Ethereum. By scaling Ethereum, it is possible to increase transaction speed and transaction throughput, without sacrificing the other two vertices of the trilemma (decentralization and security) [10]. One way to achieve this is through what are known as layer 2 solutions [9]. Some of these solutions, such as Optimistic rollups, have already been deployed (e.g. Arbitrum [43] and Optimism [45]) and others, like Zero-knowledge rollups (or ZK rollups), are going to be deployed in the following months (e.g. zkSync 2.0 [63] and StarkNet [51]). This represents a major step towards scaling Ethereum and can increase Ethereum's transaction throughput from ∼15 tps up to ∼3000 tps, in the near future, and up to ∼100000 tps, after the Eth2 shard chains upgrade [8].

Therefore, there are plenty solutions and platforms with different trade-offs from which developers and users can choose from, in order to meet their requirements. As in any area, having the possibility to choose between different sets of features presents a major advantage to the industry. However, migrating between different blockchains (and layer 2 solutions) can often be a non-trivial challenge. As mentioned before, there are several blockchains with relevant levels of adoption and it is important to be able to connect these blockchains (and even the layer 2s) in order to allow users to move easily between them and developers to painlessly migrate their dApps from one to another.

Currently, there is a lot in this space that may indicate that the industry is moving towards a multi-chain future with cross-chain communication, transactions and applications. For instance, there has been an increasing adoption in blockchains that present themselves as an alternative to Ethereum and several applications have been launching on different chains, allowing users to experience different platforms and to move and bridge across them.

Thus, it is important to work towards the improvement of the user experience (UX) and ensure that developers can conveniently deploy their applications on multiple blockchains without having to manually code the same algorithms in several programming languages. Hence, a tool to convert Hyperledger Fabric [29] chaincode into Ethereum smart contracts is proposed.

1.1 Objectives

Given the importance of cross-chain communication, transactions and applications in the future of blockchain technology, the main goals of this project consists of:

- Studying blockchain interoperability techniques and smart contract migration between heterogeneous blockchains;
- Developing a tool for migrating smart contracts written in Typescript chaincode for Hyperledger Fabric to Solidity.

Regarding the latter, it will be implemented by extending an existing tool (Osprey [1]) that is currently only capable of migrating smart contracts in Solidity to Hyperledger Fabric chaincode. By doing so, this framework will be more complete and will allow a full two-way migration between the aforementioned smart contract languages and respective blockchains. The primary focus of this tool is to:

- Translate the target contract from Typescript chaincode to Solidity;
- Guarantee that the behavior it presents in the Ethereum blockchain is the same as the original contract had in the Hyperledger Fabric blockchain, for all supported functionalities.

The ultimate objective of this project is to help and encourage enterprises and developers that wish to migrate their applications from Hyperledger Fabric to Ethereum or to any other EVM-based blockchain (e.g. Avalanche, Binance Smart Chain, Fantom, Harmony) or EVM-compatible layer 2 solution (e.g. Arbitrum, Optimism, zkSync 2.0).

1.2 Document Structure

This document is organized into six more sections. First of all, Section 2 presents the necessary background to follow the main subject of this document. In particular, it does an overview of what blockchain technology is and the differences between permissionless and permissioned systems. Then, it introduces Ethereum – the biggest smart contract platform and most used permissionless blockchain –, as well as Hyperledger Fabric – a widely used permissioned blockchain framework by enterprises – and other Hyperledger projects, namely Besu and Cactus. Furthermore, this section covers the topics of *Cross-Blockchain Communication* and several approaches to *Blockchain Interoperability*, such as *Public Connectors*, *Blockchain of Blockchains* and *Hybrid Connectors*.

Then, Section 3 presents some related work developed in the field of smart contract migration between heterogeneous blockchains. Moreover, for each subsection, it is done an overview of the particular implementation that is being addressed and an elementary discussion about its advantages and drawbacks.

Next, Section 4 covers the proposed solution to the problem trying to be solved by summarizing the main aspects of the proposed tool as well as outlining its architecture.

Following, in Section 5, it is suggested a method to evaluate the proposed tool in order to be possible to draw some conclusions about it and assess the obtained results. In addition, it covers the likely procedures that are intended to be carried out during the evaluation phase.

Section 6 proposes the work schedule that is expected to be conducted by presenting a Gantt chart with the estimated duration of each task that composes the project.

Finally, Section 7 draws some conclusions about the current panorama of blockchain technology and the state-of-the-art in blockchain interoperability, mainly focusing on smart contract migration between heterogeneous blockchains. Furthermore, it briefly recaps the problem being addressed and the relevance of the proposed solution to the current landscape of the industry.

2 Background

This section gives an overview of what blockchain is and the two main types of this technology: Permissionless and Permissioned. In this section, some specific implementations and use cases like Ethereum and Hyperledger Fabric are introduced. After that, the concept of smart contracts is presented, explaining what they are and some of the most used languages to write them (Solidity, Vyper, etc.). To further understand the concept of migration between blockchains, there is a discussion about the topics of Cross-Blockchain Communication and Blockchain Interoperability, which provides an understanding of how different blockchains can communicate with each other, how they can be connected and how information can be migrated between them.

2.1 Blockchain Overview

A blockchain is a distributed system composed of several machines that maintain a shared state. This shared state is often denominated *distributed ledger* and it facilitates the process of storing transactions and tracking digital assets in the network. Blockchains are also known to be decentralized and immutable [42], but those properties may vary between different implementations. The machines that participate in the network, called nodes, ensure the system behaves as expected through the use of computational and storage resources. Usually, all nodes are treated equally and no single node is trusted. Instead, the network is trusted as a group, given the assumption that the majority of nodes are honest (i.e. do not relay false or malformed information) [42]. This makes blockchain tolerant to *crash faults* and *Byzantine faults* [5]. The bigger the number of nodes, the more resistant the blockchain will be. The diversity of these nodes and the way they are widespread can also have an impact on the resistance and availability of the network.

What particularly differentiates a blockchain from other distributed ledger systems is its peculiar type of data structure [4]. The ledger is composed of blocks of transactions that together represent the entire history of the blockchain. These blocks have certain storage capacities and, once filled with several transactions, are linked to the previous block. The final result is a ledger of multiple blocks chained chronologically, where each block contains the cryptographic hash of the previous block in the header field [5].

Blockchains can also have a currency attached to the system [42]. Usually, permissionless blockchains have financial incentives and reward the nodes that secure the network through the means of a cryptocurrency (also denominated cryptoasset). Permissioned blockchains, on the other hand, do not necessarily need to have a currency nor incentives [5]. The major differences between these two types of blockchains are outlined on Table 1 and will be addressed in detail in Sections 2.2 and 2.3.

	Permissionless Blockchains	Permissioned Blockchains
Access to the network	Open to anyone	Restricted
Distributed	✓	✓
Decentralized	✓	✗
Censorship resistant	✓	✗
Trustless	✓	✗
Cryptocurrency-agnostic	✗	✓
No transaction fees	✗	✓
Examples	Bitcoin, Ethereum	Hyperledger Fabric, Corda

Table 1. Comparison between permissionless and permissioned blockchains

In 2014, Ethereum's white paper [7] redefined the concept of smart contracts, initially proposed by Nick Szabo in 1997 [53]. A smart contract is a program, consisting of a collection of code and data, that is deployed in a blockchain [7]. These programs can then be run to execute what they were programmed for. For instance, users can interact with them through transactions that execute a particular function defined on a smart contract. Smart contracts are used to implement fungible (ERC-20 [56]) and non-fungible (ERC-721 [20]) tokens, as well as many decentralized applications, mainly in the realm of Decentralized Finance (also known as DeFi). Currently, the most used programming language to write smart contracts is Solidity, although Vyper is also gaining some popularity [24]. Both these languages can be used to write smart contracts for any EVM-based blockchain. Other blockchains usually use other languages: for example, TypeScript can be used to write smart contracts for Hyperledger Fabric, while Rust can be used in permissionless platforms, such as Polkadot [59] and Solana [60].

2.2 Permissionless Blockchains and Cryptocurrencies

Permissionless blockchains are open blockchain systems that do not require any type of authentication for users to join the network or to access the ledger [5]. This type of system uses incentives to maintain trust and secure the network and, therefore, has its own currency. The most popular permissionless blockchains are Bitcoin and Ethereum, which use bitcoin [42] and ether [7] as currency, respectively. These incentives motivate nodes to produce new correct blocks and, thus, avoids malicious behavior. The production of new blocks, in Bitcoin and

Ethereum, follows a consensus mechanism called Proof-of-Work [42]. In this algorithm, participants compete to solve a cryptographic puzzle in order to produce the next block and, consequently, to receive the block reward. This mechanism not only allows to have an open and decentralized peer-to-peer network of nodes that collectively agree on the state of the ledger, but also prevents sybil attacks. Another popular consensus algorithm in permissionless blockchains is Proof-of-Stake [7], which has been increasingly adopted in more recent implementations of blockchain technology, including Ethereum's Eth2 upgrade [21].

These blockchain currencies, such as bitcoin and ether, are usually called cryptocurrencies (or cryptoassets) because they rely on cryptography. It is relevant to state that fungible tokens (i.e. ERC-20 tokens) may also fall into this category. Cryptocurrencies are very distinguishable from fiat currencies (such as the US dollar and the euro). In the first place, fiat currencies are backed by a central government or central bank [58], while cryptocurrencies are backed by the decentralized network [57]. Secondly, governments and central banks control the supply of fiat currency and are constantly altering its monetary policy without citizens agreeing upon those changes [58]. On the other hand, cryptocurrencies' supply and monetary policy are algorithmically determined and can only be changed if the majority of the participants in the network agrees to make those changes [57]. Ultimately, both fiat currency and cryptocurrency have no intrinsic value. The former only has value because governments and central banks ensure their value, while the value of the latter is determined by the users themselves.

There are several properties of permissionless blockchains that differentiate them from permissioned blockchains and other distributed ledger technology systems. The most distinguishing properties of permissionless blockchains are [37]:

- Decentralization;
- Openness;
- Borderless;
- Censorship-resistance;
- Neutral;
- Pseudonymity/Privacy;
- Immutability;
- Reliability.

Nowadays, most blockchains support smart contracts. Regarding permissionless blockchains, these are often separated into two categories: EVM-based (or EVM-compatible) blockchains and non-EVM blockchains. The first type uses the Ethereum Virtual Machine and shares compatibility in the runtime. The second type uses a different execution environment and does not have the same runtime as Ethereum. When two (or more) blockchains share compatibility in the runtime and, thus, allow smart contracts to run on the same execution environment, they are homogeneous. Contrarily, two blockchains are heterogeneous if they do not share any compatibility, meaning smart contracts from one platform cannot be directly deployed onto the other [1]. For instance, Ethereum, Avalanche and

Fantom are homogeneous blockchains, while Ethereum and Solana are heterogeneous.

2.2.1 Ethereum. Ethereum is the second most popular blockchain, after Bitcoin. However, it is notably the most used platform and, by far, the one that generates the most fee revenue [39]. Currently, Ethereum settles more value than Bitcoin [38] and has a higher number of transactions on a daily basis [25,26]. This is mainly due to the fact that Ethereum supports smart contracts (while Bitcoin does not), allowing the deployment of ERC-20 tokens and dApps, which generate a lot more traffic in the network compared to Bitcoin, where transactions are mainly just simple transfers of value from one account to another.

Ethereum's smart contracts are usually written in Solidity (although Vyper has been gaining some traction), a Turing-complete programming language heavily influenced by JavaScript, Python and C++, that was specifically invented to write smart contracts for the EVM [50].

Listing 1.1 [49] shows a simple smart contract developed with Solidity. The Solidity contract is a simplified version of a token that can be deployed on an EVM-compatible blockchain. When it is launched, the constructor function (*Token()*) is run, where the creator has to choose the total supply of the token and all coins are credited to the creator's address. Then, by using the *transfer()* function, users can transfer tokens to other users. Moreover, the *balanceOf()* function allows to check the balance of the token in a particular address.

```
pragma solidity ^0.4.18;

contract Token {

  mapping(address => uint) balances;
  uint public totalSupply;

  function Token(uint _initialSupply) {
     balances[msg.sender] = totalSupply = _initialSupply;
  }

  function transfer(address _to, uint _value) public returns (bool) {
     // <yes> <report> ARITHMETIC
     require(balances[msg.sender] - _value >= 0);
     // <yes> <report> ARITHMETIC
     balances[msg.sender] -= _value;
     balances[_to] += _value;
     return true;
  }

  function balanceOf(address _owner) public constant returns (uint balance) {
     return balances[_owner];
  }
}
```

Listing 1.1. Solidity smart contract example [49]

Furthermore, there is a public state variable of type *uint*, named *totalSupply*, and users' balances are kept by a state variable of type *mapping*, called *balances*,

that stores the data in the form of key-value pairs, where keys are users' addresses and values correspond to the number of tokens in each address.

In Solidity, only public functions can be called by users who want to interact with the smart contract and, therefore, *transfer()* and *balanceOf()* functions need to be public. The same applies the *totalSupply* state variable. The *mapping* variable does not need to be public because *balanceOf()* queries it to get the balance of the specified address. Besides *public*, there are other types of visibility in Solidity. For further clarification, functions can be declared as [19]:

- *public*: any contract and account can call the function;
- *private*: the function can only be called inside the contract where it is defined;
- *internal*: same as private, except it is also accessible to contracts that inherit from the contract where the function is defined;
- *external*: similar to public, except it cannot be called by other functions inside the contract where it is defined.

State variables can be declared as *public*, *private* or *internal*, but not as *external* [19].

Apart from visibility, functions can also be marked with the following keywords: *view* and *pure*. A *view* function does not modify the state of the blockchain and a *pure* function does not change or read the state of the blockchain [19]. In both cases, functions are free to be executed because no transactions are created and, therefore, there is no need to pay gas.

Additionally, variables can be declared as either *storage*, *memory* or *calldata*. These keywords specify the location of where data is stored [19]:

- *storage*: state variable (i.e., it is stored on the blockchain) that persists between function calls and transactions;
- *memory*: variable is in memory and it exists while a function is being called;
- *calldata*: special data location only available for external functions.

Furthermore, in Solidity, *function modifiers* are used to modify the behavior of a particular function, for instance, adding a prerequisite to a function. *Modifiers* are code that can be run before and/or after a function call, which allows to make verifications or assertions prior and following the execution of the function. Mainly, *modifiers* can be used to validate the function inputs and to restrict access to the function [19].

Since Solidity is a Turing-complete language that supports most functionalities of a common programming language (unlike Bitcoin's scripting language), it can bring some vulnerabilities to a decentralized system such as Ethereum, mainly concerning *Denial of Service* (DoS) [7]. For instance, infinite loops and other computational wastage in code can potentially lead to a *Denial of Service*.

To prevent such attacks, Ethereum introduced the concept of gas [19]. Gas is the unit of computation in the system and every user must pay gas fees according to every resource that they consume (including computation and storage) [7]. These fees are paid in ether to the network. Currently, fees are divided into a *basefee* and a *tip*. While the *basefee* is burned (i.e. it is destroyed by the protocol),

the *tip* is paid to *miners*. Burning the *basefee* plays a crucial role in the protocol, since it reduces the risk of miners manipulating the fee in order to extract more value from users [11]. This practice is called miner extractable value (MEV).

At the moment, gas fees in Ethereum are really high, which represents a severe drawback and prevents mainstream adoption of the protocol. To fix this, Ethereum 2.0 is currently being worked on. To begin, Eth2 will first change the consensus algorithm from Proof-of-Work (PoW) to Proof-of-Stake (PoS) and, later, implement up to 64 shard chains, which will expand Ethereum's capacity to process transactions and store data [21]. At the same time, layer 2 solutions such as Optimistic rollups and ZK rollups can also help Ethereum's scalability, by moving computation from layer 1 to layer 2. As mentioned before, the combination of Eth2 with rollups can enhance Ethereum up to ∼100000 transactions per second, substantially reducing gas fees [8].

2.3 Permissioned Blockchains

Permissioned blockchains are blockchains with an access control layer, in which users need to authenticate themselves in order to have access to the ledger [5]. These blockchains (often also called private or consortium) are mostly used by organizations or businesses that wish to process private transactions within a permissioned group of known participants, usually from different organizations and sometimes even geographic locations.

Unlike permissionless systems, these do not need to be censorship-resistant, transparent and certainly not anonymous. Concerning centralization, permissioned blockchains can be fully centralized (yet distributed) or partially decentralized. Governance is determined by the participants of the network in a very different way from permissionless blockchains, where consensus algorithms are more flexible and adjustable, and do not depend on cryptocurrencies nor monetary incentives [5]. Therefore, permissioned blockchains are highly customizable and allow better scalability.

Hyperledger [30] is an umbrella project of the Linux Foundation with the goal of advancing the adoption and development of blockchain technology, mainly focusing on enterprise blockchain ecosystems. Hyperledger is known for open-sourcing its projects, like Fabric [29], Besu [27], Cactus [28], among others. Corda [17] and Quorum [16] are other examples of permissioned blockchain systems.

2.3.1 Hyperledger Fabric. Hyperledger Fabric is a widely used private blockchain platform known for its modular and versatile design. This permissioned distributed ledger framework enables running distributed applications written in Java, Go, JavaScript and Typescript [29]. Like typical private blockchains and unlike public blockchains, Fabric is cryptocurrency-agnostic in order to better suite the needs of enterprise-grade distributed ledgers.

In Fabric, the participating organizations create a consortium, i.e., a group of non-trusting parties that work to accomplish a consensus. Each organization has its own set peer nodes that trust each other [29]. Instead of having a order-execute model identical to public blockchains, Fabric has its own architecture,

called order-execute-validate, that is deterministic and allows parallel execution. Fabric adopted a modular and pluggable consensus algorithm that makes use of a permissioned voting-based scheme that can be broken into four phases: *Proposal*, *Endorsement*, *Ordering* and *Execution* [32].

First, in the *proposal phase*, a client (representing an organization) creates a transaction proposal and forwards it to a subset of peers [32]. The *endorsement phase* is driven by an *endorsement policy* that defines which peer nodes need to attest the correct execution of a particular smart contract. Therefore, each transaction is only endorsed (executed) by the subset of peer nodes specified by the endorsement policy [29]. After that, the *ordering phase* groups the endorsed transactions into ordered blocks to be committed to the ledger, by using the Raft protocol, which offers crash-fault tolerance [29]. Then, blocks are broadcasted and disseminated to all peer nodes through the use of a *gossip* protocol, for instance [29]. In the *validation phase*, all peer nodes take a block of ordered transactions and validate the correctness of the result [1].

Each peer node must have at least one of the following roles in the network, as defined by Fabric [32]:

- *Clients* submit transactions for execution and broadcast endorsed transactions for the ordering phase.
- *Committing peers* maintain the current snapshot of the distributed ledger, in the form of a key-value store. These peers do not execute any transactions.
- *Endorser peers* are a group of peers specified by the endorsement policy of the chaincode. These peers simulate the transaction execution on an isolated setting and attest for its correctness.
- *Orderer peers* receive endorsed transactions and gather them to build ordered blocks. Then, blocks are propagated across the network to be validated and thereafter committed to the blockchain. These peers keep track of both valid and invalid transactions.

Moreover, Fabric establishes *anchor peers* and *leader peers* [32]. The former works as a middleman between peers from its organization and peers from an outside one. The latter is accountable for disseminating transactions from orderer to committing peers.

In Hyperledger Fabric, channels allow for private communication tunnels between a subset of network participants, enabling confidential and private transactions [32]. This means that only authenticated and authorized members can transact on that channel and visualize the subset of transactions that go through it.

Similarly to Ethereum, Hyperledger Fabric also has the concept of smart contracts. Furthermore, it also defines a new concept called *chaincode* [29]. Chaincode and smart contracts implement the logic behind applications and use cases, and generate new facts to be added to the shared ledger. Developers can write chaincode in four different programming languages: Java, Go, Javascript and Typescript [1]. As an illustration, Fabric allows to create an underlying token with chaincode [32], since it does not support built-in cryptocurrencies. Thus, allowing its members to transfer and exchange assets throughout the network.

In essence, smart contracts set the rules between participating organizations and applications make calls to a smart contract to create transactions that are registered on the shared ledger. One or more smart contracts are packaged into a chaincode, which is then deployed on the blockchain in order to be made available to such applications [29]. Apart from being used by administrators to group related smart contracts for deployment, chaincode can also be employed for low level system programming of Hyperledger Fabric [29].

2.3.2 Hyperledger Besu. Hyperledger Besu is an open source Ethereum client that can run on the Ethereum Mainnet, on testnets (i.e., Görli, Rinkeby and Ropsten) and on private networks [1].

Besu supports Ethereum's Proof-of-Work consensus mechanism (*Ethash*), as well as Proof-of-Authority consensus algorithms such as IBFT, QBFT and Clique [27]. While Mainnet and Ropsten require Proof-of-Work, Görli and Rinkeby are proof-of-authority testnets. Regarding private networks, Besu allows to create them with any of these consensus protocols.

Furthermore, Besu supports standard and general Ethereum functionalities, namely [27]:

- Smart contract development;
- Decentralized application (dApp) development;
- Ether mining.

Therefore, Besu allows organizations to both interact with Ethereum public networks and develop enterprise-level applications in private networks. Doing so, Besu enables secure and high-performance transaction processing and supports features like privacy and permissioning [27].

2.3.3 Hyperledger Cactus. Hyperledger Cactus is a blockchain integration tool in the form of a Blockchain Connector [1], more precisely, a Trusted Relay, as discussed in Section 2.5.3. This type of connector ensures interoperability between cross-chain transactions, detailed in Section 2.5.

Cactus was designed to allow users to securely integrate different blockchains. This tool is composed of nodes, where each has a connector, a validator and a group of plugins, which allow the system to have high modularity and flexibility. While the connector is responsible for establishing communications with the source and target blockchains, the validator checks the validity of transactions [1].

Regarding its features, Cactus is compatible with most Hyperledger technologies (in particular, Fabric and Besu) and other blockchain protocols such as Corda, Quorum and Ethereum [28]. Therefore, it enables several use cases: one-way and two-way ledger transfers, one-way and two-way ledger interactions, ledger entry point coordination, and atomic swaps [28].

2.4 Cross-Blockchain Communication

Cross-blockchain communication (or simply cross-chain communication) allows distinct blockchains, usually referred to as source and target, to verify data and transactions without the intermediation of a centralized third-party [5]. This communication between different blockchains is essential to guarantee the decentralized functionality of the technology.

Since blockchain systems have different structures, exchanging data between chains without additional software or third-party intervention can be a challenging task. Therefore, cross-chain communication methods are an essential part of blockchain interoperability. For that reason, two communication concepts arose [5]: *cross-chain communication protocol* (CCCP) and *cross-blockchain communication protocol* (CBCP).

On the one hand, a *cross-chain communication protocol* (CCCP) manages the communication between a pair of homogeneous blockchains in order to guarantee that both systems can properly synchronize cross-chain transactions [5]. For example, this method can be applied across Ethereum and other EVM-based blockchains, such as Avalanche and Fantom. On the other hand, a *cross-blockchain communication protocol* (CBCP) manages the communication between a pair of heterogeneous blockchains in order to guarantee that both systems can properly synchronize cross-blockchain transactions [5]. For instance, this method can be applied across Ethereum (or other EVM-based blockchain) and a non-EVM blockchain, public (e.g., Solana) or private (e.g., Hyperledger Fabric).

These two pillars of cross-chain communication represent a requirement for blockchain interoperability considering that, without communication, two different blockchain systems cannot interact and, consequently, there would not exist any actual interoperability. Since CCCPs work between similar platforms, these protocols can be more straightforward to implement than CBCPs, given that the underlying technology is predominantly the same (i.e., the Ethereum Virtual Machine). Therefore, there is high compatibility between these blockchains and their smart contract functionality (e.g., developers do not have to translate or rewrite smart contracts with the purpose of ensuring the same behavior in both chains).

Some studies, mainly by Zamyatin et al. [62], and Lafourcade and Lombard-Platet [36], state that cross-chain transactions are only possible with the intervention of a third-party. However, this third-party does not necessarily need to be a centralized one. In reality, cross-chain transactions can be achieved through permissionless, trustless and decentralized systems:

1. Cross-chain liquidity protocols (also know as cross-chain Automated Market Makers [48]), such as THORChain [55];
2. Cross-chain bridges, such as Synapse Protocol [52] or Multichain (previously known as AnySwap) [41].

While the latter usually only works between homogeneous blockchains, the former already works between heterogeneous blockchains allowing cross-chain swaps with the native assets instead of wrapped or synthetic tokens.

2.5 Blockchain Interoperability

One reason why the adoption and use of blockchain technology is limited today is related to the fact that the industry is fragmented. As presented throughout this paper, there are several blockchain platforms to build and develop on. However, these distributed systems are separated and there is yet to exist a predominant technology to enable their interaction.

Therefore, blockchain interoperability is the ability of independent blockchain infrastructures to communicate with each other, not necessarily with the intent of making direct state changes to the other blockchain, but in order to trigger a predetermined set of operations and services on the other system [1]. In other words, blockchain interoperability should enable homogeneous and heterogeneous blockchain systems to exchange and use data, and to move value in the form of digital assets while preserving consistency and validity throughout this process [5].

A *cross-chain transaction* (CC-Tx) [5] is a transaction issued by a *cross-chain communication protocol* (CCCP) to enable transactions between homogeneous blockchains, namely EVM-based chains. In contrast, a *cross-blockchain transaction* (CB-Tx) [5] is a transaction issued by a *cross-blockchain communication protocol* (CBCP) to enable transactions between heterogeneous blockchains, for instance, Ethereum and Hyperledger Fabric.

The concept of blockchain interoperability can be separated into three classifications [5]: *Public Connectors*, *Blockchain of Blockchains* and *Hybrid Connectors*. For the purpose of this study, the main focus will be on *Hybrid Connectors*, particularly in *Blockchain Migrators*.

2.5.1 Public Connectors. Public connectors enable interoperability between different public blockchains that use cryptocurrencies. This category can be subdivided into four solutions [5]: sidechains, notary schemes, hashed time lock contracts and hybrid solutions.

Sidechains are separate blockchains which run in parallel to the mainchain and operate independently, having their own consensus algorithm and block parameters to efficiently process transactions [22]. A sidechain can be considered as an extension to the main blockchain [1] that usually works in the same way (e.g., an Ethereum sidechain would be based on the EVM) and communicates with the mainchain via a cross-chain communication protocol [5]. Therefore, the separate system is attached to the mainchain and helps offloading transactions. However, these systems are less decentralized and less secure, and cannot be considered as a layer 2 solution given that the network is not secured by the layer 1 [22].

Public sidechains are useful in multiple contexts including micro-transactions, stable transactions, and application-specific transactions (Application Specific Sidechain, or DAppChain). For instance, sidechains can be used for transferring digital assets. A *two-way peg* is a mechanism that allows this transfer between the mainchain and the sidechain by locking the number of assets that were

transferred in the source chain and registering it in the target chain. Three possible implementations of this mechanism are [5]:

- *Simplified payment verification* (SPV);
- *Centralized two-way peg*;
- *Federated two-way peg*;

State channels [23] are a good example of sidechains that are used to implement payment channels in a very similar way to the concept of Bitcoin's Lightning Network, except the fact that they also support state updates. Upon closing the channel, only the final state is broadcasted to the mainchain.

Another great example of a sidechain in the Ethereum ecosystem is Polygon PoS chain (previously known as Matic Network) [33]. This EVM-based chain works like any independent Proof-of-Stake blockchain with its own infrastructure, cryptocurrency (MATIC) and nodes. However, unlike traditional alternative blockchains, blocks of transactions are ultimately batched and settled on the Ethereum mainchain [33], allowing for greater performance and user experience. In order to implement this, there is a set of smart contracts on the Ethereum network that handle the communication between Ethereum and Polygon PoS chain, as well as other aspects, such as staking and transaction finality.

Notary Schemes involve third-party entities that monitor transactions in several blockchains at the same time [1]. Then, transactions can be triggered on one chain upon an event happening in another chain. Centralized exchanges (CEXs) such as Binance, Coinbase and FTX can be considered centralized notary schemes. Contrarily, decentralized exchanges like Uniswap and Sushiswap can be considered decentralized notary schemes [5].

Hashed Time Lock Contracts (HTLCs) enable a decentralized and trustless way of exchanging cryptocurrencies through cross-chain atomic operations, usually referred to as *atomic swaps*, eliminating the need for third-parties with the custody of funds like centralized exchanges [5]. This technique uses hashlocks and timelocks to guarantee the atomicity of all operations. Hashlocks work like a form of two-factor authentication, requiring the recipient to provide the correct secret phrase in order to claim the funds. Similarly, timelocks function like a timeout, requiring the recipient to claim the funds prior to expiry. Upon expiry, the sender is able to reclaim the original funds and the recipient is no longer able to claim them [40].

2.5.2 Blockchain of Blockchains. Blockchain of blockchains (sometimes also referred to as the *Internet of blockchains*) are frameworks that allow the reutilization of the system's layers (i.e., network, consensus, incentive, data and contract) in order to create highly customized, application-specific blockchains that directly inherit the ability of interoperating between each other [5], despite being heterogeneous.

Two of the most relevant examples of this multi-chain technology are Cosmos [35] and Polkadot [59]. Both platforms allow the creation of independent, parallel blockchains (called zones [35], in Cosmos, and parachains [59], in Polkadot) that can communicate and transfer value freely.

Although this technology removes the problem of blockchain interoperability within the own ecosystem, it does not guarantee interoperability between different platforms. For instance, Cosmos and Polkadot are not natively interoperable. However, the same mechanisms and techniques that can be applied to traditional blockchains (e.g., public connectors) can, in theory, be also applied to blockchain of blockchains systems to achieve interoperability [5].

Regarding smart contract migration, this technology abstracts this process as developers can write their code in a specific programming language and, upon deployment, the platform will compile it to the target parallel chain [59]. In the case of Cosmos and Polkadot, smart contracts are compilable to WASM (Web Assembly) and, consequently, smart contracts can be writen in several languages, such as C++, Rust, Go and Javascript.

Since blockchain of blockchains systems require the use of cryptocurrencies and the payment of transaction fees to operate the network, private blockchains like Hyperledger Fabric do not necessarily benefit as much from this technology as public blockchains [5].

2.5.3 Hybrid Connectors. Hybrid Connectors consist of interoperability solutions that are neither public connectors nor blockchain of blockchains. Unlike those, hybrid connectors are oriented to both public and private blockchain systems. This category can be subdivided into three solutions [5]: *Trusted Relays*, *Blockchain-Agnostic Protocols* and *Blockchain Migrators*.

Trusted Relays are trusted intermediaries that route transactions from a source blockchain to a target blockchain [5], working like a proxy. As it has been mentioned in Section 2.3.3, Hyperledger Cactus falls into this sub-category. The same goes for bridging protocols such as the ones mentioned in Section 2.4 (e.g., Synapse Protocol and Multichain). Another solid example of such technology are decentralized oracle systems like Chainlink [18] and Band Protocol [3]. Chainlink, in particular, is a framework that has been used to solve the *oracle problem* in blockchain technology, allowing for any off-chain data resource and computation to be connected to smart contracts. Moreover, Chainlink's *Cross-Chain Interoperability Protocol* (CCIP) [13] is a new cross-chain and cross-blockchain messaging standard that provides infrastructure for transferring data and smart contract commands across private and public blockchain networks.

Blockchain-Agnostic Protocols are cross-chain and cross-blockchain protocols that enable an interoperable multi-blockchain ecosystem with communication between homogeneous and heterogeneous blockchains [5]. Ripple's [14] *Interledger Protocol* (ILP) [46] can be considered as an example of a blockchain-agnostic protocol that can be used for payments and cross-border transfers across different payment networks.

Blockchain Migrators enable the migration of the state of a blockchain system to another [1]. Nowadays, migrating data across blockchains is a fairly achievable process [5]. However, migrating smart contracts is not as trivial, although there is some work being done in this field. For instance, there are several patterns [2] that can be explored in order to build a smart contract migration

tool. These patterns are called *Virtual Machine Emulation* and *Smart Contract Translation* and will be properly addressed in Section 3.1.

3 Related Work

This section covers the current state-of-the-art regarding interoperability and migration of smart contracts between heterogeneous blockchains. Here, existing solutions to this problem are reviewed and their advantages and disadvantages are accordingly discussed. Each presented solution is appropriately compared with the proposed tool of this study. To get started, Section 3.1 reviews the migration patterns of smart contracts and the limitations and strengths of using two of them. Then, Sol2js, a translation tool that converts Solidity smart contracts to Hyperledger Fabric chaincode written in Javascript, is introduced in Section 3.2. Last but not least, Section 3.3 analyses Osprey, a tool that allows the translation of smart contracts written in Solidity to Hyperledger Fabric chaincode written in Typescript. In this case, the importance of this tool to this study is also addressed, as well as the similarities between it and the proposed tool.

3.1 Smart Contract Migration Patterns

Patterns for Blockchain Data Migration is a paper published by HMN Dilum Bandara, Xiwei Xu and Ingo Weber [2] on migration of data and smart contracts across homogeneous and heterogeneous blockchains. In this case, a pattern represents a model for a particular context in a specific scenario designed to facilitate some form of migration from a source blockchain to a different blockchain system. This paper mentions several migration patterns [2]: *State Extraction Patterns*, *State Transformation Patterns*, *State and Transaction Load Patterns* and *Smart Contract Patterns*. Naturally, for the purpose of this study, the interest lays on patterns for smart contract migration. Therefore, the focus will be on *Smart Contract Migration Patterns*.

3.1.1 Overview. This paper introduces two different kinds of patterns regarding smart contract migration [2]: *Virtual Machine Emulation* and *Smart Contract Translation*.

The *Virtual Machine Emulation Pattern* should allow a smart contract or a batch of smart contracts written in one language (e.g., Solidity) to run on another blockchain system. In addition, it could also enable the use of the embedded states of the contracts on the target platform, when required. If the target blockchain shares the execution environment with the source blockchain, there is no need to perform any nontrivial operations. For instance, smart contracts can be easily moved from Ethereum to Binance Smart Chain because both are EVM-based blockchains. Otherwise, it is required to make a copy of the VM from the source blockchain and install it on the target system. After that, the original smart contracts must be marked as unusable on the source blockchain,

which can be done through the *Token Burning Pattern* [2]. Afterwards, the list of smart contracts must be deployed to the target blockchain as well as their state, which can be done through the *State Initialization Pattern* [2]. Since addresses most likely very across blockchain platforms, one should then update the smart contract address on the ID database and add a PoE (Proof-of-Existance) entry of all updated identifiers.

The *Smart Contract Translation Pattern* translates a smart contract written in a particular programming language to another language with the objective of running it on a distinct blockchain platform. Ideally, this process should be done automatically (or semi-automatically) through a compiler or transpiler. However, it is necessary to verify that the translated smart contract did not lose its functional correctness nor its security properties during the process. After guaranteeing that the translated code maintained the original behavior, the smart contract can be deployed on the target blockchain along with its embedded state. The final steps coincide with the ones from the previous pattern, which means that the ID database needs to be updated and a PoE of the original and translated code needs to be added [2].

3.1.2 Discussion. The *Patterns for Blockchain Data Migration* paper is remarkably relevant to the field of blockchain interoperability, as it covers a wide specter of key aspects. In particular, it formalizes two patterns that allow smart contract migration across both homogeneous and heterogeneous blockchain platforms, which deeply coincides with the work objectives of this document.

The former pattern is a simple and straightforward approach, but rather limited. In fact, the pattern works really well for homogeneous blockchains such as EVM-based platforms. However, when migrating between heterogeneous blockchains, it can get complicated. A first scenario addresses the possibility of the target blockchain being compatible with the installation of the execution environment (e.g., the EVM). In that case, the process can be more trivially accomplished. On the other hand, a second scenario, where the target blockchain cannot natively support the installation of the execution environment, is much more complex. In that event, developers must make protocol changes to accommodate this special feature. Therefore, it is a more exhaustive and sophisticated task.

The latter pattern is closely related to the solution proposed in this document. Unlike the previous pattern, this one is not byte code dependent and, thus, it does not require compatibility at the execution layer (i.e., the virtual machine). Nonetheless, it has one handicap: translating smart contract code from one programming language to another can be a complicated exercise, given that transpilers are complex tools and there is not much software available to perform such tasks. That is the main reason behind the primary purpose of this study.

3.2 Sol2js

Sol2js [61] is an open source translator tool [31], developed to convert Solidity smart contracts into Javascript, adding the necessary code to enable its

deployment on Hyperledger Fabric [61]. The authors of the tool claim that it can successfully translate up to approximately 70-75% of Solidity keywords and constructs (types, functions, inheritance and events) [31].

3.2.1 Overview. Sol2js is a tool that allows the conversion of Solidity smart contracts into Javascript chaincode through source-to-source translation. Therefore, it supports smart contract migration from Ethereum to Hyperledger Fabric.

This tool goes through the inputted smart contract and maps Solidity constructs to Hyperledger Fabric chaincode written in Javascript. This translation process is done in two major steps [61]:

1. Parsing the original source smart contract to create an Abstract Syntax Tree (AST).
2. Applying methods on the generated AST to produce the target Javascript chaincode.

Sol2js is able to parse and convert to Javascript up to approximately 70-75% of Solidity constructs and keywords, including some well-known Ethereum smart contracts, such as the ERC-20 token standard, and other smart contracts from the OpenZeppelin library [44].

3.2.2 Discussion. This translator tool is similar to the proposed solution as it allows migrating smart contracts across heterogeneous blockchains. However, there are two main differences to point out:

- While Sol2js enables the migration of smart contracts from Ethereum to Hyperledger Fabric, the proposed solution supports migration in the reverse direction (i.e., from Hyperledger Fabric to Ethereum).
- Sol2js uses Javascript as the programming language to code Fabric chaincode, whereas the proposed tool uses Typescript.

Apart from that, Sol2js does not support all Solidity features. Some of these not supported features (e.g., multiple inheritance, function overloading, function types, fixed-point number types, library and type overriding, etc.) are used in more complex smart contracts and applications. Thus, this tool would not fit them.

Furthermore, Sol2js is only capable of migrating smart contracts between these two blockchains in one direction (i.e., from Ethereum to Hyperledger Fabric). Therefore, it is not suited for developers and enterprises looking to migrate their projects from Hyperledger Fabric to Ethereum.

3.3 Osprey

Osprey [1] is a tool that allows the translation of smart contracts written in Solidity to Hyperledger Fabric chaincode written in Typescript. Moreover, by integrating Hyperledger Cactus and Hyperledger Besu, Osprey can compare the execution of the smart contracts in both blockchain environments (i.e., Ethereum and Hyperledger Fabric) and guarantee that the translated chaincode presents the same behavior as the original smart contract.

3.3.1 Overview. Osprey can be separated into two major modules: the *smart contract module* and the *test module*. The former processes input Solidity smart contracts by converting them into an Abstract Syntax Tree (AST) that is then iterated to construct the translated Typescript chaincode. The latter translates Javascript unit tests that are used to test the original Solidity smart contracts into Typescript test files that are used to test the translated chaincode in Hyperledger Fabric, also by first assembling an Abstract Syntax Tree of the unit test code [1].

The integration with Hyperledger Cactus and Hyperledger Besu allows Osprey to directly obtain a smart contract from Ethereum (through the smart contract's address) and instantiate a test over the original smart contract. Moreover, it allows to instantiate a test over the translated smart contract on Hyperledger Fabric and, in case of succeeding, it deploys the translated chaincode on the target blockchain [1].

3.3.2 Discussion. Osprey was specifically designed and developed to provide modularity and flexibility. These properties allow other developers to use this tool and build on top of it to add new sets of features. For instance, developers can used Osprey to translate Solidity smart contracts to other programming languages besides Typescript chaincode without dealing with the intermediate step (i.e., the construction of the Abstract Syntax Tree from the original smart contract code), since that is already implemented.

On the other hand, Osprey has some drawbacks. At its current state, it does not support all Solidity features. For instance, Osprey does not support data structures (e.g., *mappings* and *arrays*), multiple inheritance nor other Ethereum-specific features such as *payables*, EVM objects (e.g., *msg*, *tx*, etc.) and EVM functions (*transfer*, *send*, etc.). Moreover, Osprey is currently only capable of translating smart contracts written in Solidity into Hyperledger Fabric chaincode. However, as it as been mentioned before, its architecture allows to build more functionalities on top of the existing tool and, for example, add support for new programming languages.

This tool is particularly interesting to this study because the proposed solution of this document will try to exploit Osprey's modularity and flexibility to address its flaw of only allowing smart contract migration between Ethereum and Hyperledger Fabric. Section 4 will cover how the proposed tool will be built on top of Osprey to extend its functionalities by implementing smart contract migration between Hyperledger Fabric and Ethereum.

3.4 Summary

Table 2 summarizes the state-of-the-art solutions to blockchain interoperability and migration of smart contracts between heterogeneous blockchains that were covered along this section.

	Smart Contract Migration Patterns	**Sol2js**	**Osprey**
Type of work	Study	Tool	Tool
Brief	Published paper on migration of data and smart contracts across homogeneous and heterogeneous blockchains.	Translator tool developed to convert Solidity smart contracts into Javascript chaincode.	Migrating tool that translates Solidity smart contracts to Typescript chaincode.
Advantages	Relevant paper to the fields of blockchain interoperability and migration of smart contracts.	Helps developers and enterprises to migrate their applications and projects to another blockchain.	Helps developers and enterprises to migrate their applications and projects to another blockchain.
Disadvantages	It is only a study and nothing was actually implemented.	Does not support all Solidity features. Allows translation between Ethereum and Fabric only in one direction.	Does not support all Solidity features. Allows translation between Ethereum and Fabric only in one direction.

Table 2. Summary of related work

4 Solution

This section presents the proposed solution to the issues that have been addressed throughout this document. In particular, it approaches the problem of smart contract migration across heterogeneous blockchains (more precisely, between Hyperledger Fabric and Ethereum). In the first place, it will cover the project's architecture and layout, as well as how it is integrated with Hyperledger Cactus. Then, it will do an overview of the planned implementation and the different modules that compose the tool.

4.1 Overview

This solution will be built on top of Osprey to extend its features and functionalities. As mentioned in Section 3.3, Osprey enables the translation of smart contracts written in Solidity to Typescript chaincode. As such, it allows the migration of smart contracts from Ethereum to Hyperledger Fabric. However, the inverse conversion (i.e., from Hyperledger Fabric to Ethereum) is not yet supported. Therefore, the ultimate goal is to make Osprey a full two-way migrating tool for smart contracts written in Solidity and Typescript chaincode on the Ethereum and Hyperledger Fabric blockchain platforms, respectively. That was the main reason for choosing these two programming languages. Since Osprey currently allows translation from Solidity to Typescript, it would be valuable to extend the tool and add support for migration in the inverse direction. Thereby,

the future version of Osprey will allow developers and enterprises to migrate their applications from Hyperledger Fabric to Ethereum, and vice versa.

As such, the architecture of the project is the same as the Osprey system. Figure 2 shows how Osprey is integrated with Hyperledger Cactus as a plugin and how it interacts with Fabric, Besu and, subsequently, Ethereum.

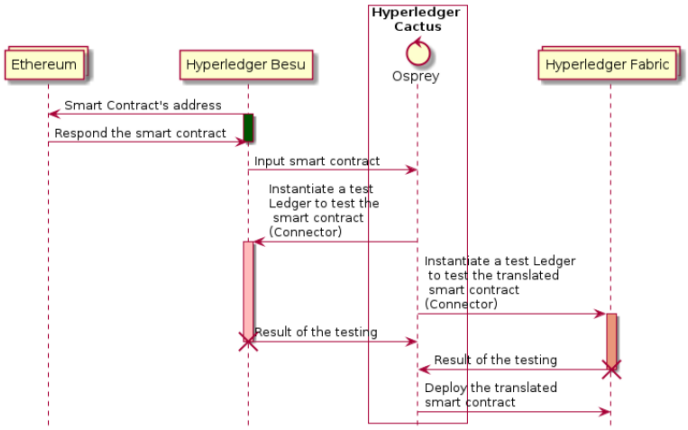


Figure 2. Sequence diagram [1]

Contrarily to the current implementation of Osprey, this solution will implement the inverse of its existing logic. Currently, as discussed in Section 3.3 and as pictured in Figure 2, Osprey takes advantage of Hyperledger Besu to acquire smart contracts from Ethereum and, at the end of the translation process, it deploys them on Fabric. This projects intends to implement the opposite path. That is, contracts are acquired from Hyperledger Fabric and, later, the translated version can be deployed on Ethereum (most likely on a testnet) through Hyperledger Besu. Table 3 summarizes the differences between the current and the future implementations of Osprey.

Features	Current version of Osprey	Future version of Osprey
Solidity → Typescript	✓	✓
Typescript → Solidity	✗	✓
Translation of other languages	✗	✗

Table 3. Comparison between versions of Osprey

In more detail, the upgraded version of Osprey will be able to receive a smart contract written in Typescript chaincode and parse it into an Abstract Syntax Tree, according to the syntax and rules of the programming language. The *parser* module is responsible for this first task. Then, the AST is properly interpreted

by the *converter* module and the solidity smart contract is constructed. The final step consists of testing the behavior of both the original and the translated smart contracts to demonstrate that the target contract kept the original intended behavior. This last stage is performed by the *tester* module. Figure 3 presents an overview of the aforementioned steps that take place during the translation process of the proposed solution.

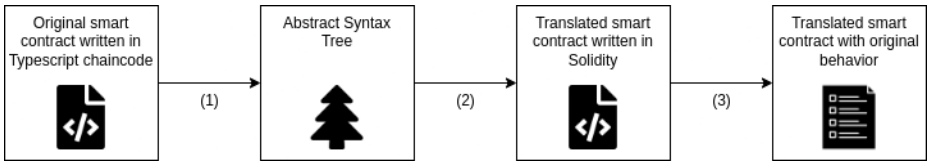


Figure 3. Flowchart of the solution

Ultimately, the work flow of the tool can be categorized into three distinct phases:

1. Translation phase;
   (a) Parsing;
   (b) Conversion.
2. Testing phase;
3. Deployment phase.

The first phase is subdivided into two stages that are respectively conducted by the *parser* and *converter* modules. These equate to steps 1 and 2 displayed by Figure 3. The parsing stage takes the code of the original smart contract and generates the Abstract Syntax Tree that represents that program. The conversion stage interprets the AST and constructs the target smart contract.

The second phase is accomplished by the *tester* module and is represented by the third step of Figure 3. As mentioned before, this phase ensures that the translated code kept the behavior of the original smart contract and executes as intended. This stage is particularly important since it validates if the tool managed to properly translate the original smart contract or not. If not, it means that the tool does not correctly handle a certain type of operation, and improvements and fixes should be applied.

The last phase is optional and consists of allowing the deployment of the translated contract on the target blockchain (i.e., Ethereum mainnet or testnets). This can be achieved because of the integration with Hyperledger Cactus, which provides mechanisms to deploy the translated smart contract on Ethereum through an interaction with Hyperledger Besu.

5 Evaluation

After being implemented, the proposed tool should be properly evaluated. This will be done in four different ways:

- By assessing the performance of the migrating tool according to two parameters:
  1. Throughput: the amount of smart contracts that can be translated within a unit of time.
  2. Latency: The average time it takes for the tool to translate a smart contract.
- By conducting a survey with a target audience of individuals with knowledge and interest in blockchain technology and smart contract programming about the execution and the output of the tool.
- By performing manual verifications to validate the execution and the output of the tool.
- By using the tool to translate a Solidity smart contract to Typescript chaincode and then using it again to translate it back to Solidity.

The objective of the first approach is to evaluate the performance of the tool, by measuring the average time of translations and the average amount of translations that can be done within a unit of time. Moreover, it may also be possible to reach some conclusions regarding how the complexity of a smart contract and its operations can affect the throughput and the latency of the tool. To do so, a set of smart contracts will be selected from *GitHub* and other online sources to create a dataset suitable to these experiments. The tests that were carried out to evaluate the current version of Osprey show that the tool takes an average of 3.68 milliseconds to translate a fairly simple smart contract. These tests were executed with a dataset of 13 smart contracts that were translated 10000 times [1]. Therefore, the future version of Osprey should be evaluated with a dataset of smart contracts of similar complexity and should have a similar performance. This implies that the upgraded version of Osprey should, on average, take about the same time to translate a smart contract (i.e., both measurements should have the same order of magnitude) as the current one.

The second approach will allow to measure the readability of the output of the proposed tool and perhaps help drawing some conclusions regarding a generalized opinion about the utility of the tool and the value it may bring to the ecosystem.

Furthermore, manual verifications can also be important to validate the behavior of the translated smart contract and the appropriate functioning of the tool. This can be specially significant in scenarios where the tool fails to automatically assert the validity of the translated code (i.e., whether or not the target smart contract does what it is expected to do). Such cases can happen in two possible scenarios:

- The target smart contract was not properly translated and, in fact, it does not do what was initially intended;
- The target smart contract was properly translated but the automatic tests failed to assert that.

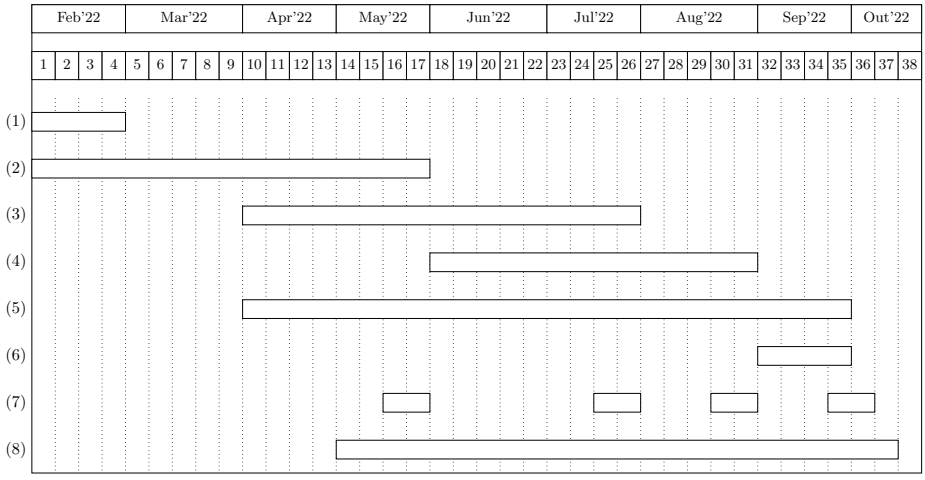
In such scenarios, resorting to manual validation can help assess what went wrong: the translation process or the testing process. This can be achieved, for

instance, through manual deployment of the translated smart contract on the target blockchain and respective execution of its functions. If the execution of the translated code is not aligned with the original behavior of the smart contract, it means that the translation process failed. Otherwise, it indicates that the testing process was miscarried.

Lastly, when converting a Solidity smart contract to Typescript chaincode and then back to Solidity, the final output of the tool should match the original input. In theory, for each Solidity smart contract, the tool should be able to translate it to Typescript chaincode and then translate it back to Solidity, outputting the same code that was originally inputted.

6 Work Schedule

This section will go over the work schedule that that is expected to be conducted during the implementation of the proposed tool. Figure 4 presents the estimated time that each planed task of the project will take to be completed throughout the semester.



Legend:
(1) - Familiarization with Osprey source code
(2) - Parser implementation
(3) - Converter implementation
(4) - Tester implementation
(5) - Tool review
(6) - Last improvements and fixes
(7) - Tool evaluation
(8) - Writing dissertation

Figure 4. Planned Work Schedule

The initial focus of the project will be on the implementation of the parser component that will allow to obtain an Abstract Syntax Tree from the original Typescript chaincode. Then, the implementation of the converter will begin. By integrating this component with the parser, it will be possible to convert the AST into Solidity code. Next, comes the implementation of the tester module that will allow to compare the behavior of both the original and the target smart contracts. In the meantime, the tool should be thoroughly reviewed along the development of each component. Lastly, any last improvements and fixes should be applied and the tool should be properly evaluated based on the defined criteria. When a significant part of the project has already been done, the task of writing the dissertation should begin. This will be the last phase of the project to be finalized.

7 Conclusion

Blockchain interoperability and smart contract migration have become outstanding subjects in blockchain technology. The latter, in particular, can unquestionably help developers once fully implemented and settled in this space. This industry can benefit a lot from it since it can significantly reduce the time of development and allow both developers and enterprises to move across different blockchains or deploy their projects on several platforms at the same time without having to redo previous work (i.e., manually code the same algorithm and applications in other programming languages).

The proposed solution extends an existing tool called Osprey, that is already able to migrate smart contracts from Ethereum to Hyperledger Fabric, by translating Solidity code to Typescript chaincode. By extending this tool to also support smart contract migration from Hyperledger Fabric to Ethereum through translation of Typescript chaincode to Solidity, Osprey will become a full two-way migrating tool. Therefore, enabling developers and enterprises to move between these two platforms in both directions.

References

1. Abrunhosa, L.: Migrating Smart Contracts Across Heterogeneous Blockchains. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa (2021)
2. Bandara, H.D., Xu, X., Weber, I.: Patterns for blockchain data migration (2019), https://arxiv.org/pdf/1906.00239.pdf
3. BandProtocol: Bandchain whitepaper (2019), https://docs.bandchain.org/whitepaper/
4. Belchior, R.: Blockchains privadas e interoperabilidade (2020), https://www.youtube.com/watch?v=2IWrfgw-uiw
5. Belchior, R., Vasconcelos, A., Guerreiro, S., Correia, M.: A survey on blockchain interoperability: Past, present, and future trends (2020), https://arxiv.org/abs/2005.14282
6. Binance: Binance smart chain: A parallel binance chain to enable smart contracts (2020), https://github.com/binance-chain/whitepaper/blob/master/WHITEPAPER.md
7. Buterin, V.: Ethereum: A next-generation smart contract and decentralized application platform (2014), https://ethereum.org/en/whitepaper/
8. Buterin, V.: A rollup-centric ethereum roadmap (2020), https://ethereum-magicians.org/t/a-rollup-centric-ethereum-roadmap/4698
9. Buterin, V.: An incomplete guide to rollups (2021), https://vitalik.ca/general/2021/01/05/rollup.html
10. Buterin, V.: Why sharding is great: Demystifying the technical properties (2021), https://vitalik.ca/general/2021/04/07/sharding.html
11. Buterin, V., Conner, E., Dudley, R., Slipper, M., Norden, I., Bakhta, A.: Eip-1559: Fee market change for eth 1.0 chain (2019), https://eips.ethereum.org/EIPS/eip-1559
12. CertiK: The blockchain trilemma: Decentralized, scalable, and secure? (2019), https://medium.com/certik/the-blockchain-trilemma-decentralized-scalable-and-secure-e9d8c41a87b3
13. Chainlink: Introducing the cross-chain interoperability protocol (ccip) (2021), https://blog.chain.link/introducing-the-cross-chain-interoperability-protocol-ccip/
14. Chase, B., MacBrough, E.: Analysis of the xrp ledger consensus protocol (2018), https://arxiv.org/pdf/1802.07242.pdf
15. Choi, S.M., Park, J., Nguyen, Q., Cronje, A.: Fantom: A scalable framework for asynchronous distributed systems (2018), https://arxiv.org/pdf/1810.10360.pdf
16. ConsenSys: Consensys/quorum: A permissioned implementation of ethereum supporting data privacy, https://github.com/ConsenSys/quorum
17. Corda: Corda/corda: Corda is an open source blockchain project, designed for business from the start, https://github.com/corda/corda
18. Ellis, S., Juels, A., Nazarov, S.: Chainlink: A decentralized oracle network (2017), https://research.chain.link/whitepaper-v1.pdf
19. Engineer, S.C.: Solidity by example, https://solidity-by-example.org/
20. Entriken, W., Shirley, D., Evans, J., Sachs, N.: Eip-721: Non-fungible token standard (2018), https://eips.ethereum.org/EIPS/eip-721
21. Ethereum: The eth2 upgrades: Upgrading ethereum to radical new heights, https://ethereum.org/en/eth2/

22. Ethereum: Sidechains, https://ethereum.org/en/developers/docs/scaling/sidechains/
23. Ethereum: State channels, https://ethereum.org/en/developers/docs/scaling/state-channels/
24. Ethereum: Smart contract languages (2021), https://ethereum.org/en/developers/docs/smart-contracts/languages/
25. Glassnode: Bitcoin: Number of transactions, https://studio.glassnode.com/metrics?a=BTC&category=Transactions&m=transactions.Count
26. Glassnode: Ethereum: Number of transactions, https://studio.glassnode.com/metrics?a=ETH&category=Transactions&m=transactions.Count
27. Hyperledger: Hyperledger/besu: An enterprise-grade java-based, apache 2.0 licensed ethereum client, https://github.com/hyperledger/besu/
28. Hyperledger: Hyperledger/cactus: Hyperledger cactus is a new approach to the blockchain interoperability problem, https://github.com/hyperledger/cactus
29. Hyperledger: Hyperledger/fabric: Hyperledger fabric is an enterprise-grade permissioned distributed ledger framework for developing solutions and applications, https://github.com/hyperledger/fabric
30. Hyperledger: Open source blockchain technologies, https://www.hyperledger.org/
31. Hyperledger-Labs: Hyperledger-labs/solidity2chaincode: This tool converts solidity contract into javascript chaincode through source-to-source translation for running them onto hyperledger fabric, https://github.com/hyperledger-labs/solidity2chaincode
32. HyperledgerLabs: Hyperledger-labs/university-course: A hyperledger lab focused developing materials for a university course, https://github.com/hyperledger-labs/university-course
33. Kanani, J., Nailwal, S., Arjun, A.: Matic whitepaper (2018), https://github.com/maticnetwork/whitepaper
34. Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: A provably secure proof-of-stake blockchain protocol (2019), https://eprint.iacr.org/2016/889.pdf
35. Kwon, J., Buchman, E.: Cosmos: A network of distributed ledgers (2016), https://github.com/cosmos/cosmos/blob/master/WHITEPAPER.md
36. Lafourcade, P., Lombard-Platet, M.: About blockchain interoperability (2020), https://eprint.iacr.org/2020/643.pdf
37. Lopp, J.: What are the key properties of bitcoin? (2020), https://nakamoto.com/what-are-the-key-properties-of-bitcoin/
38. Mihal, D.: https://money-movers.info/
39. Mihal, D.: Crypto fees, https://cryptofees.info/
40. Min, A.: Hash time locked contracts (htlcs) explained (2019), https://liquality.io/blog/hash-time-locked-contracts-htlcs-explained/
41. Multichain: Multichain is the ultimate router for web3 (2020), https://docs.multichain.org/
42. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008), https://bitcoin.org/bitcoin.pdf
43. OffchainLabs: Inside arbitrum, https://developer.offchainlabs.com/docs/inside_arbitrum
44. OpenZeppelin: Openzeppelin/openzeppelin-contracts: Openzeppelin contracts is a library for secure smart contract development, https://github.com/OpenZeppelin/openzeppelin-contracts

45. Optimism: Contract overview, https://community.optimism.io/docs/protocol/protocol-2.0.html
46. Ripple: Implementing the interledger protocol in ripple (2015), https://ripple.com/insights/views/implementing-the-interledger-protocol/
47. Sekniqi, K., Laine, D., Buttolph, S., Sirer, E.G.: Avalanche platform whitepaper (2020), https://www.avalabs.org/whitepapers
48. Sergeenkov, A.: What is an automated market maker? (2021), https://www.coindesk.com/learn/2021/08/20/what-is-an-automated-market-maker/
49. Smartbugs: Smartbugs/dataset at master · smartbugs/smartbugs, https://github.com/smartbugs/smartbugs/tree/master/dataset
50. Solidity: https://docs.soliditylang.org/en/v0.8.11/
51. StarkWare: Starknet and cairo documentation, https://starknet.io/docs/
52. Synapse: Welcome to synapse (2021), https://docs.synapseprotocol.com/
53. Szabo, N.: Formalizing and securing relationships on public networks. First Monday **2**(9) (1997). https://doi.org/10.5210/fm.v2i9.548
54. Team, H.: Harmony: Technical whitepaper, https://harmony.one/whitepaper.pdf
55. THORChain: A decentralised liquidity network (2020), https://github.com/thorchain/Resources/blob/master/Whitepapers/THORChain-Whitepaper-May2020.pdf
56. Vogelsteller, F., Buterin, V.: Eip-20: Token standard (2015), https://eips.ethereum.org/EIPS/eip-20
57. Wikipedia: Cryptocurrency, https://en.wikipedia.org/wiki/Cryptocurrency
58. Wikipedia: Fiat money, https://en.wikipedia.org/wiki/Fiat_money
59. Wood, G.: Polkadot: Vision for a heterogeneous multi-chain framework, https://polkadot.network/PolkaDotPaper.pdf
60. Yakovenko, A.: Solana: A new architecture for a high performance blockchain, https://solana.com/solana-whitepaper.pdf
61. Zafar, M.A., Sher, F., Janjua, M.U., Baset, S.: Sol2js: Translating solidity contracts into javascript for hyperledger fabric. Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers (2018). https://doi.org/10.1145/3284764.3284768
62. Zamyatin, A., Al-Bassam, M., Zindros, D., Kokoris-Kogias, E., Moreno-Sanchez, P., Kiayias, A., Knottenbelt, W.J.: Sok: Communication across distributed ledgers (2019), https://eprint.iacr.org/2019/1128.pdf
63. zkSync: Introduction to zksync for developers, https://zksync.io/dev/