Comparison of Programming Languages in Blockchain

Anant Shukla, Prayuj Pillai

May 16, 2023

Abstract

Solidity, TEAL, Rust, and C++ are all programming languages used for developing smart contracts. Each of these languages has its own features and characteristics that differentiate it from the others. We intend to investigate how these languages might agree and differ on key points in a blockchain context, such as security, resource allocation, Turing completeness, and programming paradigms.

1 Introduction

Here is a brief primer on the languages in question:

1. TEAL: TEAL is a low-level language used for developing smart contracts on the Algorand blockchain. It is a stack-based language that is designed to be simple and efficient. These contracts can be written directly or with Python using the PyTeal library. TEAL is not Turing-complete but can still be used to write complex smart contracts. [1, 2]
2. Solidity: Solidity is the most popular language used for programming smart contracts on the Ethereum platform. It is a high-level language that is easy to learn and has a syntax similar to that of JavaScript. It also supports inheritance, libraries, and complex user-defined types. Solidity is Turing-complete, meaning that it can compute any algorithm that a regular computer can. [3, 4]
3. Rust is a popular programming language in the blockchain ecosystem due to its strong emphasis on performance, reliability, and security. Many blockchain projects use Rust to build their core protocol, smart contracts, and other blockchain-related software, such as [5].
4. C++: C++ is a widely used programming language that is often used for developing system software, games, and other performance-critical applications. It is a low-level language that is efficient and provides direct access to the computer's hardware. C++ can be used for developing smart contracts, but it requires a deep understanding of the blockchain platform and its programming interfaces. [6]

2 Topics of comparison

Although these languages differ on many fronts, we will focus on the following factors:

- Turing Completeness
- Programming Paradigm
- Resource Allocation (Gas)
- Security

We will take a deep dive into each of the factors in the following sections.

3 Turing Completeness

A programming language is said to be Turing-complete if it can be used to simulate any Turing machine. In practical terms, a Turing-complete language is one that is powerful enough to express any computable function or program.

Not all blockchain languages are Turing complete. For example, Bitcoin's "Script" language is not Turing complete. It may be advantageous to prevent certain capabilities, such as allowing infinite loops, and also narrows the scope of vulnerabilities that may be introduced with the use of that language.

1. TEAL: TEAL is a non-Turing complete language that allows branch forwards but prevents recursive logic to maximize safety and performance. This implies that TEAL cannot perform an arbitrary amount of computational steps and it cannot compute procedures that would use unbounded memory. This is advantageous since it does not need a concept of 'gas' or resource fee to perform operations on the blockchain.
2. Solidity: Solidity is a quasi-Turing complete language. The language is designed to be Turing complete, however, the reason for 'quasi' will be discussed in the section we discuss 'Gas'.
3. Rust and C++: Both Rust and C++ are Turing complete.

4 Programming Paradigm

Programming paradigms are a way to classify programming languages based on their features. Languages can be classified into multiple paradigms, some common paradigms are:

- Imperative: Programmers instruct the machine on how to change the state.
  - Procedural: Instructions are grouped into procedures, like in FORTRAN and COBOL
  - Object-oriented: Instructions are grouped into states (objects) they are part of, e.g. Java
- Declarative: Programmers instruct the machine on how to process input data.
  - Functional: Result is the value returned from successive function calls. These functional calls are deterministic. Example: Haskell

In the context of the languages we are comparing:

1. TEAL is a Domain-specific language, not Object Oriented.
2. Solidity, Rust, and C++ are all Multi paradigm languages, in the sense that they support both functional and object-oriented languages.

5 Resource allocation (Gas)

Gas refers to the unit that measures the amount of computational effort required to execute specific operations on a blockchain network.

1. TEAL has a flat fee for transactions (0.001 Algo) [which equals about $0.0017 at the time of writing]. All fees go into a "fee sink" wallet which the foundation controls - it will later be decided by the foundation and other participants as to what to do with fees accumulated in the sink wallet. Gas fees for groups of transactions can be grouped e.g. 4 transactions can be made for one fee of 0.001 Algo, as long as the transactions put together still satisfy the memory and other resource limits of one transaction.
2. Solidity: used to pay for the computational resources required to execute smart contracts. Part of it goes to the validators that confirm transactions. The rest of it is burnt. One transaction roughly equals $10-20 (Or even more, depending on network congestion).

3. Rust and C++: No inherent concept of gas fees. Will be subject to blockchain's gas fees depending on which chain the contracts are written for. For example, in the EOS.IO blockchain, you stake tokens for CPU usage, RAM storage and network bandwidth, and then you gain access to system resources proportional to the total amount of tokens staked by all other users for the same system resource at the same time.

6 Security

6.1 Solidity

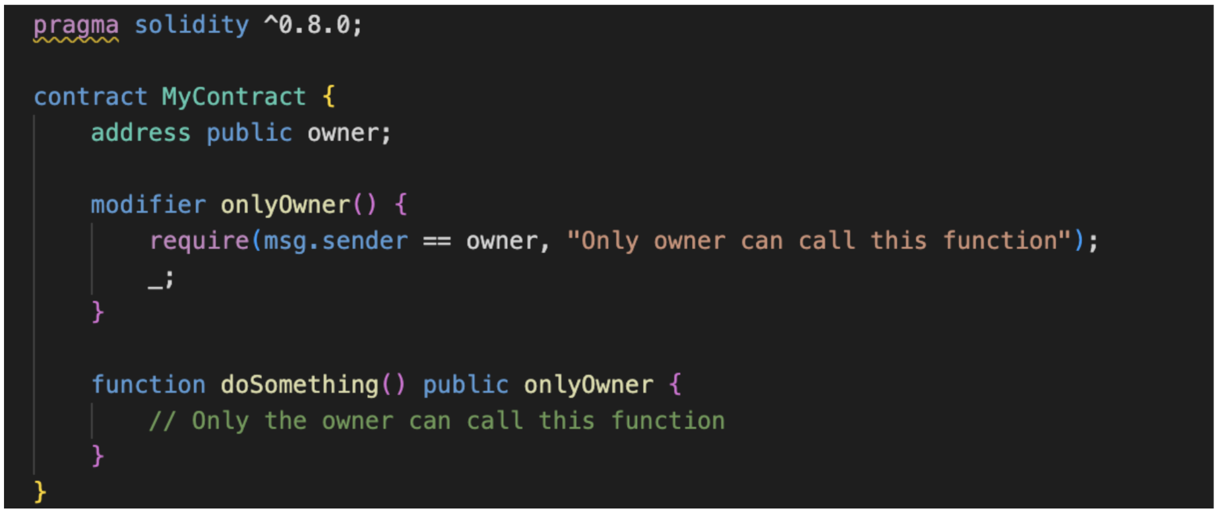
Solidity offers various features that contribute to its security and functionality. Solidity includes libraries like Safemath, which protect against underflow and overflow attacks, and this has become a default feature starting from version 0.8. Access control mechanisms are in place to prevent external contracts from interfering with the execution of smart contracts. Input validation is another feature that Solidity provides, which helps prevent re-entrancy attacks, ensuring that contracts are not reentered during critical execution points. Gas helps prevent distributed denial-of-service (DDoS) attacks by requiring transaction submitters to pay for computational resources. Furthermore, smart contracts in Solidity are immutable by default, meaning their logic cannot be changed on the fly, enhancing security and predictability. These features contribute to the resilience and security of smart contracts and transactions in Ethereum.

6.1.1 Access Control

Solidity provides several ways to control access to functions and data in a smart contract, including:

- public functions: These can be called by anyone, both from within the contract and from outside it.
- external functions: These can only be called from outside the contract.
- internal functions: These can only be called from within the contract, or from derived contracts.
- private functions: These can only be called from within the contract.

Here's an example of how access control can be used in Solidity:



```
pragma solidity ^0.8.0;

contract MyContract {
    address public owner;

    modifier onlyOwner() {
        require(msg.sender == owner, "Only owner can call this function");
        _;
    }

    function doSomething() public onlyOwner {
        // Only the owner can call this function
    }
}
```

Figure 1: An example of using function modifiers for security

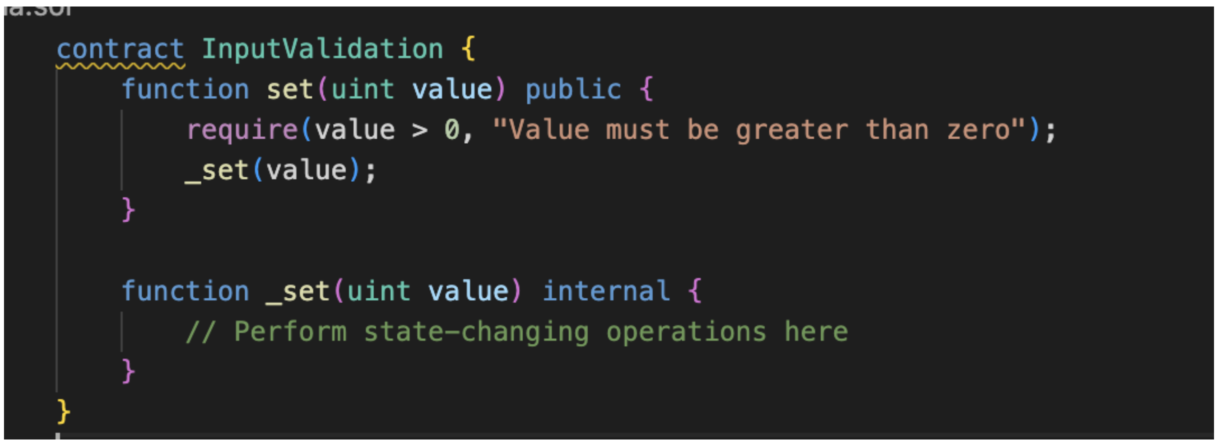
The onlyOwner modifier is used to restrict access to the transferOwnership function. Only the owner of the contract (the account that deployed it) can call this function. This helps prevent unauthorized changes to the contract state.

6.1.2 Input Validation

Solidity also provides several ways to validate inputs to functions in a smart contract, including:

- require: This function checks a condition and throws an exception if it evaluates to false.
- assert: This function checks a condition and throws an exception if it evaluates to false, but also consumes all remaining gas.
- revert: This function aborts the execution of the current transaction and reverts any changes made so far.

Here's an example of how input validation can be used in Solidity:



```
contract InputValidation {
    function set(uint value) public {
        require(value > 0, "Value must be greater than zero");
        _set(value);
    }

    function _set(uint value) internal {
        // Perform state-changing operations here
    }
}
```

Figure 2: An example of using require for input validation

6.2 TEAL

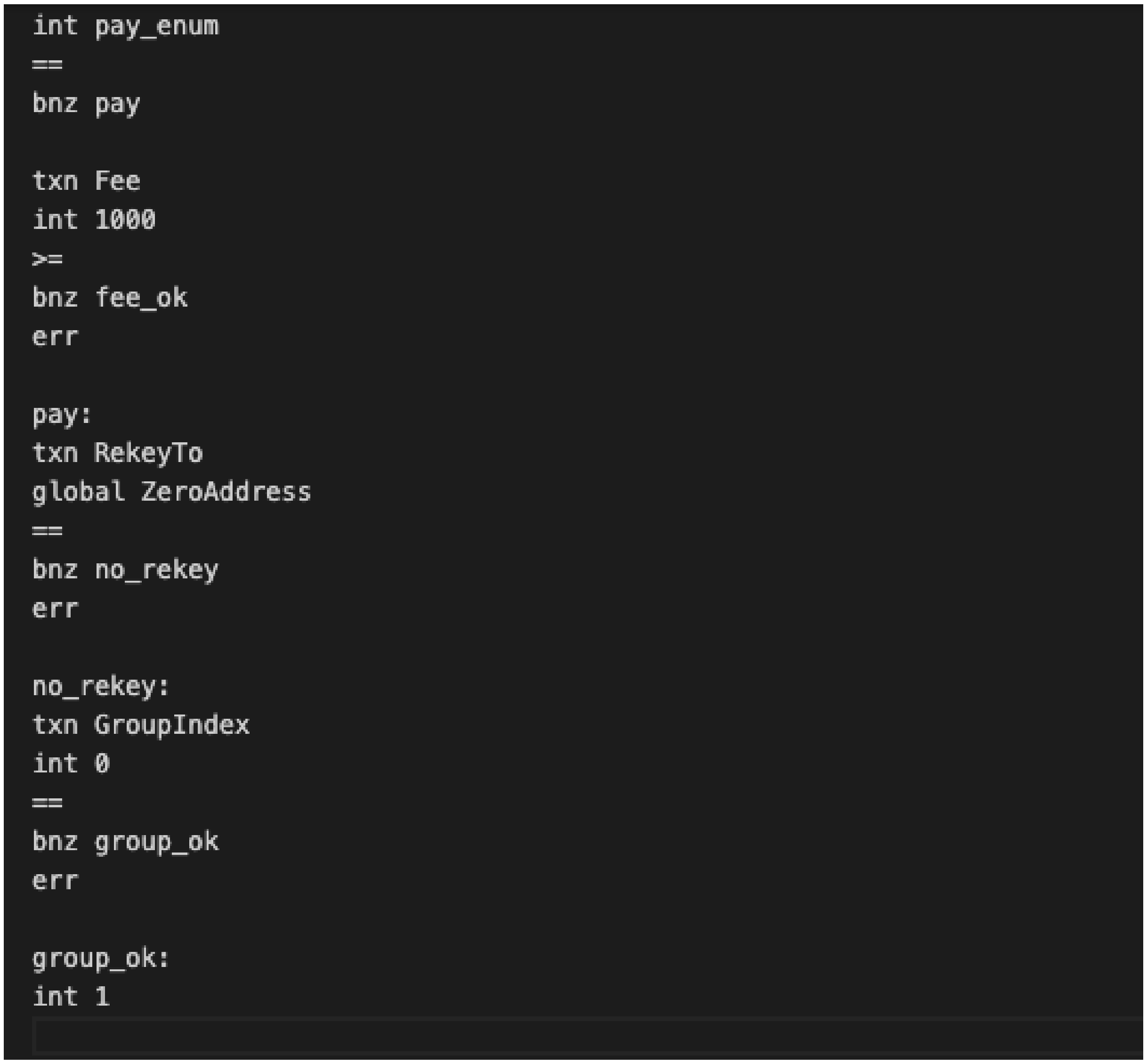
TEAL offers a range of features that contribute to its functionality and security. It provides built-in support for atomic transfers, ensuring that transactions either succeed entirely or fail completely. It also offers the ability to rekey, allowing for the remapping of a static public address to rotating private addresses internally. Algorand's state proofs enable the verification of a set of transactions without requiring the operation of a full node, offering transparency and efficiency. TEAL also allows for mutable smart contracts, providing the flexibility to modify contract logic and adapt to changing requirements. These features collectively enhance the robustness and usability of smart contracts and transactions on the Algorand blockchain.

6.2.1 Logic Constraints

TEAL provides several logic constraints that can be used to enforce security requirements, including:

- txn TypeEnum: This checks the type of the transaction, such as pay, keyreg, or appl.
- txn Fee: This checks the fee paid for the transaction, and can be used to enforce minimum and maximum fees.
- txn RekeyTo: This checks whether the transaction is trying to rekey the sender's account, and can be used to prevent unauthorized changes to the account.
- txn GroupIndex: This checks whether the transaction is part of a group, and can be used to enforce atomicity requirements.

Here's an example of how logic constraints can be used in TEAL:



```
int pay_enum
==
bnz pay

txn Fee
int 1000
>=
bnz fee_ok
err

pay:
txn RekeyTo
global ZeroAddress
==
bnz no_rekey
err

no_rekey:
txn GroupIndex
int 0
==
bnz group_ok
err

group_ok:
int 1
```

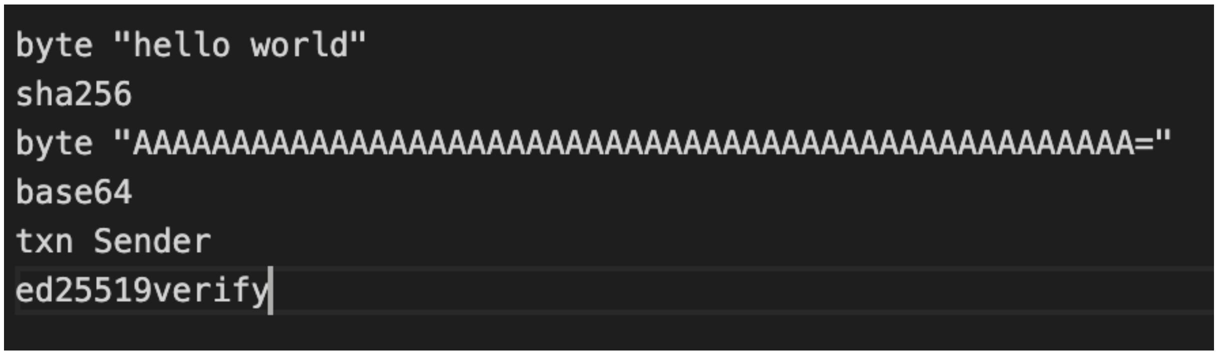
Figure 3: An example of using logic constraints for TEAL contract validation

The TEAL program checks that the transaction type is pay, the fee is greater than or equal to 1000 microalgos, the transaction is not trying to rekey the sender's account, and the transaction is not part of a group. If any of these conditions are not met, the program will throw an error and the transaction will fail.

6.2.2 Signature Verification

TEAL also provides built-in support for signature verification, which can be used to enforce access control and prevent unauthorized changes to state. This is done using the sig and ed25519verify opcodes.

Here's an example of how signature verification can be used in TEAL:



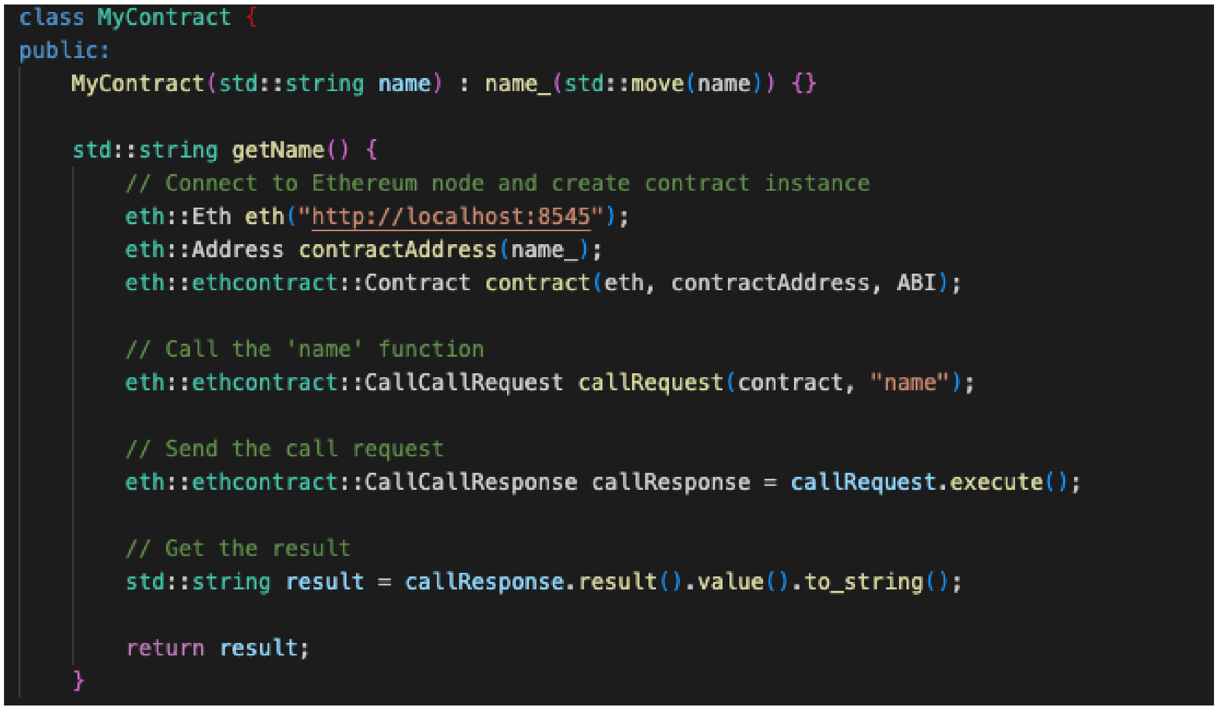
```
byte "hello world"
sha256
byte "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA="
base64
txn Sender
ed25519verify
```

Figure 4: An example of using require for input validation

The TEAL program hashes the string ”hello world” using the SHA-256 algorithm, and then verifies that the resulting hash matches a precomputed value. It then verifies that the transaction was signed by the sender's private key using the ed25519verify opcode. If either of these checks fail, the program will throw an error and the transaction will fail. (Note that ”hello world” may not actually hash to the byte shown above - it's just to illustrate how it may be done.)

6.3 C++

C++ offers strong typing, which helps enforce type safety and reduces the risk of type-related vulnerabilities. It also provides a range of secure cryptographic libraries and tools, such as the OpenSSL library, which can be leveraged to build secure blockchain applications. C++, however, does have some drawbacks. One of them is memory leaks and memory management, where developers are responsible for correctly allocating and freeing memory to avoid potential issues. Additionally, C++ exhibits unpredictable behavior, allowing access out of bounds in arrays and potentially displaying garbage values. These factors highlight the importance of diligent memory management and careful array handling when developing in C++.



```cpp
class MyContract {
public:
    MyContract(std::string name) : name_(std::move(name)) {}

    std::string getName() {
        // Connect to Ethereum node and create contract instance
        eth::Eth eth("http://localhost:8545");
        eth::Address contractAddress(name_);
        eth::ethcontract::Contract contract(eth, contractAddress, ABI);

        // Call the 'name' function
        eth::ethcontract::CallCallRequest callRequest(contract, "name");

        // Send the call request
        eth::ethcontract::CallCallResponse callResponse = callRequest.execute();

        // Get the result
        std::string result = callResponse.result().value().to_string();

        return result;
    }
```

Figure 5: An example of a basic C++ contract using the aleth library

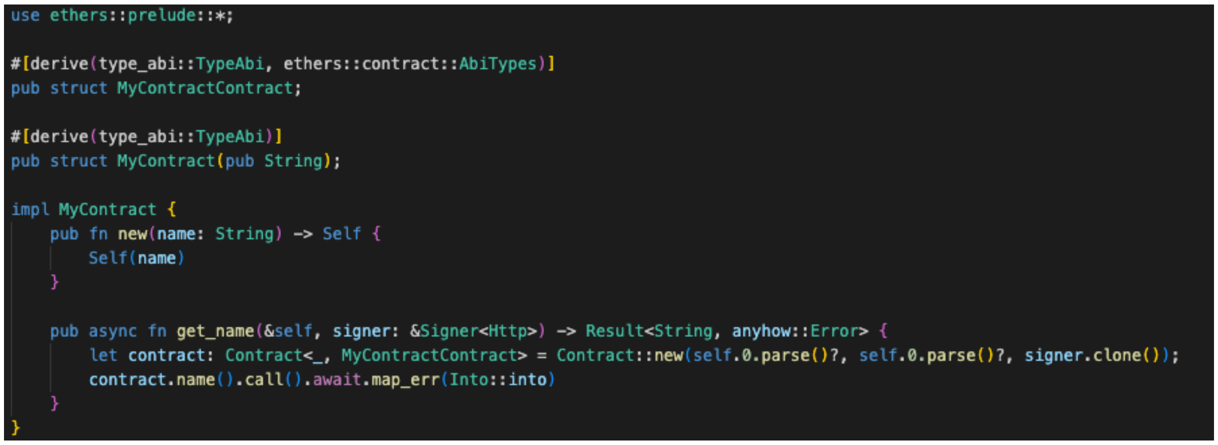
6.4 Rust

Similar to C++, Rust has features like type safety, cryptographic libraries, etc. along with safe memory management. This is possible due to its ownership model. In Rust, every value has a single owner, and ownership can only be transferred, not copied.

Rust's ownership and borrowing system can help prevent common memory errors such as null pointer dereferencing and use-after-free bugs, which can be especially important in a blockchain context where security is paramount. Concurrency is also possible in Rust since they provide access to different threads and channels.

Rust's built-in support for safe concurrency through its use of threads and channels can be especially useful in a blockchain context where multiple transactions may be processed concurrently. Rust's ownership and borrowing system can help prevent data races and other synchronization issues that can lead to security vulnerabilities in a concurrent context.

On the other hand, there is currently no standard for smart contract portability in Rust - making it hard to translate from rust to other smart contract languages.



```rust
use ethers::prelude::*;

#[derive(type_abi::TypeAbi, ethers::contract::AbiTypes)]
pub struct MyContractContract;

#[derive(type_abi::TypeAbi)]
pub struct MyContract(pub String);

impl MyContract {
    pub fn new(name: String) -> Self {
        Self(name)
    }

    pub async fn get_name(&self, signer: &Signer<Http>) -> Result<String, anyhow::Error> {
        let contract: Contract<_, MyContractContract> = Contract::new(self.0.parse()?, self.0.parse()?, signer.clone());
        contract.name().call().await.map_err(Into::into)
    }
}
```

Figure 6: An example of a basic rust contract using the ethers library

7 Conclusion

While Rust, C++, Solidity, and TEAL serve different purposes and are build for different domains, they can all be effectively used in a blockchain context. The choice of language depends on the specific requirements of the project, considering factors such as performance, safety, concurrency, blockchain compatibility, and the ecosystem available.

As blockchain technology and smart contracts continu to evolve, it is crucial to consider the strengths and weaknesses of different languages and select the most appropriate one for the task at hand. Ultimately, the success of a project lies not only in the choice of language but also in the skills and expertise of the developers who wield these powerful tools.

References

[1] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, “Algorand: Scaling byzantine agreements for cryptocurrencies,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP ’17, (New York, NY, USA), p. 51–68, Association for Computing Machinery, 2017.

[2] Y. Xu, T. Slaats, B. Düdder, S. Debois, and H. Wu, “Distributed and adversarial resistant workflow execution on the algorand blockchain,” 2022.

[3] D. Macrinici, C. Cartofeanu, and S. Gao, “Smart contract applications within blockchain technology: A systematic mapping study,” *Telematics and Informatics*, vol. 35, no. 8, pp. 2337–2354, 2018.

[4] P. Hegedűs, “Towards analyzing the complexity landscape of solidity based ethereum smart contracts,” *Technologies*, vol. 7, no. 1, 2019.

[5] P. Ning and B. Qin, “Stuck-me-not: A deadlock detector on blockchain software in rust,” *Procedia Computer Science*, vol. 177, pp. 599–604, 2020. The 11th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2020) / The 10th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH 2020) / Affiliated Workshops.

[6] N. H. et. Al, *Smart Contracts in C++.*, 2021. Version 2.3.