Smart Contract Programming Languages on Blockchains: An Empirical Evaluation of Usability and Security

Reza M. Parizi[1(✉)], Amritraj[1], and Ali Dehghantanha[2]

[1] Department of Software Engineering and Game Development, Kennesaw State University, Kennesaw, GA 30060, USA
rparizi1@kennesaw.edu, amritra@students.kennesaw.edu
[2] Department of Computer Science, University of Sheffield, Sheffield, UK
a.dehghantanha@sheffield.ac.uk

Abstract. Blockchain is a promising infrastructural technology that is finding its way into a growing number of domains like big data, finance, and medical. While blockchain has come to be thought of primarily as the foundation for Bitcoin, it has evolved far beyond underpinning the virtual currency. As it becomes progressively popular, the need for effective programming means would be more demanding. Blockchain programming as a core means provides accounts of the 'code is law' that specifies agreements between parties and allows its stakeholders to still trust the platform to execute the agreed-upon contract (known as smart contract) as expected. Although it seems straightforward in theory, it is hardly the case when it comes to real-life situations. There have been several instances that show smart contracts are riddled with issues and vulnerabilities in code, causing damages. What's for sure is lacking is that the existing languages are not living up to the point to be able to unleash the full potential of the blockchain, as often have resulted in buggy code with a steep learning curve for developers. This denotes that the current research on contract development is not sufficient and is still in a stage of infancy. In order to advance the state of the research in this area, an evaluation of the current state-of-the-art practices in a thorough and experimental manner is required. Thus, the objective of this paper is to give a comprehensive analysis of such domain-specific programming practices from critical points of usability and security to provide a working guideline for newcomers and researchers.

Keywords: Blockchain · Blockchain coding · Smart contract platforms
Smart contract programming · Decentralized computing and development

1 Introduction

Blockchain [1] is a new trend rising fast from the community and the enterprise world. A blockchain is theoretically an incremental list of records called blocks which are linked together and secured using cryptography, forming a chain in the process. Copies of this chain are stored across several peers on a network who can all see the chain and its contents. To add a new block, a peer must find a key to a random pattern generated

© Springer International Publishing AG, part of Springer Nature 2018
S. Chen et al. (Eds.): ICBC 2018, LNCS 10974, pp. 75–91, 2018.
https://doi.org/10.1007/978-3-319-94478-4_6

using cryptography and verify the block itself. As soon as a peer adds a new block, it also broadcasts this addition to all the other peers on the network, so they can update their copies of the blockchain.

Blockchain has already disrupted a wide range of industries including Finance, Cloud computing, Privacy, Security etc. Also, in the recent years, an interesting new application of blockchain has surfaced, i.e. Smart contract [2]. Smart contracts are self-executing contracts where the terms of the agreement between multiple parties are directly written into lines of code. The code and the agreements contained therein exist across a blockchain network. Smart contracts allow trusted transactions and agreements to be carried out among disparate, anonymous parties without the need for a central authority, legal system, or an external enforcement mechanism. They render transactions traceable, transparent, and irreversible. Recognition of the unique challenges of smart contract programming has inspired developers to create domain-specific languages, such as Solidity [3] to ease development.

Although it is a promising domain, Smart contracts, in its first decade has been plagued by unfortunate incidents. In June 2016, vulnerabilities in the DAO code was exploited to empty out more than 2 Million (40 Million USD) ether [4]. The attack took advantage of the reentrancy problem in the 'splitDAO' function of the code. Since, the program was not designed carefully, a call to the function that behaved as a regular call was modified into a recursive call and used to make multiple withdrawals when only one was to be authorized.

Also, in November 2017, a developer [5] whilst fixing a bug that let attackers steal 32 million USD from a few multi-signature wallets accidentally left a second bug in the system that allowed one user to become the sole owner of every single multi-signature wallet. Realizing the mistake, the developers tried to fix damages by deleting the program instead of returning the funds to their original owners. This act of deletion of the program simply locked all the funds in those multi-signature wallets permanently. Unlike most cryptocurrency hacks, however, the money was not deliberately taken instead, it was permanently locked by accident and lack of understanding of the program.

The above incidents show that even the most experienced developers can leave behind security vulnerabilities and bugs that are exploitable and failure prone. Thus, there is still a steep learning curve for developers when it comes to contract programming. This steep learning curve makes it even more difficult for new developers to write correct and safe contracts. As of current date, the state of empirical studies in the domain of smart contract development is still in infancy. Hence, the objective of this paper is to take this initiative by providing an empirical evaluation of smart contract programming languages, in order to shed light on future directions of its development research, education and practices. To this end, we assessed the usability and security vulnerability aspects of three domain-specific languages namely Solidity, Pact and Liquidity (see Sect. 2). The results demonstrated that although Solidity is the most usable language for a new developer to program smart contracts, it is the least secure language to vulnerabilities. While, Liquidity and Pact demonstrated better security results, implying it is harder for new developers to leave behind bugs and security vulnerabilities when working with Pact and Liquidity, but they show less usability compared to Solidity.

The remainder of this paper is organized as follows. Section 2 gives an overview of the smart contract programming languages and state-of-the-art practices for building smart contracts; Sect. 3 presents and describes the details of the experiment and its evaluation results; Sect. 4 presents the related work; and lastly, Sect. 5 reports the conclusion and future work.

2 Smart Contract Programming Languages

In this section, we discuss widely used smart contract programming languages namely Solidity, Pact, Liquidity. We have also discussed sample contracts implemented with each of the mentioned programming languages to provide an insight into real-world contract development.

2.1 Solidity and the EVM

Solidity [3] is a statically-typed programming language with a similar syntax to ECMAScript (JavaScript) built for writing smart contracts. It is the primary choice language for implementing smart contracts on the Ethereum [6] platform.

Ethereum is an open source, decentralized platform for building smart contracts. It facilitates the development and execution of complex applications such as financial exchanges and insurance contracts on a distributed platform. The core of Ethereum is the Ethereum Virtual Machine (EVM), which executes code of random algorithmic complexity. Solidity is designed for developing smart contracts that run on the EVM. Solidity contracts are first compiled to bytecode which is ultimately executed on the EVM.

Like other blockchains, Ethereum includes a peer-to-peer network protocol. The Ethereum blockchain database is maintained and updated by several nodes connected to the network. Every node on the network runs the EVM and executes the same set of instructions.

The Ethereum platform itself is featureless or value-agnostic. It is up to organizations and developers to decide what it should be used for. However, certain application types benefit more than others from Ethereum's capabilities. Specifically, Ethereum is suited for applications that automate direct interaction between peers or facilitate coordinated group action across a network. For instance, applications for coordinating peer-to-peer marketplaces, or the automation of complex financial contracts. When it comes to programming on Ethereum, there are some key points to notice from the Ethereum Design Rationale document [7].

Sample Contract Implementation. In Fig. 1, we show a smart contract for an imaginary cryptocurrency, we named 'SampleCrypto'. SampleCrypto can only be issued by its developer and can be transferred to a receiver with his/her address.

```
pragma solidity ^0.4.0;

/*  Sample contract for SampleCrypto  */

contract SampleCrypto{
   address public developer;
   mapping (address => uint) public balance;

   //   Notifies when a transaction is complete
   event transaction(address from, address to, uint amount);

   function SampleCrypto()
   {
       developer = msg.sender;
   }

    function create(address receiver, uint amount)
    {
        if(msg.sender != developer)
            throw;
        balance[receiver] += amount;
    }

    function receiver(address receiver, uint amount)
    {
        if(balance[msg.sender] < amount)
            throw;
        balance[msg.sender] -= amount;
        balance[receiver] += amount;
        transaction(msg.sender, receiver, amount);
    }
}
```

Fig. 1. SampleCrypto implementation in Solidity

2.2 Pact

Pact [8] is a programming language for writing smart contracts to be executed by the Kadena [9] blockchain. Pact empowers developers to implement robust, performant transactional logic, executing mission-critical business operations quickly and safely.

Pact is immutable, Turing-incomplete and favors a declarative approach over complex control-flow. This makes bugs harder to write and easier to spot. Pact smart contracts are designed to enforce business rules guarding the update of a system-of-record: complex, speculative application logic simply does not belong in this critical layer.

Sample Contract Implementation. We present an example Pact code [8], implementing a simple "account balance" smart contract, with functions to create accounts and transfer funds, in Fig. 2. A detailed description of the above contract including information on Installing the module, Keyset Definition, Module Definition, Table Creation and finally, Invoking the 'accounts' module can be found at [8].

2.3 Liquidity

Liquidity [10] is a high-level typed smart-contract language for Tezos [11]. It is a fully typed functional language, it uses the syntax of OCaml [12] and strictly complies with Michelson [13] security restrictions. A formal verification framework for it is under development, to prove the correctness of smart-contracts written in Liquidity.

```
(define-keyset 'accounts-admin
  (read-keyset "accounts-admin-keyset"))

(module accounts 'accounts-admin
 "Simple account functionality."

 (defschema account
  "Schema for accounts table."
  balance:decimal
  amount:decimal
  keyset:keyset
  note)

 (deftable accounts:{account})

 (defun create-account (address keyset)
  (insert accounts address
         { "balance": 0.0, "amount": 0.0, "keyset": keyset,
           "note": "Created account" }))

 (defun transfer (src dest amount)
   "transfer AMOUNT from SRC to DEST"
  (with-read accounts src
             { "balance":= src-balance
             , "keyset" := src-ks }
   (enforce-keyset src-ks)
   (check-balance src-balance amount)
   (with-read accounts dest { "balance":= dest-balance }
    (write accounts src
           { "balance": (- src-balance amount)
           , "amount": (- amount)
           , "note": { "transfer-to": dest } })
    (write 'accounts dest
           { "balance": (+ dest-balance amount)
           , "amount": amount
           , "note": { "transfer-from": src } })))))

 (defun check-balance (balance amount)
   (enforce (<= amount balance) "Insufficient funds"))
)

(create-table accounts)
```

Fig. 2. Account balance smart contract in Pact

The Liquidity language provides three key features: (1) full coverage of the Michelson language: anything that can be written in Michelson can be written in Liquidity. (2) local variables instead of stack manipulations: values can be stored in local variables. The only restriction is that local variables do not survive to Contract.call, following the philosophy of Michelson to force explicit storage of values to limit reentrancy bugs. (3) high-level types: types like sum-types and record-types can be defined and used in Liquidity programs. Liquidity's contract format can be found in [10].

Sample Contract Implementation. The following contract [14], shown in Fig. 3, is a simple voting system that requires a user to have at least 5 tz to submit a vote.

The contract will display an error message "Not enough money, at least 5 tz to vote" if the user attempts to vote with a balance lower than 5 tz.

In Table 1, we have summarized the smart contract programming languages discussed in this section. The table lists the major platform that supports or plans to support these languages. We have also listed some of the key features of these languages in the designated column.

```
[%%version 0.15]

let%init storage (myname : string) =
  Map.add myname 0 (Map ["ocaml", 0; "pro", 0])

let%entry main
    (parameter : string)
    (storage : (string, int) map)
  : unit * (string, int) map =

  let amount = Current.amount() in

  if amount < 5.00tz then
    Current.failwith "Not enough money, at least 5tz to vote"
  else
    match Map.find parameter storage with
    | None -> Current.failwith "Bad vote"
    | Some x ->
        let storage = Map.add parameter (x+1) storage in
        ( (), storage )
```

Fig. 3. Smart contract with Liquidity

Table 1. Summary of smart contract programming languages

Programming languages	Major platforms	Key features
Solidity	Ethereum	• Statically typed • Supports inheritance, libraries and complex user-defined types
Pact	Kadena	• Turing-incomplete safety-oriented design • Human-readable, on-ledger code • Atomic execution (transactions) • Module definition and import • Unique "key-row" + columnar database metaphor • Expressive syntax and function definition • Single-signature and multi-signature public-key authorization • Type inference
Liquidity	Tezos	• High-level types: types like sum-types and record-types can be defined and used in Liquidity programs • Full coverage of the Michelson language: Anything that can be written in Michelson can be written in Liquidity, • Local variables instead of stack manipulations

3 Empirical Evaluation

The goal of our empirical study is to determine the usability and security vulnerabilities of the smart contract programming languages discussed in the previous section. Our study was designed around the scenario in which the formal descriptions of three smart contracts were provided to human test subjects to implement using an assigned smart contract programming language while assessing the usability and analyzing the types of

bugs and security vulnerabilities that developers can leave behind in the contracts. We therefore, designed our experimental study based on the following research questions (referred to as RQ's):

RQ1. How do the languages under study stack up in terms of usability to new contract developers?

RQ2. What are the common security issues left behind in the contract by new developers?

3.1 Experimental Planning

Conducting an empirical study involving human subjects can lead to several challenges and pitfalls. Guidelines exist in the literature [15] to help researchers to carry out such type of studies. These guidelines helped us to design our experiment, especially because a frequent problem with controlled empirical studies is that, due to their cost and complexity, they are often limited in size.

We have divided our experimental planning into four parts, namely *test subjects*, *test contracts*, *measures*, and *experimental design*. We discuss the experimental setup in the sub-sections below:

Test Subjects. We sent email invitations to undergraduate students and graduate research assistants in the College of Computing and Software Engineering (CCSE) at Kennesaw State University (KSU), US to participate in our study. The email described the aim and objective of our experiment (which is to perform an empirical evaluation of smart contracts programming language based on usability and security), location, time, expected length of the experiment and an RSVP link. We received a response from a total of 15 undergraduate students and Graduate research assistants within the mentioned deadline. Each subject had prior experience with at least one general purpose programming language and object-oriented concepts.

Test Contracts. We prepared formal descriptions of three test contracts in the form of scenario paragraphs for the test subjects in our experiment. The three contracts were selected after a careful evaluation of several smart contracts from various online sources that are prone to security vulnerabilities when implemented by new developers. The prepared formal descriptions were carefully checked for completeness and ambiguity.

Table 2 gives a brief description of the three test contracts chosen for our experiment. Each test subject had to implement these three contracts in a randomly assigned smart contract programming language, as described in detail in the experimental design.

Measures. To quantify usability, we used a built-in timer (as part of the helper program in the experiment environment) to measure the average times of implementation of each contract in an assigned programming language for each test subject (in minutes). The longer the subjects took to implement a contract, the lower would be the usability of the language to a new developer. To arrive at more solid results, we additionally asked the subjects to answer a questionnaire regarding the usability of the assigned smart contract programming language at the end of the experiment in the exit

Table 2. Summary of the test contracts for our study

Test contract	Description	Reference
Contract 1: HoneyPot	A contract to keep a record of balances for each address that puts currency in it and allow these addresses to get them later	[16]
Contract 2: Bank Account	A contract to deposit/withdraw money into a user's bank account	[17]
Contract 3: King of the currency	A simple contract in which the highest bidder becomes the leader of a group	[18]

questionnaire. We asked the subjects to rate the usability of the language on a scale of 1.0–10.0, 1.0 being extremely difficult to use and 10.0 being extremely usable. We also asked them for their comments on the language, such as what did they find easy? What did they find difficult?

We used a two-facet method to capture and analyze security vulnerabilities in the implemented smart contracts. Firstly, we ran the implemented contracts against six known security vulnerabilities including, *Callstack Depth Attack Vulnerability*, *Reentrancy Vulnerability*, *Assertion Failure*, *Timestamp Dependency*, *Parity Multigeniture Bug 2* and *Transaction-Ordering Dependence (TOD)* with the help of 'OYENTE' [19, 20] tool. OYENTE is an automated security analysis tool for revealing the above-mentioned security vulnerabilities in smart contracts. Secondly, we analyzed the implemented contracts manually to check for further vulnerabilities that were not covered by the tool including DoS (Denial of Service) with (Unexpected) revert and DoS with Block Gas Limit in Solidity [21]. The more these vulnerabilities surface in a contract (from both automated and manual parts), the less secure the underlying contract programming language would be.

Experimental Design. We prepared 15 envelopes each of which contained the formal description of the three smart contracts and the smart contract programming language that these contracts need to be implemented with. Each envelope was also assigned an Envelope ID number which helped us to keep track of the language the contracts need to be implemented in. The envelopes were prepared such that only 5 envelopes would contain the language L_i (where $L_i \in$ {Solidity, Pact, Liquidity}). Hence, out of our 15 test subjects, only 5 random chosen subjects would implement the test contracts in a language L_i. We made sure that we assigned test subjects computers such that no two subjects who had to implement the contracts in the same language sit alongside each other. Each language was to be implemented on an online compiler, i.e. we used Remix [22] for Solidity, Try-Pact [23] editor/compiler for Pact, and Liquidity online editor/compiler [14] for Liquidity.

Table 3 summarizes the organization of our experiment. As shown in the table, there were 5 test subjects who worked on implementing the given three smart contracts in the assigned smart contract programming language.

Table 3. Organization and assignment of envelopes and languages in the experiment

Subject ID	Envelope ID	Language
01, 04, 07, 10 and 13	03, 06, 09, 12 and 15 respectively	Solidity
02, 05, 08, 11 and 14	01, 04, 07, 10 and 13 respectively	Pact
03, 06, 09, 12 and 15	02, 05, 08, 11 and 14 respectively	Liquidity

3.2 Experimental Execution

To avoid fatigue, we decided to conduct our experiment in two sessions. The first session was a background and demo session. During this session, each subject received a starter pack consisting of their subject ID, a statement of consent, a background questionnaire, instructions regarding the experiment, a printout of the demo slides and an exit questionnaire. Before commencing with the demo, each subject was required to fill in the questionnaire based on their background and programming experience and sign a statement of consent to participate in our experimental study.

After the statement of consent and background questionnaire were signed, completed and collected, we proceeded with a small presentation on the basics of smart contracts programming to familiarize the subjects with the same. Next, we conducted a Q&A session with the subjects to answer any of their questions and concerns. When all the questions were answered and any confusions cleared, we asked the test subjects to take a 45-min break to refresh themselves. We also asked them to keep their starter packs with them in case they needed to review the slides during the break.

After the break, we commenced the second session of our experiment. This session was for the practical implementation of test contracts. Each test subject was handed a sealed envelope with an envelope ID number on it (this helped us to keep track of the smart contract language that the envelope's contracts need to be implemented in). Each envelope had a formal description of the three test contracts and the language in which the subjects were assigned to implement these contracts. After all the envelopes were handed, we matched the subject ID's with envelope ID's to keep a track on our experiment. We then asked the subjects to open their envelopes and read all the problem statements thoroughly, we then conducted a second Q&A session to remove all doubts and confusions regarding the problem statements and the programming languages that they were assigned. When this was over, we asked them to implement a simple warm-up "Hello world" exercise in the language they were assigned. Finally, when all the subjects were done with the warm-up exercise, we asked them to begin working on their problems and started a timer for each subject.

The subjects were given two hours to implement all the contracts, and we asked them to remain seated even if they finished their task before the time limit. To be considered "finished", we required them to be certain that their test contracts compiled successfully on the online compilers mentioned earlier.

Including presentation and break, the duration of the experiment was three hours and forty-five minutes. The experiment was completed under "exam conditions", i.e., subjects were not allowed to communicate with others, or consult with other sources to avoid introducing biases into the experimental findings. Finally, after the end of the experiment, each subject was asked to fill in the exit questionnaire before leaving.

3.3 Experimental Results and Analysis

Following the experimental execution process described in the preceding section, we collected the required experimental data and carefully analyzed all the data collected to arrive at conclusions. We now present these results in response to our research questions.

RQ1: How do the languages under study stack up in terms of usability to new contract developers? The data collected for measuring the usability of smart contracts programming languages is shown in Tables 4 and 5. Table 4 summarizes the average implementation times of each test smart contract with a given smart contract programming language (i.e. Solidity, Pact and Liquidity). We found that the average implementation time of each contract was significantly lower in case of Solidity as compared to Pact and Liquidity.

Table 4. Summary of the implementation times of the test contracts

Language	Average implementation time (Contract 1)	Average implementation time (Contract 2)	Average implementation time (Contract 3)	Total average implementation time
Solidity	13 min 26 s	20 min 07 s	19 min 14 s	52 min 47 s
Pact	17 min 31 s	31 min 16 s	33 min 32 s	82 min 19 s
Liquidity	14 min 21 s	23 min 21 s	27 min 51 s	65 min 33 s

We observed the data extracted from the experiment to be consistent across all three test contracts and languages for all test subjects, i.e. each test subject who implemented the three test contracts with Solidity did so faster than every test subject who implemented the contracts with Pact or Liquidity. Similar observation was made in case of Liquidity and Pact, i.e. each subject who implemented test contracts with Liquidity did so faster than every subject who implemented the contracts with Pact.

In Fig. 4, we represent the average implementation times of each test contract (shown in Table 4) with each smart contract programming language in our experiment. We made an interesting observation for Test Contract 1, i.e. the average implementation times of test contract 1 with Solidity and Liquidity were almost similar, i.e. $\Delta t_{LS1} < 1$ min (where Δt_{LS1} = Average implementation time of test contract 1 with Liquidity - Average implementation time of test contract 1 with Solidity = 55 s). But, this time difference increased significantly (i.e. Δt_{LS2} = 3 min 14 s and Δt_{LS3} = 8 min 37 s) with the increased complexity of test contracts 2 and 3 as compared to test contract 1. We made another anomalous observation for implementation times of test contract 2 and 3, i.e. In case of Solidity, the average implementation time of test contract 3 is lower than average implementation time of test contract 2. On the other hand, this is not the case for Pact and Liquidity, as the average implementation time of test contract 3 is higher with these languages when compared to the average implementation time of test contract 2. But, since, the average implementation time of test contract 2 and 3 is lower with Solidity when compared to average implementation times with Pact and Liquidity, this observation has negligible value.

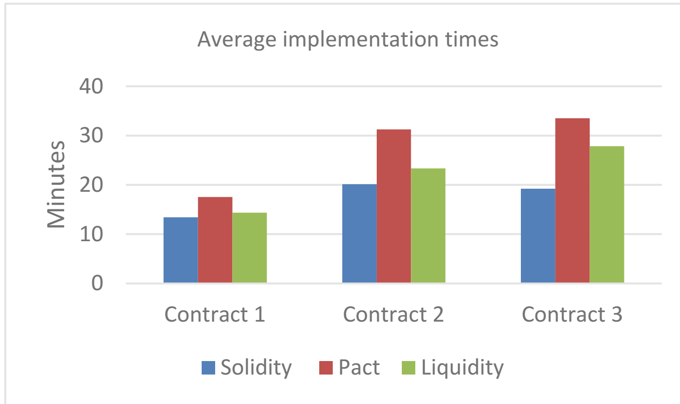

Fig. 4. Average implementation times of all contracts with Solidity, Pact and Liquidity

Additionally, Fig. 5 represents the total average implementation times of all three test contracts implemented with each smart contract programming language in our experiment, i.e. Solidity, Pact and Liquidity. The figure shows that the total average implementation time of all test contract with Solidity is 52 min and 47 s. The total implementation time increased by 24.2% to 65 min and 33 s with Liquidity and almost by 56% to 82 min and 19 s with Pact as compared to total average implementation time with Solidity.

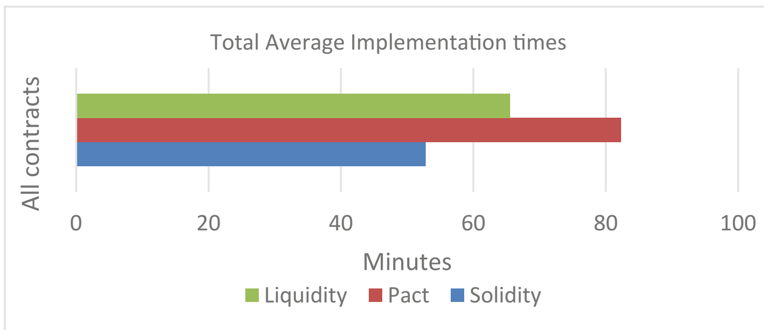

Fig. 5. Total average implementation times of all contracts with Solidity, Pact and Liquidity

Finally, Table 5 summarizes the results from the exit questionnaire which was completed by the test subjects at the end of the experiment. The table represents the average usability score as graded by the test subjects to their respective assigned language for the experiment. The higher the average usability score (see Sect. 3.1 - Measures) the more usable the language is to a new developer.

Table 5. Average usability score from the exit questionnaire of the test subjects

Language	Average usability score by test subjects
Solidity	8
Pact	4.5
Liquidity	5.5

Summarizing the overall usability results of our experiment, we see that the average implementation times of each contract are such that - $t_{SC1} < t_{LC1} < t_{PC1}$ for test contract 1, $t_{SC2} < t_{LC2} < t_{PC2}$ for test contract 2 and $t_{SC3} < t_{LC3} < t_{PC3}$ for test contract 3 respectively (where, t_{SCi}, t_{LCi} and t_{PCi} are the average implementation times of test contract 'i' with Solidity, Liquidity and Pact respectively and $i \in \{1,2,3\}$). This leads to the subsequent result regarding the total implementation times of all 3 test contracts, i.e. $T_S < T_L < T_P$ (where, T_S, T_L and T_P are the total average implementation times of all 3 test contracts). Additionally, from the results of the exit questionnaire in Table 5 we see – $U_S > U_L > U_P$ (where, U_S, U_L and U_P are the average usability scores of Solidity, Liquidity and Pact respectively). For a language to have higher usability, we require it to have faster implementation times and high usability scores in the exit questionnaire. Hence, it is clear from the results presented in this section that the usability of Solidity > Liquidity > Pact for a new developer.

RQ2: What are the common security issues left behind in the contract by new developers? We analyzed the implemented test contracts for security vulnerabilities using the 'OYENTE' tool and methods described in Sect. 3.1. While we couldn't find any security vulnerabilities for Liquidity and Pact implemented contracts similar could not be said for Solidity implemented contracts. The results of our security analysis are summarized in Table 6.

Table 6. Security issues found in the implemented test contracts

Contract	Solidity	Pact	Liquidity
Contract 1	Reentrancy vulnerability - (5/5 contracts)	None	None
Contract 2	Reentrancy vulnerability - (1/5 contracts)	None	None
Contract 3	DoS with (Unexpected) revert - (5/5 contracts)	None	None

For test contract 1, we found that all implemented contracts by the test subjects were vulnerable to Reentrancy attacks. Similar results were found for test contract 3, where all the implemented contract were prone to DoS with (unexpected) revert vulnerability [21]. Meanwhile, for test contract 2, only one of the implemented contract was vulnerable to Reentrancy vulnerability.

A Reentrancy attack occurs when a function 'x' calls a function 'y' in an external contract, which makes a reentrant call to 'x'. If x's call to 'y' occurs while the contract is in an inconsistent state, then the reentrant call may make invalid assumptions about the initial state of the contract. This reentrancy vulnerability was recently exploited to steal over $40 million [24]. This problem is difficult and error-prone to avoid in

Solidity, this is because one shall reason about reentrant call anytime an external call is made from a function. Since, sending money always results in an external call when using Solidity, this becomes a frequent vulnerability. In general, external calls from function 'x' in contract C may invoke additional calls on C to any function, not just 'x', via an intermediate external invocation. This is more problematic than internal-only calls because the external contract is likely to assume C is in a consistent state.

Figure 6 visually represents the security vulnerabilities in all the implemented contracts with the Solidity, Pact and Liquidity languages. No security vulnerabilities were found in the Pact and Liquidity implemented contracts but 11 (approx. 73%) Solidity contracts were found to be vulnerable out of a total of 15 implemented. Out of these 11 vulnerable contracts, 6 had Reentrancy vulnerabilities and 5 were vulnerable to DoS with (unexpected) revert. Only 4 out of 15, i.e. about 27% of Solidity implemented contracts were found to be secure.

Fig. 6. Security vulnerabilities in all implemented contracts

While using Solidity it is often difficult for even experienced developers to avoid certain pitfalls [5], it is no surprise that the implemented Solidity contracts were prone to security vulnerabilities as the subjects for our experiment can be considered as new and inexperienced contract developers. Hence, from the results of our experiment and in response to our second research question: RQ2, we found that contracts implemented by new developers with Solidity are more prone to security vulnerabilities as compared to smart contracts implemented with Liquidity and Pact. Even though, no vulnerabilities were found in Pact and Liquidity implemented contracts, this by no means necessarily suggests that Pact and Liquidity contracts are 100% immune to security vulnerabilities. It just implies that it is harder for new developers to leave behind bugs and security vulnerabilities when working with Pact and Liquidity as compared to Solidity.

3.4 Threats to Validity

This section discusses the threats to validity for this experiment. The threats to external validity primarily answer the question of how representative the human subjects, the

test contracts, and used tools are. Some of our subjects were essentially students and did not have professional smart contract development experience. However, analysis of the results indicated that subjects had similar programming experience and managed to implement their contracts in similar times as others in their group. The test contracts used in our study were not developed by the subjects and may have been unfamiliar to them. To mitigate this, we had organized Q&A and warm-up sessions. We made sure to provide sufficient time to our test subjects for the experiment and this was confirmed in the exit questionnaire, where we asked the subjects if they felt they had been given enough time for the experiment. All test subjects stated to have had enough time to complete the experiment. Additionally, the test contracts selected for the experiment can be considered relatively simple for experienced developers. Hence, it is possible more complex contracts may yield different results. As no previous human studies have been done in this area, we believe beginning with reasonable-scale studies and the lessons learned is prudent to pave the way for larger studies.

The threats to internal validity are implementation effects that could have possibly biased our test results. We designed the formal descriptions of the chosen test contracts carefully for intuition. To check the completeness of our description we conducted pilot research where we provided the prepared formal description to experienced developers and professors for constructive criticism, fault detection and completeness checking. Only when all errors were rectified in our formal descriptions, we decided to go ahead and commence our experiment.

3.5 Discussion

Based on the presented results and comparisons from our experiment, we found that Solidity is the most usable language to a new developer when it comes to programming smart contracts. We found from our exit questionnaire, that this was because of Solidity's intimacy to general purpose programming languages such as Java or C#, which are often used by developers and students in professional and academic environments respectively.

But unfortunately, when it comes to security vulnerabilities in smart contract implemented by new developers, Solidity lacks behind as we found it to be most prone to security vulnerabilities. Although being usable is a huge plus, on the other hand being prone to security vulnerabilities is a huge downside as these security vulnerabilities can be exploited by malicious users to cause financial damages as seen from the recent attacks on the Ethereum platform [25]. Meanwhile, Liquidity and Pact are still new languages which lack high usability at this time but seem more secure than Solidity for now. Ultimately, all the three languages have their pros and cons. The exciting thing to note here is all the three languages are changing and evolving with time and research. Hence, our work aims to help contribute towards forming the body of knowledge for the continuous growth and evolution of this infant domain.

4 Related Work

Our work is related to assessing and comparing the programming languages for smart contract development. There are in fact very few numbers of related studies known for evaluation of smart contract programming languages or platforms. In this section, we present an overview of all the related work, which has been proposed in the literature in recent years.

The work in [26], compares Ethereum, IBM Open Blockchain (Hyperledger project) [27], Intel Sawtooth lake [28], BlockStream Sidechain Elements [29] and Eris [30] platforms. This study suggests Ethereum to be the primary choice of platform in terms of scalability, development, documentation and support. As Intel Sawtooth Lake wasn't fully implemented at the time, the authors conclude that Ethereum is the better solution for developers as it had no security issues known at the time.

In another work [31], the authors analyze the usage of smart contracts platforms from various perspectives. The study examines a sample of 6 platforms, namely Bitcoin [32], Ethereum [33], Counterparty [34], Stellar [35], Monax [36] and Lisk [37] for smart contracts by highlighting some of the key differences in terms of type of blockchain, contract language and volume of daily currency transfers. A sample of 834 contracts was studied for the 2 platforms — Bitcoin and Ethereum, categorizing each of them by application domain, and measuring the relevance of each of these categories. They concluded that about 80% of the Ethereum contracts use at least one of the nine design patterns presented in the paper.

There have also been assessment kind of works that study different blockchain technologies. Anderson et al. [38] compare three blockchains - Ethereum, Namecoin, and Peercoin. For Ethereum, the authors briefly analyze the issues that are introduced by the negligent design of smart contracts. In the case of Namecoin, the focus was, how the name registration is used and had developed over time. For Peercoin, the interest was in the use of proof-of-stake. Similarly, Seijas et al. [39] compare a variety of smart contract platforms. Their work also provides an overview of the scripting languages used in cryptocurrencies, particularly scripting languages of Bitcoin, Nxt and Ethereum. Their work covers technologies that might be used to underpin extensions and innovations in scripting and contracts, including technologies for verification (e.g., zero-knowledge proofs, proof-carrying code and static analysis), as well as approaches to making systems more efficient, e.g. Merkelized Abstract Syntax Trees.

Studies that analyze the security of Ethereum smart contracts have been growing recently. For instance, the work proposed in [25] surveys vulnerabilities and attacks on the Ethereum contracts, while, works [20, 40] propose analysis techniques to detect these vulnerabilities.

Majority of the work listed in this section mainly compares and analyzes the various smart contract platforms. There has been a lack of empirical data on how domain-specific smart contract programming languages might work in tandem with developers and their comparative quality measures such as usability and security. Hence, our work took the first step by providing an experimental analysis of three current domain-specific programming languages. We hope that our proposed work can

be useful in the process of building a body of empirical knowledge helping developers and organizations write safer and more secure smart contracts.

5 Conclusion and Future Work

Research on smart contract programming's evaluation is quite young and there is still a long road ahead to reach its maturity. This paper realizes an evaluation of the current languages as a fundamental step towards reaching this maturity and obtaining useful advances. The given evaluation included an experiment that was performed to compare the usability and security vulnerability of the three domain-specific languages, namely Solidity, Pact and Liquidity. The experiment results demonstrated that although Solidity is the most usable language for a new developer to program smart contracts, it is the least secure language to vulnerabilities. On the other hand, Liquidity and Pact show lower usability but seem secure for now. Consequently, our results contribute to the body of experimental evidence about the usability and security of the smart contract programming languages, which is currently scarce.

In future, we intend to conduct more experiments in order to improve the generalizability of our results in this paper. There are several points that can be suggested to be reinforced towards obtaining more solid conclusions for the comparison of smart contract programming languages. These might include the conduction of experiments with (i) extra object programs (test contracts) that will comprise larger systems with varied context parameters such as application domain or size; (ii) more diversified-background of human subjects; (iii) new upcoming smart contract programming languages that are being developed.

References

1. Peck, M.E.: Blockchains: how they work and why they'll change the world. IEEE spectrum (2017)
2. Cuccuru, P.: Beyond bitcoin: An early overview on smart contracts. Int. J. Law Inf. Technol. **25**, 179–195 (2017)
3. Solidity. https://solidity.readthedocs.io/en/develop/
4. Sirer, E.G.: Thoughts on The DAO Hack. http://hackingdistributed.com/2016/06/17/thoughts-on-the-dao-hack/
5. Hern, A.: "$300 M in Cryptocurrency" Accidentally Lost Forever Due To Bug. https://www.theguardian.com/technology/2017/nov/08/cryptocurrency-300m-dollars-stolen-bug-ether
6. Ethereum Project. https://www.ethereum.org/
7. Design Rationale. https://github.com/ethereum/wiki/wiki/Design-Rationale
8. Popejoy, S.: The pact smart-contract language (v1.5), pp. 1–15 (2017)
9. Kadena. http://kadena.io/#/
10. Liquidity, a simple language over Michelson. https://github.com/OCamlPro/liquidity/blob/master/docs/liquidity.md
11. Tezos. https://www.tezos.com/
12. OCaml Documentation. https://ocaml.org/docs/

13. Ii, S.: Michelson : the language of Smart Contracts in I - Semantics
14. Liquidity Online Editor. http://www.liquidity-lang.org/edit/
15. Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., El Emam, K., Rosenberg, J.: Preliminary guidelines for empirical research in software engineering. IEEE Trans. Softw. Eng. **28**, 721–734 (2002)
16. Guimaraes, G.: Reentrancy attack on smart contracts: how to identify the exploitable and an example of an attack contract. https://medium.com/@gus_tavo_guim/reentrancy-attack-on-smart-contracts-how-to-identify-the-exploitable-and-an-example-of-an-attack-4470a2d8dfe4
17. Martinsson, F.: Smart contract programming on Ethereum - solidity beginners tutorial part 2. https://www.youtube.com/watch?v=F4XQFEievJI
18. Konstantopoulos, G.: How to secure your smart contracts: 6 solidity vulnerabilities and how to avoid them (Part 2). https://medium.com/loom-network/how-to-secure-your-smart-contracts-6-solidity-vulnerabilities-and-how-to-avoid-them-part-2-730db0aa4834
19. Oyente. https://oyente.melon.fund/#version=soljson-v0.4.21+commit.dfe3193c.js
20. Luu, L., Chu, D.-H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of 2016 ACM SIGSAC Conference on Computer and Communications Security – CCS 2016, pp. 254–269 (2016)
21. Smart Contracts - Best practices (Known attacks). https://github.com/ConsenSys/smart-contract-best-practices/blob/master/docs/known_attacks.md
22. Remix. http://remix.ethereum.org/#optimize=false&version=soljson-v0.4.21+commit.dfe3193c.js
23. Try Pact. http://kadena.io/try-pact/
24. Omohundro, S.: Cryptocurrencies, smart contracts, and artificial intelligence. AI Matters **1**, 19–21 (2014)
25. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on Ethereum smart contracts (SoK), pp. 1–24 (2017)
26. Macdonald, M., Liu-Thorrold, L., Julien, R.: The blockchain: a comparison of platforms and their uses beyond bitcoin. Work. Pap., pp. 1–18 (2017)
27. Hyperledger. https://www.hyperledger.org/
28. Intel: Intel: Sawtooth Lake. https://intelledger.github.io/
29. BlockStream Sidechain Elements. https://blockstream.com/technology/
30. Documentation for Eris. https://abal.moe/Eris/docs
31. Bartoletti, M., Pompianu, L.: An empirical analysis of smart contracts: platforms, applications, and design patterns (2017)
32. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system, p. 9 (2008). www.Bitcoin.Org
33. Buterin, V.: A next-generation smart contract and decentralized application platform. http://buyxpr.com/build/pdfs/EthereumWhitePaper.pdf
34. Counterparty: Protocol Specification. https://counterparty.io/docs/protocol_specification/
35. Stellar. https://www.stellar.org/
36. Monax. https://monax.io/
37. Lisk. https://lisk.io/
38. Anderson, L., Holz, R., Ponomarev, A., Rimba, P., Weber, I.: New kids on the block: an analysis of modern blockchains (2016)
39. Seijas, P.L., Thompson, S., McAdams, D.: Scripting smart contracts for distributed ledger technology. Cryptology ePrint Archive, Report 2016/1156 (2016). http://eprint.iacr.org/2016/1156
40. Bhargavan, K., Swamy, N., Zanella-Béguelin, S., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T.: Formal verification of smart contracts. In: Proceedings of 2016 ACM Workshop on Programming Languages and Analysis for Security – PLAS 2016, pp. 91–96 (2016)