Protocol-Based Smart Contract Generation

Afonso Falcão[1], Andreia Mordido[1], and Vasco T. Vasconcelos[1]

LASIGE, Faculty of Sciences, University of Lisbon, Portugal
anfalcao@lasige.di.fc.ul.pt afmordido@ciencias.ulisboa.pt
vmvasconcelos@ciencias.ulisboa.pt

Abstract. The popularity of smart contracts is on the rise, yet breaches in reliability and security linger. Among the many facets of smart contract reliability, we concentrate on faults rooted in out-of-order interactions with contract endpoints. We propose SmartScribble, a protocol language to describe valid patterns of interaction between users and endpoints. SmartScribble not only ensures correct interactive behaviour but also simplifies smart contract coding. From a protocol description, our compiler generates a smart contract that can then be completed by the programmer with the relevant business logic. The generated contracts rely on finite state machines to control endpoint invocations. As a proof of concept, we target Plutus, the contract programming language for the Cardano blockchain. Preliminary evaluation points to a 75% decrease in the size of the code that developers must write, coupled with an increase of reliability by enforcing the specified patterns of interaction.

Keywords: Programming language · Smart contract · Protocol specification · State machine

1 Introduction

Smart contracts are a focal point of modern blockchain environments. Such contracts were firstly popularized by Ethereum [19], but soon thereafter other networks developed their own smart contract languages, enabling the implementation of blockchain-based decentralized applications between untrusted parties.

Smart contracts usually operate over user owned assets and, thus, vulnerabilities in programs and in the underlying programming languages can lead to considerable losses. The famous attack on the DAO resulted in a theft of approximately 60 million USD worth of Ether [1,5,7]. Due to recent exploitations of vulnerabilities in smart contracts, blockchain providers turned their attention to the development of robust programming languages, often relying on formal verification, including Liquidity by Tezos [15], Plutus by IOHK [4], Move by Facebook [3], and Rholang by RChain [17]. Such languages aim at offering flexible and complex smart contracts while assuring that developers may fully trust contract behaviour. Unfortunately, for Plutus, the last objective has not been completely achieved yet. As we show in the next section, in Plutus, assets can be easily lost forever to the ledger with a simple unintended interaction.

To counter unplanned interactions with smart contracts endpoints while automating the development of boilerplate code, we propose SmartScribble, a protocol specification language for smart contracts. Its syntax is adapted from the Scribble protocol language [20] to the smart contract trait and features primitives for sequential composition, choice, recursion, and interrupts. Protocols in SmartScribble specify interactions between participants and the ledger, as well as triggers to interrupt protocol execution. The business logic underlying the contract can be added by the programmer after the automatic generation of the smart contract boilerplate code. The generated code relies on a finite state machine to validate all interactions, precluding unexpected behaviours.

SmartScribble currently targets Plutus, a native smart contract programming language for the Cardano blockchain [14], based on the Extended Unspent Transaction Output model [6], a solution that expands the limited expressiveness of the Unspent Transaction Output model. In UTxO, transactions consist of a list of inputs and outputs. Outputs correspond to the quantity available to be spent by inputs of subsequent transactions. Extended UTxO expands UTxO's expressiveness without switching to an account-based model, that introduces a notion of shared mutable state, ravelling contract semantics [6]. Nevertheless, the framework we propose can be integrated with other smart contract languages and blockchain infrastructures expressive enough to support state machines.

Several works have been adopting state machines to control the interaction of participants with smart contracts. FSolidM [16]—the closest proposal to SmartScribble—introduces a model for smart contracts based on finite state machines. FSolidM relies on the explicit construction of finite state machines for contract specification; instead, we automatically generate all state machine code. On a different fashion, the model checker Cubicle [9] encodes smart contracts and the transactional model of the blockchain as a state machine.

SmartScribble distinguishes itself from other domain-specific languages—BitML, integrated with the Bitcoin blockchain [2], Obsidian [8], a typestate-oriented language, and Nomos [10], a functional (session-typed) language—by abstracting the interactive behaviour and details of the target programming language through a protocol specification, only relying on the smart contract language to implement the business logic and thus flattening the learning curve.

The rest of this paper is organised as follows. Section 2 motivates SmartScribble via an example where assets are lost to the ledger; section 3 presents the protocol language and section 4 focuses on contract generation from SmartScribble protocols. Section 5 summarizes some preliminary results of our evaluation of SmartScribble, and section 6 concludes the paper and points to future work. Appendix A contains the source code for the vulnerable contract we explore in section 2, appendix B presents input and logs for some simulations, appendix C provides the source code for the business logic of our running example and appendix D provides a detailed analysis of the preliminary evaluation results. An implementation is available [11].

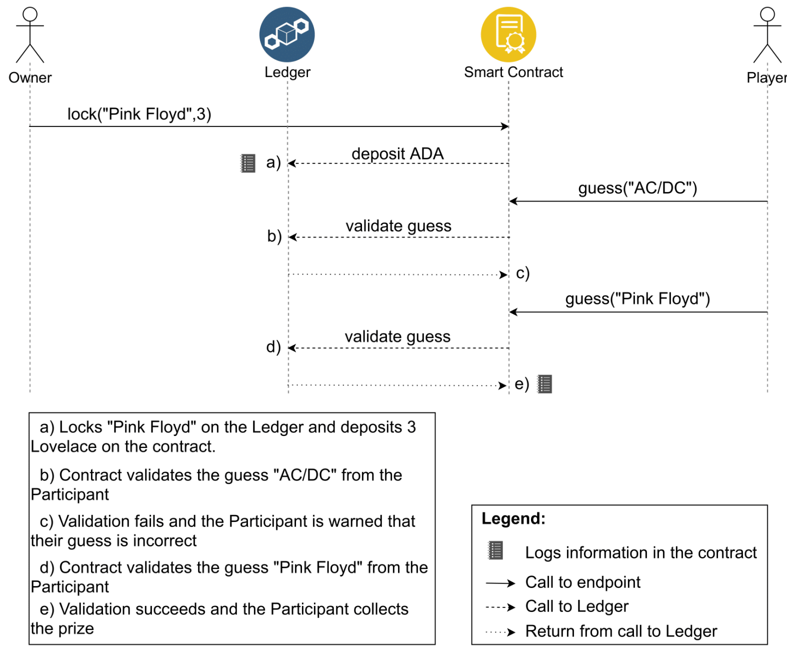

Owner
Ledger
Smart Contract
Player
lock("Pink Floyd",3)
deposit ADA
a)
guess("AC/DC")
validate guess
b)
c)
guess("Pink Floyd")
validate guess
d)
e)
a) Locks "Pink Floyd" on the Ledger and deposits 3 Lovelace on the contract.
b) Contract validates the guess "AC/DC" from the Participant
c) Validation fails and the Participant is warned that their guess is incorrect
d) Contract validates the guess "Pink Floyd" from the Participant
e) Validation succeeds and the Participant collects the prize
Legend:
Logs information in the contract
Call to endpoint
Call to Ledger
Return from call to Ledger

Fig. 1: Diagram of a particular well-behaved flow of operations

2 Smart Contracts Can Go Wrong

This section identifies a weakness of the Plutus smart contract programming language. Although Plutus is developed with a clear focus on reliability, it lacks mechanisms to enforce correct patterns of interactions. As a running example we consider the popular guessing game [4,12], the paradigm for secret-based contracts where participants try to guess a secret and get rewarded if successful; the contract can be found in appendix A. (An example that falls in this category is a lottery.) Figure 1 represents a correct sequence of events in the guessing game:

1. The *owner* of the contract locks a secret and deposits a prize in ADA[1] to be retrieved by the first player who correctly guesses the secret.
2. The *player* tries to guess the secret.
3. If the guess matches the secret, the player retrieves the prize and the game ends; otherwise, the player is warned that the guess did not succeed and the game continues.

In Plutus, the parties involved in the protocol are not required to follow valid patterns of interaction. We explore a scenario where one of the parties deviates from the (implicitly) expected flow and show that this leads to a faulty behaviour

[1] ADA is the digital currency of the Cardano blockchain. 1 ADA = 1,000,000 Lovelace.

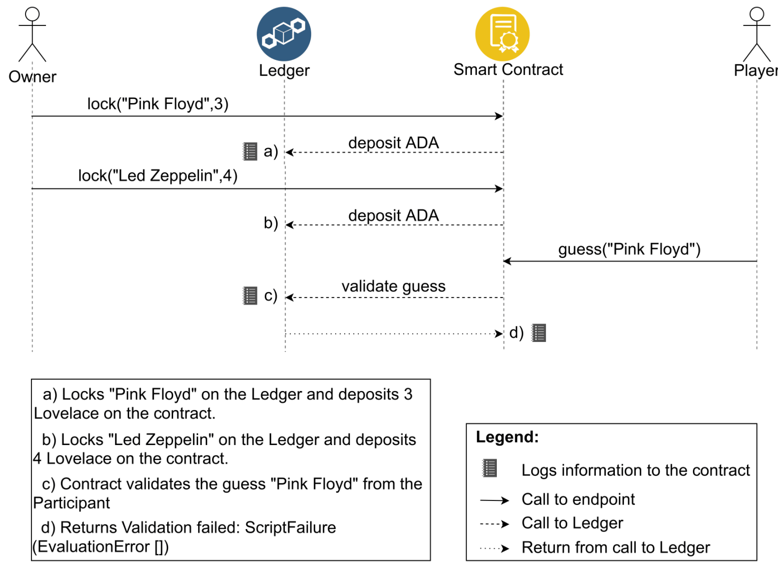
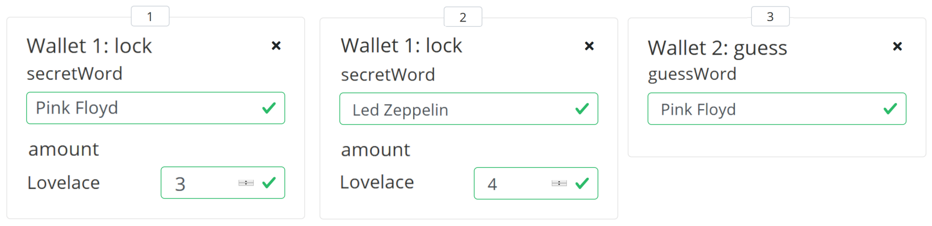

Owner
Ledger
Smart Contract
Player
lock("Pink Floyd",3)
deposit ADA
a)
lock("Led Zeppelin",4)
deposit ADA
b)
guess("Pink Floyd")
validate guess
c)
d)
a) Locks "Pink Floyd" on the Ledger and deposits 3 Lovelace on the contract.
b) Locks "Led Zeppelin" on the Ledger and deposits 4 Lovelace on the contract.
c) Contract validates the guess "Pink Floyd" from the Participant
d) Returns Validation failed: ScriptFailure (EvaluationError [])
Legend:
Logs information to the contract
Call to endpoint
Call to Ledger
Return from call to Ledger

Fig. 2: The owner (incorrectly) locks twice and a correct guess results in a failure


1
Wallet 1: lock
secretWord
Pink Floyd
amount
Lovelace 3

2
Wallet 1: lock
secretWord
Led Zeppelin
amount
Lovelace 4

3
Wallet 2: guess
guessWord
Pink Floyd

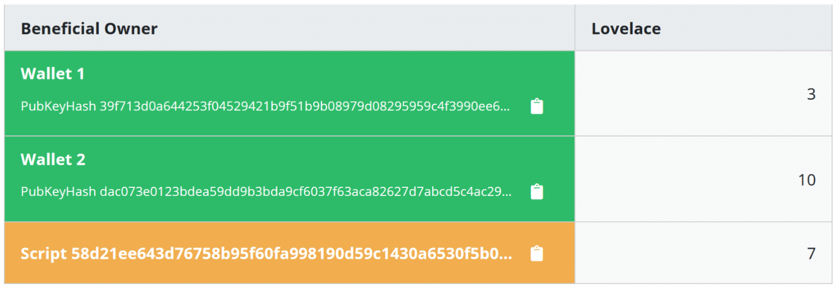
Fig. 3: Playground setup for figure 2: the owner makes two consecutive locks, the player guesses the first secret (complete version in figure 9, appendix B)

that is silenced by the blockchain. Figure 2 represents a scenario where the owner incorrectly executes two consecutive lock operations and the player provides a correct guess. A simulation of this scenario in the Plutus Playground [12] is illustrated in figure 3. Starting with 10 Lovelace in both the owner's and player's wallets, we reach a situation where the log identifies a validation failure for a guess that coincides with that of the first lock (figure 4). The final balances in figure 5 show that the player did not collect the reward despite having guessed the first secret and the owner lost their money to the contract.

This behaviour is certainly unexpected: rather than overriding the first lock or having the second lock fail, both secrets are stored in the ledger as outputs. When a guess is performed, both stored outputs are compared to the guess and, as a consequence, no guess will ever validate against two different secrets. Furthermore, the prize is irrevocably lost to the contract without any possibility of retrieval. This is an unexpected band of silent behaviour that we want to

```
 === Slot 1 ===
Wallet 1:
ReceiveEndpointCall ("tag":"lock","value":3,"secretWord":"Pink Floyd")
Validating transaction: [..]
=== Slot 2 ===
=== Slot 3 ===
Wallet 1:
ReceiveEndpointCall ("tag":"lock","value":4,"secretWord":"Led Zeppelin")
Validating transaction: [..]
=== Slot 4 ===
=== Slot 5 ===
Wallet 2:
ReceiveEndpointCall ("tag":"guess","guessWord":"Pink Floyd")
Validation failed: [..] (ScriptFailure (EvaluationError []))
```

Fig. 4: Log resulting from the setup in figure 3 with a *vanilla* Plutus smart contract; observe that both locks are validated (`Slot 1` and `Slot 3`) and that a correct guess later results in a validation failure (`Slot 5`)



Beneficial Owner	Lovelace
Wallet 1 PubKeyHash 39f713d0a644253f04529421b9f51b9b08979d08295959c4f3990ee6...	3
Wallet 2 PubKeyHash dac073e0123bdea59dd9b3bda9cf6037f63aca82627d7abcd5c4ac29...	10
Script 58d21ee643d76758b95f60fa998190d59c1430a6530f5b0...	7

Fig. 5: Final balances for input in figure 3 (Plutus Playground)

prevent. Even in this simple scenario users lose assets to the ledger. Similar situations are very likely to occur in complex contracts, with devastating results.

We propose specifying the interaction behaviour of smart contracts through protocols that describe the valid patterns of interactions between different classes of users and the contract. Our approach prevents unexpected contract behaviours by having contracts automatically validating interactions. The protocol for the guessing game described at the end of next section, detects and avoids further attempts to lock secrets, among other unintended interactions. Note that this type of vulnerability is different from Transaction Ordering Dependence [18] that is related to corrupt miners maliciously changing the order of transactions, and not the order in which the endpoints are called.

3 Specifying Protocols in SmartScribble

Scribble [20] is a language to describe application-level protocols for communicating systems. It comes with tools to generate Java or Python APIs on which developers can base correct-by-construction implementations of protocols.

SmartScribble is based as much as possible on Scribble, even if it covers only a fragment of the language and includes support for smart contract specific features. The base types of SmartScribble include **String**, **HashedString** (strings stored in the ledger), **PubKeyHash** (wallet public key identifiers) and **Value** (an amount in ADA). The protocol constructors are as follows.

Interaction: <EndpointSignature> **from** <Role> {<Trigger>∗} describes an interaction between a user playing role Role and the smart contract, by calling a specific endpoint. Endpoint signatures comprise the endpoint name followed by the types of the parameters, e.g., lock (**String**, **Value**).

Choice: **choice** at <Role> {Label1: {...} Label2: {...}} denotes a choice made by Role and featuring different alternative branches.

Recursion: **rec** Label {... Label; } is used to express recurring protocols.

Global escape: **do** {...} **interrupt** {...} denotes an interaction that can be interrupted by a trigger. The first statement inside the **interrupt** block must be one of the declared triggers; it can be succeeded by any protocol constructor.

In addition, the following declarations can be used in protocols.

Triggers come in two sorts:
- **funds trigger** <TriggerName> introduces a trigger activated based on the amount of funds in the contract.
- **slot trigger** <TriggerName> introduces a time-related trigger (slot is the measure of time in a blockchain).

State: **field** <Type>∗; introduces the fields to be stored in states.

To watch SmartScribble in action we start with a very simple version of the guessing game protocol and gradually make it more robust. Our first version is the straight line guessing game, featuring a sequential composition of three endpoints: lock, guess, closeGame.

```
protocol StraightLineGuessingGame (role Owner, role Player) {
  field HashedString; // save the secret in the contract
  // the owner locks the secret and deposits a prize
  lock (String, Value) from Owner;
  guess (String) from Player;  // the player makes a guess
  // the owner closes the game (no further guesses allowed)
  closeGame () from Owner;
}
```

The StraightLineGuessingGame protocol introduces, in the first line, the roles users are expected to take when interacting with the smart contract: the Owner owns the game; the Player makes guesses. Stateful protocols require state to be kept in the contract. The **field** declaration introduces the types of the fields that are stored within the state machine. In this case, we need a **HashedString** for the secret. Along with the declared fields, SmartScribble creates an extra **Value** field by default, this field is used to manage the funds in the contract, in this instance, we use it to store the prize. The fields are stored in the state machine in the form of tuples, with each element of the tuple corresponding to one of the declared fields. Users may declare repeated types. The tuple with stored contents can be used by the programmer when implementing the business logic. Protocol StraightLineGuessingGame makes use of interaction constructs to describe interactions between a user and the endpoints lock, guess and closeGame. In this protocol, the three endpoints must be exercised once, in the order by which they appear in the protocol, and by users of the appropriate role.

It should be stressed that the guessing nature of the contract is nowhere present in the protocol. Nothing in the protocol associates the **HashedString** in endpoint lock to a secret, or **Value** to the prize. Nowhere it is said that guessing the secret entails the transfer of the prize to the Player's account. Instead, the protocol governs interaction only: which endpoints are available to which roles, at which time. The business logic associated with the contract is programmed later, in the contract language of the blockchain.

Our next version allows the owner to cancel the game after locking the secret (perhaps the secret was too easy or the prize was set too low). The **choice** operator denotes a choice made by a user, featuring different alternative branches.

```
protocol ChoiceGuessingGame (role Owner, role Player) {
  field HashedString;
  lock (String, Value) from Owner;
  choice at Owner {
    proceedWithGame: { // owner wants players to guess
      guess (String) from Player;
    }
    cancelGame: { // the owner chooses to cancel the game
    }
  }
  closeGame () from Owner;
}
```

After locking the secret, the Owner is given two choices: to cancel the game (cancelGame) or to allow a player to make a guess (proceedWithGame). The two branches represent two different endpoints in the contract. The **choice** is in the hands of a single role, Owner in this case. This role should exercise one endpoint or the other (but not both). Protocols for the two branches are distinct. In the case of proceedWithGame, endpoint guess is to be called by Player. The cancelGame branch is empty. In either case, the Owner is supposed to close the game after making the choice.

The third version allows one or more players to continue guessing while the game is kept open by the owner. We make use of **rec**-loops for the effect.

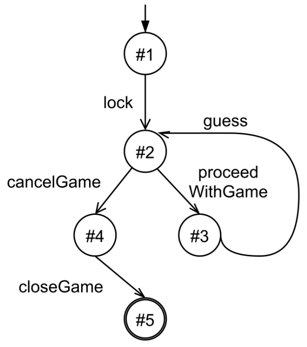
```
protocol RecGuessingGame (role Owner, role Player) {
  field HashedString;
  lock (String, Value) from Owner;
  rec Loop {
    choice at Owner {
    proceedWithGame : { // owner wants players to guess
      guess (String) from Player;
      Loop;
    }
    cancelGame : { // the owner wants to cancel the game
    }
  }
  closeGame () from Owner;
}
```

The **rec** constructor introduces a labelled recursion point. In this case the protocol may continue at the recursion point by means of the Loop label. In any iteration of the loop the owner is called to decide whether the game continues or not (perhaps the secret was found or the owner got tired of playing). If she decides proceedWithGame, a player is given the chance of guessing (by calling endpoint guess) and the owner is called again to decide the faith of the game.

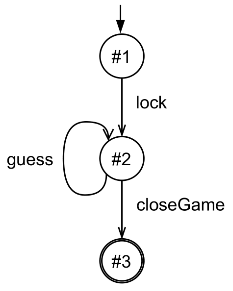
Version three requires a lot of Owner intervention: the continuation of the game depends on her choice—proceedWithGame or cancelGame—after each guess. We want protocols able to terminate automatically, based on guess validation or on the passage of time. Our fourth version takes advantage of the **do**—**interrupt** constructor and the **trigger** declaration:

```
protocol GuessingGame (role Owner, role Player) {
  field HashedString;
  lock (String, Value) from Owner {
    // triggers for funds and slot
    funds trigger closeGame;
    slot trigger closeGame;
  };
  do {
    rec Loop {
      guess (String) from Player;
      Loop;
    }
  }
  interrupt {
    // close the game when one of the triggers is activated
    closeGame () from Contract;
  }
}
```

In this last version of the protocol, after locking the secret, the Owner has no further involvement. The game ends when one of the triggers is activated. Declarations **slot trigger** closeGame and **funds trigger** closeGame contain the keywords **slot** and **funds**, that instruct the compiler to generate functions in the business logic module where the programmer may define the conditions for these triggers. The primitive role Contract signs the closeGame operation. This is not an endpoint, but an interaction that is executed automatically.



(a) RecGuessingGame



(b) GuessingGame

Fig. 6: Finite state automata for two SmartScribble protocols

Constructors **rec**, **choice**, **do**—**interrupt** and protocol definitions share Scribble's syntax entirely. Interactions are simplified: we remove to <recipient> present in Scribble syntax because in our setting the recipient is always the contract. The declarations **field**, **funds trigger**, **slot trigger** are exclusive to SmartScribble.

4 Smart Contract Generation from SmartScribble

This section details the smart contracts generated from SmartScribble protocols and explain how developers can add custom business logic to complete contract code. To ensure the validation of participants' interactions with the contract we construct a finite state machine from each SmartScribble protocol, whose implementation is automatically generated by our compiler.

Protocols are governed by finite state machines. Figure 6 depicts the automata for the recursive and the **do**—**interrupt** guessing game in section 3. The correspondence is such that endpoints in the automaton correspond to edges, and pre- and post-interaction points in a protocol correspond to nodes. Sequential composition, **choice** and **rec** generate appropriate wiring. Interrupts call the associated function generating new edges, as in the case of closeGame where an edge links state #2 to the terminal state #3 (figure 6b).

The SmartScribble compiler [11] generates code divided in three different modules: *Domain and Library Module*, *Smart Contract Module* and *Business Logic Module*. In this section we give a brief overview of the three modules.

Domain and Library Module includes declaration of errors, the interface for the contract, the definition of state machine inputs, states and the functions to interact with the state machine:

- initialiseSM initialises the state machine; succeeds only if not initialised before.
- runContractStepSM performs a transition, given an input.

– getCurrentStateSM consults the current state.

To implement the state machine we use the Plutus State Machine library, part of the standard Plutus package [13].

Smart Contract Module contains code activated when interacting with endpoints. For example, in the GuessingGame protocol, lock registers the two triggers and calls the corresponding function in the Business Logic Module. The latter function returns either an error or the fields to be stored at the new state. If an error is received, no state transition is performed. Otherwise, the machine advances to the next state and sets its new contents. Changes to the value field stored in the state results in the transfer of funds between the node interacting with the endpoint and the contract. This module also contains code for state transition that is used to define the state machine and boilerplate code specific to Plutus' contracts.

Business Logic Module contains signatures for each of the endpoints in the contract. The actual code is meant to be written by the contract developer. The interaction

```
lock (String, Value) from Owner;
```

in the protocol requires a function

```
lock :: String -> Value -> Contract (Either StateContents
    Error)
```

(signature simplified) in the Smart Contract Module that returns either a new StateContents (a tuple composed of the fields stored in the state) or an error. If the value is non-positive, lock returns an error including an error message; otherwise returns a StateContents, that is a pair, composed of the hashed string corresponding to the input and the value. These are the two fields to store in the new state of the state machine (state#2 in figure 6b, reached via the guess-labelled edge). The triggers: **funds trigger** and **slot trigger** associated with lock, generate functions with signatures:

```
lockFundTrigger :: String -> Value -> Contract (Value -> Bool)
lockSlotTrigger :: String -> Value -> Contract Slot
```

for the respective triggers. In lockFundTrigger, the developer should add an expression with the condition to activate the trigger, e.g., (\funds -> **funds** `V.leq` 0). In lockSlotTrigger we specify the Slot that activates the trigger.

Corresponding to the

```
guess (String) from Player;
```

in the protocol, a function with the following signature must be written.

```
guess :: String -> Contract (Either StateContents Error)
```

Function guess reads the secret from the machine state (a HashedString) and compares it with the hashed version of the input string. If they match, it returns a pair whose second component is zero ADA, otherwise it returns an appropriate error message. The caller to guess (in module Smart Contract Module) detects the difference in the value field of the state and credits the difference in the client's account. Finally, the closeGame () from Contract interaction point needs a function with the same name that, in this case, returns a state with HashedString "Game over" and zero ADA as its fields.

The complete code of the module is in appendix C; the code for the three functions and two triggers amounts to 13 lines.

5 Evaluation

In this section we demonstrate SmartScribble effectiveness by replicating our running example with SmartScribble's generated code. We also carry out a performance analysis comparing the generated code with expert implementations. (Throughout this section, when referring to expert implementations, we intend to refer to the implementations suggested by the Plutus team.)

5.1 SmartScribble in Action

In this section we return to the guessing game. Using the Plutus code generated from the SmartScribble protocol GuessingGame presented in section 3 and the business logic briefly described in section 4, we replicate the scenario in figure 2. We use the same conditions that previously resulted in error and the same input as in figure 3. The final balances are presented in figure 7. In this case, the player can retrieve the prize for guessing correctly; the simulation sees the player terminate with 13 Lovelace. The owner ends with 7 Lovelace as a result of the prize they deposited in the first lock; no funds remain in the script. Thanks to the integration with the state machine, the contract now impedes the second lock from taking effect. As seen in figure 10 from appendix B, the second lock request is labelled as invalid, and no transaction is created:

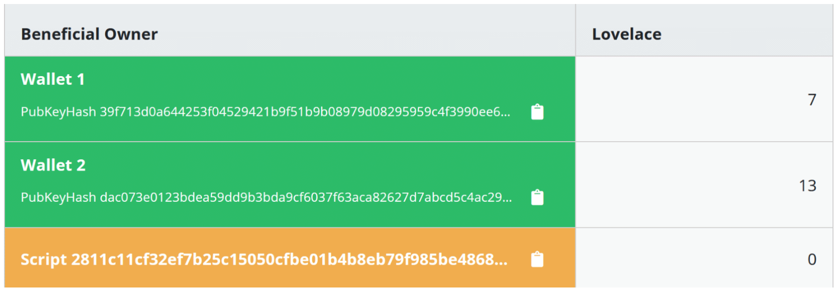
```
"Previous lock detected. This lock produces no effect"
```

This interaction is invalid because for state#2, the state machine only accepts closeGame and guess as inputs (see figure 6b). Thus, input lock is considered as invalid and is promptly detected. By doing this, the integration with a state machine guarantees that the game functions as intended by the developer of the protocol. Unlike the implementation proposed by the Plutus providers (that we tested in section 2), this version of the smart contract works as intended.[2]

5.2 Performance Analysis

The discussion for SmartScribble's results follows two different facets: the line of code (LOC) comparison and the evaluation of performance impact on the network.

[2] Source code available at https://git.lasige.di.fc.ul.pt/-/snippets/4



Beneficial Owner	Lovelace
Wallet 1 PubKeyHash 39f713d0a644253f04529421b9f51b9b08979d08295959c4f3990ee6...	7
Wallet 2 PubKeyHash dac073e0123bdea59dd9b3bda9cf6037f63aca82627d7abcd5c4ac29...	13
Script 2811c11cf32ef7b25c15050cfbe01b4b8eb79f985be4868...	0

Fig. 7: Final balances for setup in figure 3, using SmartScribble (Plutus Playground)

LOC Comparison

To perform the LOC comparison, we use the guessing game together with three other protocols, representative of other smart contract use cases.

Ping Pong A simple protocol that alternates between ping and pong operations, *ad eternum*. No business logic is required for this protocol.

```
protocol PingPongRec (role Client){
  // this protocol stays in the Loop indefinitely
  initialise() from Client;
  rec Loop {
    ping() from Client;
    pong() from Client;
    Loop;
  }
}
```

Crowdfunding A crowdfunding where the owner of the contract starts a campaign with a goal (in ADA), and contributors donate to the campaign. When the owner decides to close the campaign, all the donations stored in the contract are collected.

```
protocol Crowdfunding (role Contributor, role Owner){
  init (Value) from Owner;
  rec Loop {
    choice at Owner{
      continue : {
        contribute (Value) from Contributor;
        Loop;
      }
      closeCrowdfund : {}
    }
  }
}
```

Auction A protocol where a seller starts an auction over some token, setting the time limit. Buyers bid for the token. When the auction is over, the seller

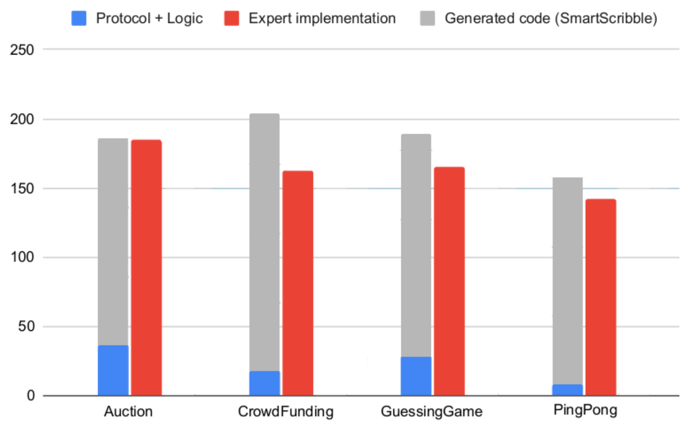


Fig. 8: Comparison of the lines of code (LOC) in SmartScribble against expert implementations, for the four use cases.

collects the funds of the highest bid and the corresponding bidder gets the token.

```
protocol Auction (role Seller, role Buyer) {
  // The ByteString is supposed to simbolize the Token
     being auctioned, Value is the minimum allowed bid
  field Value,ByteString;
  initialise (Value, ByteString) from Seller{
      trigger slot closeAuctionTrigger;
  };
  do{
    rec Loop {
      bid (Value) from Buyer;
      Loop;
    }
  } interrupt {
    closeAuctionTrigger () from Contract;
  }
  collectFundsAndGiveRewards () from Seller;
}
```

The LOC analysis (detailed in appendix D.1) is summarized in figure 8. We observe that, in all our examples, SmartScribble saves the programmer from manually writing an exorbitant amount of code (most of it boilerplate), supporting our claim that SmartScribble makes Plutus learning process easier, as it generates all standardized boilerplate code automatically, and allows developers to focus only on the logic portion of the contract. In the examples we presented, the business logic is a small portion of the contract's code. We expect this to be the case for the majority of situations, although the weight of the logic section may exceed Auction's 24.67% in complex scenarios. When comparing SmartScribble results with the suggested implementations, we observe a considerable difference between the lines of code written when using SmartScribble and the total length of the contracts proposed by the Plutus team. These results illustrate SmartScribble utility in reducing the learning floor for Plutus and the tremendous amount of effort saved.

Performance Impact

We now gather information regarding SmartScribble performance and compare it to expert implementations with similar functionality that: (i) do not use state machines; (ii) implementation that uses a state machine. Lastly we compare the performance of the different versions of the guessing game in section 3. All metrics refer to the on-chain portion of the contract, as it is the part that impacts the blockchain performance. For this comparison, we use a new protocol:

ShareLockFunds Protocol where a user locks some ADA in a script output and defines two recipients. Then the ADA is evenly split between the recipients (sort of a shared account between two people).

We cannot proceed without introducing the units we use for samples. We collect data for CPU and memory usage. These results were gathered using the budget functionality of Plutus API, as suggested by the Plutus team[3]. The API allows its users to determine the CPU and memory consumption associated with contracts. To measure CPU performance, we use *ExCPU*, a unit with no fixed base, and the output values depend on the types and structures used. For memory we use *ExMemory*: a unit that counts size in machine words, and is proportional. These units allow us to make comparisons between implementations. We stray away from estimating fee costs (in ADA) because fees are very volatile. Moreover, we do not have data about the fee history for smart contracts on Cardano because smart contracts are not deployed yet (as at the time of writing). Moreover, fees depend on the blockchain community, the crypto market, and even the economics of the real world, as a result, such estimations would be unreliable.

The performance analysis is detailed in appendix D.2 and is summarized in the CPU and memory distribution for the expert versions *without* and *with* state machines, in figures 11 and 12 of appendix D.2, respectively. Performance-wise, the results show that state machines have an inescapable additional overhead when compared to contracts that do not make use of state machines, due to the nature and implementation of the Plutus state machine library. Does this mean we should avoid state machines? The answer is not that simple if we recall the version of the Guessing Game presented in section 2, that does not use any state machine but exposes vulnerabilities that SmartScribble overcomes. SmartScribble adds a substantial overhead, but guarantees that the contract reacts to interactions as expected. Contrarily, in versions that do not employ state machines, behaviour can be unreliable. Additionally, the ratio between

[3] https://github.com/input-output-hk/plutus/issues/3489

CPU usage implementations without state machines and using SmartScribble varies significantly between different use cases: ranging from 49.7% in Guessing Game to 74.8% in Auction. This difference can indicate that, in extreme cases where the on-chain code is very complex the resource consumption may mirror a state machine implementation.

SmartScribble and implementations *with* state machines have comparable results as both have identical structures. Using a state machine appears to always carry a significant overhead. The difference between Guessing Game and Auction resource usage is less significant than their counterparts without state machine.

We conclude that using state machines does increase resource consumption of the contracts, thus increasing operation fees. Withstanding this observation, the increase in costs can be interpreted as a *price for security*, a long-term investment for Plutus developers and users, where securer contracts behaviour with costlier fees per interaction can save their assets.

6 Conclusion and Future Work

We present SmartScribble—a protocol language for smart contracts—and a compiler that automatically generates all contract code, except the business logic. The generated code relies on state machines to prevent unexpected interactions with the contract. We claim that SmartScribble improves the reliability of contracts by reducing the likelihood of programmers introducing faults in smart contract code. Our language also flattens the learning curve, allowing developers to focus on the business logic rather than on the boilerplate required to setup a contract, namely in Plutus. Preliminary results point to a 1/4 ratio between the number of lines of code written by the programmer and those in the final contract.

This paper constitutes an initial report on using protocol descriptions to generate contract code. Much remains to be done. SmartScribble protocols classify participants under different roles, but we currently do not enforce any form of association of participants to roles. We plan to look into different forms of enforcing the association. Business logic is currently manually written in the contract language (Plutus) and added to the code generated from the protocol. We plan to look into ways of adding more business logic to protocols, thus minimising the Plutus code that must be hand written. Some features of SmartScribble are strongly linked with Plutus. The trigger generation is one of those features: it depends on Plutus libraries for the effect. Nevertheless, we believe that SmartScribble can be adapted to target other languages with minimal changes to the syntax and semantics, provided the target programming languages are powerful enough to support state machines. Generating Solidity code might be an interesting option for the future.

Acknowledgements Support for this research was provided by the FCT through project SafeSessions, ref. PTDC/CCI-COM/6453/2020, by the LASIGE Research Unit, ref. UIDB/00408/2020 and ref. UIDP/00408/2020.

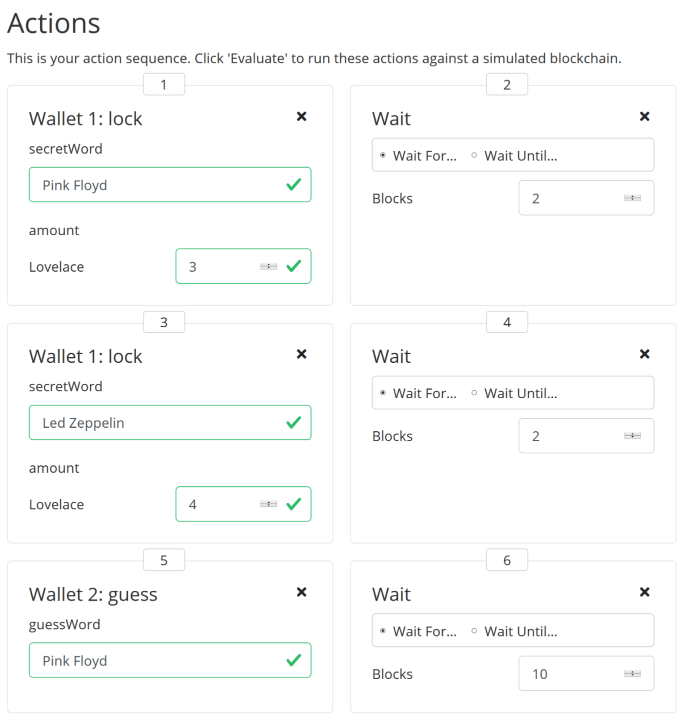
References

1. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (sok). In: International conference on principles of security and trust. Springer (2017)
2. Bartoletti, M., Zunino, R.: Bitml: a calculus for bitcoin smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (2018)
3. Blackshear, S., Cheng, E., Dill, D.L., Gao, V., Maurer, B., Nowacki, T., Pott, A., Qadeer, S., Rain, D.R., Sezer, S., et al.: Move: A language with programmable resources. Libra Assoc (2019)
4. Brünjes, L., Vinogradova, P.: Plutus: Writing Reliable Smart Contracts. IOHK (2020)
5. del Castillo, M.: The dao attacked code issue leads to $60 million ether theft (2016), https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft
6. Chakravarty, M.M., Chapman, J., MacKenzie, K., Melkonian, O., Jones, M.P., Wadler, P.: The extended UTXO model. In: International Conference on Financial Cryptography and Data Security. Springer (2020)
7. Chen, H., Pendleton, M., Njilla, L., Xu, S.: A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. ACM Computing Surveys **53**(3) (2020)
8. Coblenz, M., Oei, R., Etzel, T., Koronkevich, P., Baker, M., Bloem, Y., Myers, B.A., Sunshine, J., Aldrich, J.: Obsidian: Typestate and assets for safer blockchain programming. TOPLAS 2020 **42**(3) (2020)
9. Conchon, S., Korneva, A., Zaïdi, F.: Verifying smart contracts with cubicle. In: Formal Methods. FM 2019 International Workshops, Revised Selected Papers, Part I. Lecture Notes in Computer Science, vol. 12232. Springer (2019)
10. Das, A., Balzer, S., Hoffmann, J., Pfenning, F., Santurkar, I.: Resource-aware session types for digital contracts. In: IEEE 34th Computer Security Foundations Symposium (CSF). IEEE Computer Society (2021)
11. Falcão, A., Mordido, A., Vasconcelos, V.T.: Smartscribble's implementation (2021), https://git.lasige.di.fc.ul.pt/amordido/smartscribble
12. IOHK: Plutus playground. https://prod.playground.plutus.iohkdev.io/ (2020)
13. IOHK: Language plutus contract state machine (2021), https://marlowe-playground-staging.plutus.aws.iohkdev.io/doc/haddock/plutus-contract/html/Plutus-Contract-StateMachine.html
14. Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: A provably secure proof-of-stake blockchain protocol. In: Annual International Cryptology Conference. Springer (2017)
15. Liquidity, http://www.liquidity-lang.org/doc/index.html
16. Mavridou, A., Laszka, A.: Designing secure ethereum smart contracts: A finite state machine based approach. In: International Conference on Financial Cryptography and Data Security. Springer (2018)
17. Rholang, https://github.com/rchain/rchain/tree/master/rholang-tutorial
18. Sayeed, S., Marco-Gisbert, H., Caira, T.: Smart contract: Attacks and protections. IEEE Access **8** (2020)
19. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014) (2014)
20. Yoshida, N., Hu, R., Neykova, R., Ng, N.: The scribble protocol language. In: International Symposium on Trustworthy Global Computing. Springer (2013)

A Plutus code for vulnerable guessing game

```haskell
import             Control.Monad          (void)
import qualified   Data.ByteString.Char8  as C
import             Data.Map               (Map)
import qualified   Data.Map               as Map
import             Data.Maybe             (catMaybes)
import             Ledger                 (Address, Datum (Datum)
    , ScriptContext, Validator, Value)
import qualified   Ledger
import qualified   Ledger.Ada             as Ada
import qualified   Ledger.Constraints     as Constraints
import             Ledger.Tx              (ChainIndexTxOut (..))
import qualified   Ledger.Typed.Scripts   as Scripts
import             Playground.Contract
import             Plutus.Contract
import qualified   PlutusTx
import             PlutusTx.Prelude       hiding (pure, (<$>))
import qualified   Prelude                as Haskell

newtype HashedString = HashedString BuiltinByteString deriving
     newtype (PlutusTx.ToData, PlutusTx.FromData, PlutusTx.
    UnsafeFromData)

PlutusTx.makeLift ''HashedString

newtype ClearString = ClearString BuiltinByteString deriving
    newtype (PlutusTx.ToData, PlutusTx.FromData, PlutusTx.
    UnsafeFromData)

PlutusTx.makeLift ''ClearString

type GameSchema =
        Endpoint "lock" LockParams
        .\/ Endpoint "guess" GuessParams

data Game
instance Scripts.ValidatorTypes Game where
    type instance RedeemerType Game = ClearString
    type instance DatumType Game = HashedString

gameInstance :: Scripts.TypedValidator Game
gameInstance = Scripts.mkTypedValidator @Game
    $$(PlutusTx.compile [|| validateGuess ||])
    $$(PlutusTx.compile [|| wrap ||]) where
        wrap = Scripts.wrapValidator @HashedString
            @ClearString

hashString :: Haskell.String -> HashedString
hashString = HashedString . sha2_256 . toBuiltin . C.pack

clearString :: Haskell.String -> ClearString
clearString = ClearString . toBuiltin . C.pack

validateGuess :: HashedString -> ClearString -> ScriptContext
   -> Bool
validateGuess hs cs _ = isGoodGuess hs cs

isGoodGuess :: HashedString -> ClearString -> Bool
isGoodGuess (HashedString actual) (ClearString guess') =
   actual == sha2_256 guess'

gameValidator :: Validator
gameValidator = Scripts.validatorScript gameInstance

gameAddress :: Address
gameAddress = Ledger.scriptAddress gameValidator

data LockParams = LockParams
    { secretWord :: Haskell.String
    , amount     :: Value
    }
    deriving stock (Haskell.Eq, Haskell.Show, Generic)
    deriving anyclass (FromJSON, ToJSON, ToSchema, ToArgument)

newtype GuessParams = GuessParams
    { guessWord :: Haskell.String
    }
    deriving stock (Haskell.Eq, Haskell.Show, Generic)
    deriving anyclass (FromJSON, ToJSON, ToSchema, ToArgument)

lock :: AsContractError e => Promise () GameSchema e ()
lock = endpoint @"lock" @LockParams $ \(LockParams secret amt)
    -> do
    logInfo @Haskell.String $ "Pay " <> Haskell.show amt <> "
        to the script"
    let tx        = Constraints.mustPayToTheScript (
        hashString secret) amt
    void (submitTxConstraints gameInstance tx)

guess :: AsContractError e => Promise () GameSchema e ()
guess = endpoint @"guess" @GuessParams $ \(GuessParams
   theGuess) -> do
   -- Wait for script to have a UTxO of a least 1 lovelace
    logInfo @Haskell.String "Waiting for script to have a UTxO
         of at least 1 lovelace"
    utxos <- fundsAtAddressGeq gameAddress (Ada.
        lovelaceValueOf 1)
    let redeemer = clearString theGuess
        tx       = collectFromScript utxos redeemer

    logInfo @Haskell.String "Submitting transaction to guess
        the secret word"
    void (submitTxConstraintsSpending gameInstance utxos tx)

findSecretWordValue :: Map TxOutRef ChainIndexTxOut -> Maybe
    HashedString
findSecretWordValue =
  listToMaybe . catMaybes . Map.elems . Map.map
      secretWordValue

secretWordValue :: ChainIndexTxOut -> Maybe HashedString
secretWordValue o = do
  Datum d <- either (const Nothing) Just (_ciTxOutDatum o)
  PlutusTx.fromBuiltinData d

game :: AsContractError e => Contract () GameSchema e ()
game = do
    logInfo @Haskell.String "Waiting for guess or lock
        endpoint..."
    selectList [lock, guess]

endpoints :: AsContractError e => Contract () GameSchema e ()
endpoints = game

mkSchemaDefinitions ''GameSchema

$(mkKnownCurrencies [])
```

B Plutus playground simulation for the guessing game scenario


Actions

This is your action sequence. Click 'Evaluate' to run these actions against a simulated blockchain.

1

Wallet 1: lock

secretWord: Pink Floyd ✓

amount

Lovelace: 3 ✓

2

Wait

[✓ Wait For...] Wait Until...

Blocks: 2

3

Wallet 1: lock

secretWord: Led Zeppelin ✓

amount

Lovelace: 4 ✓

4

Wait

[✓ Wait For...] Wait Until...

Blocks: 2

5

Wallet 2: guess

guessWord: Pink Floyd ✓

6

Wait

[✓ Wait For...] Wait Until...

Blocks: 10

Fig. 9: Complete version of figure 3. This is the setup to run the scenario illustrated in figure 2 in Plutus Playground: the owner (Wallet 1) makes two consecutive locks ("Pink Floyd" and "Led Zeppelin"), the player (Wallet 2) guesses the first secret. Wait in between actions, to let the simulator process the inputs.

```
Validating transaction: [...]
=== Slot 1 ===
Wallet 1:
EndpointCall ("tag":"lock","value":3,"secretWord":"Pink Floyd")
Validating transaction: [...]
=== Slot 2 ===
Wallet 1: "No previous state found, initialising SM with state:
LockState (HashedString ...) and with value: 3"
=== Slot 3 ===
Wallet 1:
EndpointCall ("tag":"lock","value":4,"secretWord":"Led Zeppelin")
Wallet 1: "Previous lock detected.
This lock produces no effect"
=== Slot 4 ===
=== Slot 5 ===
Wallet 2:
EndpointCall ("tag":"guess","guessWord":"Pink Floyd")
Wallet 2: "Congratulations, you won!"
Validating transaction: [...]
=== Slot 6 ===
Wallet 1: "Closing the game"
Wallet 2:
"Successful transaction to state: LockState (HashedString ...)"
Validating transaction: [...]
=== Slot 7 ===
Wallet 1:
"Successful transaction to state: CancelGameState (HashedString ...)"
```

Fig. 10: Log resulting from the setup in figure 3. This time using the contract generated from the protocol using SmartScribble; observe that the second lock fails (`Slot 3`) and therefore the guess from Wallet 2 is successfully validated

C Plutus code for business logic

Guessing game logic module (SmartScribble)

```
module GuessingGameLogic
  ( lock
  , guess
  , closeGame
  ) where

import                Control.Lens
import                Control.Monad             (void)
import                GHC.Generics              (Generic)
import                Language.PlutusTx.Prelude
import qualified      Ledger.Ada                as Ada
import qualified      Ledger.Value              as V
import qualified      Data.Text                 as T
import qualified Data.ByteString.Char8    as C
import qualified GuessGameLibrary         as G
import PlutusTx.Prelude
import qualified Prelude


hashString :: String -> HashedString
hashString = HashedString . sha2_256 . C.pack

zeroLovelace :: Value
zeroLovelace = Ada.lovelaceValueOf 0

lockFundTrigger :: G.AsError e => String -> Value -> Contract
   G.Schema e (Value -> Bool)
lockFundTrigger str val =
    pure $ (\presentVal -> presentVal `V.leq` zeroLovelace)

lockSlotTrigger :: G.AsError e e => String -> Value ->
    Contract G.Schema e Slot
lockSlotTrigger param1 param2 = pure $ 10

lock :: G.AsError e => String -> Value -> Contract G.Schema e
    (Either G.State G.Error)
lock str val = pure $
  if  val `V.leq` zeroLovelace
    then Right $ Error $ T.pack $ "The prize must be greater
        than 0; got " <> show val
    else Left (hashString str, val)

guess :: G.AsError e => String -> Contract G.Schema e (Either
   G.State G.Error)
guess str = do
    secret <- mapError (review _GuessingGameSMError) $ G.
        getCurrentStateSM G.client
    if secret == hashString str
      then do
        logInfo @String "Congratulations, you won!"
        pure $ Left (hashString str, zeroLovelace)
      else pure $ Right $ Error "Incorrect guess, try again"

closeGame :: G.AsError e => Contract G.Schema e (Either G.
    State G.Error)
closeGame = do
  logInfo @String "Closing the game"
  pure $ Left (hashString "Game over", zeroLovelace)
```

D Results

In this appendix, we start by analysing the generated lines of code by SmartScribble and compare it to the amount of code written by the developer and to popular implementations of such protocols in appendix D.1. In appendix D.2 we compare the performance of SmartScribble for popular use cases with expert implementations. Section 5 presents a discussion about the evaluation results.

D.1 Lines of Code Analysis

This section compares the amount of lines of code (LOC), written by the developer, of implementations suggested by Plutus' providers, and those generated by our compiler. To carry out the comparison, we use the guessing game together with three other protocols, representative of other smart contract use cases.

Ping Pong A simple protocol that alternates between ping and pong operations, *ad eternum*. No business logic is required for this protocol.

```
protocol PingPongRec (role Client){
  // this protocol stays in the Loop indefinitely
  initialise() from Client;
  rec Loop {
    ping() from Client;
    pong() from Client;
    Loop;
  }
}
```

Crowdfunding A crowdfunding where the owner of the contract starts a campaign with a goal (in ADA), and contributors donate to the campaign. When the owner decides to close the campaign, all the donations stored in the contract are collected.

```
protocol Crowdfunding (role Contributor, role Owner){
  init (Value) from Owner;
  rec Loop {
    choice at Owner{
      continue : {
        contribute (Value) from Contributor;
        Loop;
      }
      closeCrowdfund : {}
    }
  }
}
```

Auction A protocol where a seller starts an auction over some token, setting the time limit. Buyers bid for the token. When the auction is over, the seller collects the funds of the highest bid and the corresponding bidder gets the token.

```
protocol Auction (role Seller, role Buyer) {
  // The ByteString is supposed to simbolize the Token
      being auctioned, Value is the minimum allowed bid
  field Value,ByteString;
  initialise (Value, ByteString) from Seller{
      trigger slot closeAuctionTrigger;
  };
  do{
    rec Loop {
      bid (Value) from Buyer;
      Loop;
    }
  } interrupt {
    closeAuctionTrigger () from Contract;
  }
  collectFundsAndGiveRewards () from Seller;
}
```

Protocol	Protocol (LOC)	Generated (LOC)	Logic (LOC)	Suggested (LOC)	$\frac{\text{Protocol}}{\text{Generated}}$	$\frac{\text{Protocol+Logic}}{\text{Generated}}$	$\frac{\text{Protocol+Logic}}{\text{Suggested}}$
Ping Pong	8	177	0	142	4.52%	4.52%	5.63%
Crowd funding	12	187	6	163	6.42%	9.63%	11.04%
Guessing Game	16	162	13	165	9.88%	17.90%	17.58%
Auction	15	150	22	185	10.00%	24.67%	20.00%

Table 1: From left to right: LOC of the SmartScribble protocol, LOC of the generated code, LOC of the business logic, LOC for an implementation suggested by an expert, ratio between LOC of the protocol and the generated code, ratio between LOC written by the programmer (protocol + business logic) and the entire contract, ratio between LOC written by the developer and suggested implementation

Table 1 summarizes the analysis. Depending on the protocol, the amount of generated code varies from 150 to 187 lines. In all our examples, the generated code is at least 10× larger than the protocol. The business logic greatly varies between contracts; nevertheless, it is important to note that it is extremely likely to be a small portion of the complete contract. We observe that the ratio between all the code written by the programmer (that is, the protocol and the business logic code) and the Plutus code that would otherwise be manually written is less than 1/4 in all analysed contracts. When we compare the suggested implementations[4] with SmartScribble, we conclude that the manually written code

[4] Implementations available in the Plutus repository

with SmartScribble is at most 1/5 of the code in the suggested implementation. Even in implementations developed by experts, the boilerplate portion of the contract is significant. This is represented in figure 8 (section 5), where we represent the distribution of the code in SmartScribble against the expert implementations.

D.2 Performance Analysis

In this section, we gather information regarding SmartScribble performance and compare it to expert implementations with similar functionality that: (i) do *not use* state machines; (ii) *use* state machines. Lastly we compare the performance of the different versions of the guessing game in section 3. All metrics refer to the on-chain portion of the contract, as it is the part that impacts the blockchain performance.

In order to increase the sample size for this section, we present the ShareLockFunds, inspired by an example with the same name from the Plutus team[5]:

ShareLockFunds Protocol where a user locks some ADA in a script output and defines two recipients. Then the ADA is evenly split between the recipients (sort of a shared account between two people).

```
protocol ShareLockFunds (role Recipient, role Owner) {
    field PubKeyHash, PubKeyHash; // storing owner and
        recipient public keys
    lock (Value,PubKeyHash, PubKeyHash) from Owner;
    withdrawRecipient () from Recipient;
    withdrawOwner () from Owner;
}
```

Performance Units
Before we "dive" into the data, we explain the units we use for the samples. We collect data for CPU and memory usage. These results were gathered using the budget functionality of Plutus API, as suggested by the Plutus team[6]. The API allows its users to determine the CPU and memory consumption associated with contracts. To measure CPU performance, we use *ExCPU*, a unit with no fixed base, and the output values depend on the types and structures used. For memory we use *ExMemory*: a unit that counts size in machine words, and is proportional. These units allow us to make comparisons between implementations.

We stray away from estimating fee costs (in ADA) because fees are very volatile. Moreover, we do not have data about the fee history for smart contracts on Cardano because smart contracts are not deployed yet (as at the time of writing). Moreover, fees depend on the blockchain community, the crypto market, and even the economics of the real world, as a result, such estimations would be unreliable.

[5] https://plutus-apps.readthedocs.io/en/latest/plutus/tutorials/basic-apps.html
[6] https://github.com/input-output-hk/plutus/issues/3489

Performance Results

Table 2 contains the results for the performance analysis of contracts without state machine ("Plutus") versus SmartScribble.

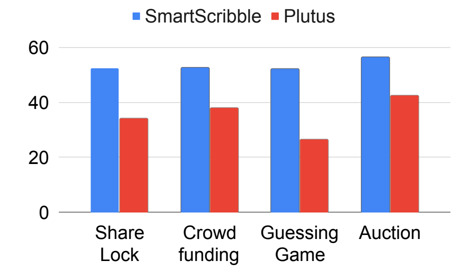
Protocol	CPU Plutus	CPU SmartScribble	$\frac{\text{CPU Plutus}}{\text{CPU SmartScribble}}$	Memory Plutus	Memory SmartScribble	$\frac{\text{CPU Plutus}}{\text{CPU SmartScribble}}$
Auction	4.25×10^8	5.65×10^8	75.3%	10650	14230	74.8%
Crowd funding	3.80×10^8	5.26×10^8	72.2%	9490	13240	71.7%
Guessing Game	2.65×10^8	5.23×10^8	50.6%	6540	13150	49.7%
Share Lock	3.41×10^8	5.25×10^8	64.9%	8490	13210	64.3%

Table 2: Performance results for contracts without state machine ("Plutus") and contracts using SmartScribble

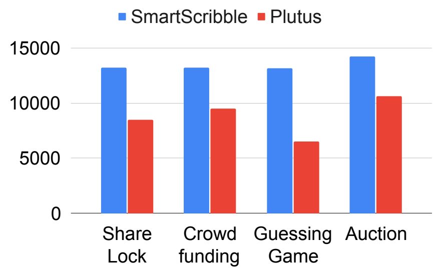
From left to right: CPU usage for "Plutus" and for SmartScribble; CPU usage ratio between "Plutus" and SmartScribble; memory usage for "Plutus", for SmartScribble and ratio between "Plutus" and SmartScribble. CPU usage measured in ExCPU and memory in ExMemory. Figure 11 contains the visual representation of the data in table 2. We observe that regardless of the contract, CPU and memory usage for SmartScribble is superior to Plutus contracts without state machine. The ratio of CPU usage between Plutus and SmartScribble ranges from 50.6% in Guessing Game to 75.3% in Auction. When comparing the memory usage, we observe similar results: 49.7% in Guessing Game to 74.8% in Auction. Keep in mind that, in this scenario, we compare performance with contracts that can hide vulnerabilities, as illustrated in section 2. The implementation of state machines ensures the robustness of the contract against out-of-order interactions. With these tests, we conclude that SmartScribble generated contracts impact the performance compared to implementations of the same contract that do not use state machines. Nonetheless, this comparison is unfair, since "Plutus" contracts are more likely to contain vulnerabilities, we expand this rationale in section 5. For this reason, we now focus on the comparison between SmartScribble and specialist implementations of Plutus contracts that rely on state machines.

Table 3 present the comparison results between contracts using state machine and SmartScribble.

Figure 12 contains the visual representation of the results. CPU usage is measured in *ExCPU* and memory in *ExMemory*. From the results, we verify that CPU and memory usage for SmartScribble is similar to suggested implementations with state machine. SmartScribble has a slightly better performance in Guessing Game and Auction, and marginally worst result in PingPong. Once again, when comparing the memory usage, the results are almost identical to



(a) CPU usage in ExCPU (scaling factor of 10^7)



(b) Memory usage in ExMemory

Fig. 11: Performance comparison between solutions without state machine and SmartScribble (using values from table 2)

Protocol	CPU Plutus	CPU SmartScribble	$\frac{\text{CPU Plutus}}{\text{CPU SmartScribble}}$	Memory Plutus	Memory SmartScribble	$\frac{\text{Plutus Memory}}{\text{SmartScribble Memory}}$
Auction	5.68×10^8	5.65×10^8	100.6%	14310	14230	100.6%
Guessing Game	5.29×10^8	5.23×10^8	108.7%	13300	13150	101.1%
Ping Pong	5.18×10^8	5.22×10^8	99.3%	13030	13120	99.3%

Table 3: Results for memory and CPU usage for suggested implementations using state machine and contracts using SmartScribble

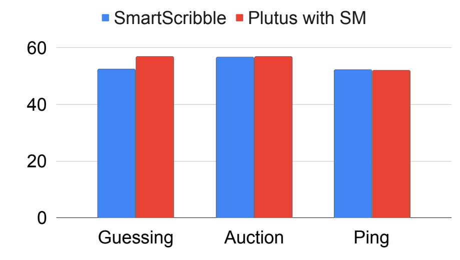
CPU. The resource usage differs less between protocols then the data we presented in table 2.

The contracts generated using SmartScribble have structures similar to those implemented by the Plutus team. The slight advantage in performance can be attributed to some verifications experts perform on-chain that are done off-chain with SmartScribble.

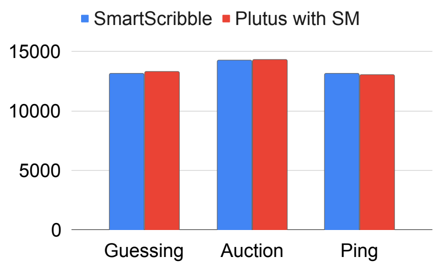
In table 4, we arrive at the data performance that enables us to compare versions of the guessing game presented in section 3.

Contract	CPU $\times 10^8$	Memory
Straight Line	5.25	13210
Choice	5.34	13450
Rec	5.32	13390
Do-Interrupt	5.23	13150

Table 4: Evolution of memory usage for the each version of the guessing game protocol in section 3 using SmartScribble



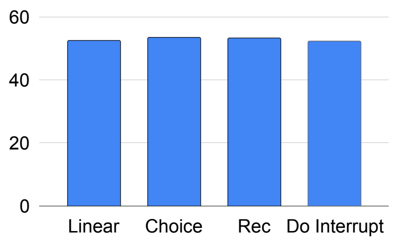
(a) CPU usage in ExCPU (scaling factor of 10^7)



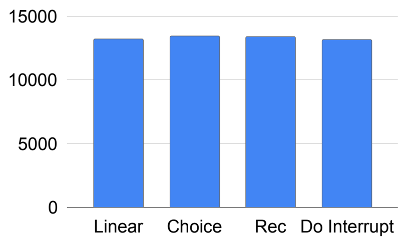
(b) Memory usage in ExMemory

Fig. 12: Performance comparison between solutions that use state machine and SmartScribble (using values from table 3)

We see that CPU and memory usage is almost identical between different versions of the guessing game. Choice version has the highest resource consumption since it is the protocol that generates the largest amount of states and transactions (recall the state machines in section 3). Do-Interrupt uses fewer resources than the straight-line version because it generates one less state, despite having a more complex protocol. A marginal 2.2% fluctuation separates the lowest and upper bound results in both CPU and memory usages. This analysis is illustrated in figure 13.



(a) CPU usage in ExCPU (scaling factor of 10^7)



(b) Memory usage in ExMemory

Fig. 13: Performance comparison between iterations of guessing game (using values from table 4)