Extending the Concert Framework to Verify Solana Programs

João Maria Correia Ramalho Marcelino Gomes
joao.m.r.gomes@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

November 2022

Abstract

Smart contracts are programs stored on a blockchain that help the agreement between two parties without involving an external trusted party. Since smart contracts may carry huge amounts of cryptocurrency and cannot be modified once deployed to the blockchain, it is crucial to ensure their correctness and that they are bug and vulnerability-free. Solana is a recent blockchain that features smart contract capabilities. This blockchain is rapidly growing due to its low transaction fees and speed and is seen as an Ethereum competitor.

In this thesis, we adapt and extend the already existing smart contract verification framework ConCert developed in Coq. Our extension allows the embedding of Solana contracts by introducing Solana concepts, like accounts and ownership. It is also possible to extract smart contracts written in Coq to Rust, programs that are nearly ready to be deployed onto the Solana blockchain. The extended ConCert allows the verification of existing Solana contracts and the development of verified contracts.

The extended framework was evaluated by measuring the proofs accomplished in the model, measuring the complexity of contracts that the framework was able to embed and extract, the time to extract, and the overhead of the extraction. We observed that due to the expressiveness of our model we were unable to embed complex contracts and, as a result, were not verified nor extracted. However, we were able to embed and extract simple contracts with a negligible overhead in terms of extraction time but a noticeable overhead after the extraction.

The main contribution of this thesis is thus the possibility of writing verified contracts in Coq and extracting them to the Solana blockchain.

Keywords: Blockchain, Smart Contracts, Solana, Formal Verification, ConCert, Coq

1. Introduction

1.1. Motivation

Blockchain technology has captured the interest of both researchers and the industry. A blockchain is a distributed ledger technology that allows transactions to be committed without a trusted third party. Ethereum was the first blockchain that separated the consensus layer from the execution of smart contracts.

Smart Contracts are programs that are executed in blockchains, like Ethereum, and allow transactions of resources between parties. Even though these programs tend to be short, it is easy to accidentally introduce errors during development. Smart contracts often control large amounts of cryptocurrency, as such, the presence of one single bug/vulnerability can lead to large financial losses. As a result of these high-value transactions, smart contracts are becoming more appealing to attackers, and as such, there have been some attacks that lead to huge losses, e.g. TheDAO [12], Parity's multi-signature wallet [23] and, more recently, Nomad's crypto bridge [13]. There has been an effort by the community to introduce formal methods and formal verification to this domain. Some examples are the formalization of Yul [19], Ethereum's intermediate language, and the Mi-Cho-Coq framework [20] that aims to verify tezos smart contracts.

Solana is a blockchain written in Rust that achieves consensus using the Proof-of-History mechanism. Both Solana and Ethereum have smart contract capabilities (used for DeFi and NFT), and with the recent rapid expansion of Solana's ecosystem, they have inevitably been compared. Solana is a recent blockchain hence, as far as we know, there are no published solutions that aim to verify smart contracts in its ecosystem.

The goal of this thesis is to formally verify smart contract properties for the Solana blockchain using the Coq proof assistant. For that, we will use the smart contract verification framework ConCert[1] [21, 6, 3], together with Coq and its specification language Gallina we will implement and verify smart contracts. These verified contracts can then be extracted into a set of target languages, including multi-paradigm languages, web development languages, and functional languages. Despite ConCert being prepared to extract for Rust, it is not yet prepared to extract to the Solana blockchain. In this work, we also extend ConCert so it is able to embed Solana contracts and subsequently extract them to Solana.

[1]https://github.com/AU-COBRA/ConCert

1.2. Problem

Once a contract is deployed onto the chain, in most cases, it is impossible to modify it. This feature combined with the lack of smart contract verification leads to unwanted bugs and vulnerabilities and, as a result, provides attackers with an attack vector. Without the use of smart contract verification techniques, bugs and vulnerabilities will keep on appearing and causing huge financial losses to both companies and users of the blockchains. However, many different tools can and are being used to verify smart contracts for distinct blockchains [30].

Since Ethereum is the leading blockchain with smart contract capabilities, several tools can be used to verify contracts written in Solidity (Ethereum's smart contract language), e.g Etherscan[2] [29], an Ethereum blockchain explorer that also offers a source code verification service, Sourcify[3], is another tool for source code verification that is open-source and decentralized that aims to be the infrastructure for other verification tools, etc.

Solana is a recent Ethereum competitor that thrives on the premise of having high transactions per second and low transaction fees. However, being a recent blockchain, Solana, to the best of our knowledge, has no tools in its ecosystem capable of verifying smart contracts. We aim to fill this gap by extending an already existing framework, developed for a different blockchain, in a way that it is possible to encode most of the existing smart contracts

[2]https://etherscan.io/verifyContract
[3]https://github.com/ethereum/sourcify

and verify and deploy them to the chain. The ability to verify smart contracts before deploying them on the Solana blockchain makes it possible to reduce the number of contracts deployed with bugs and vulnerabilities onto the chain.

1.3. Objectives

Taking into consideration the number of bugs and vulnerabilities present in smart contracts and the huge financial losses they cause, the focus of this thesis is to develop a framework capable of embedding, executing, verifying, and extracting smart contracts for the Solana blockchain. More precisely, we aim to:

1. Extend the ConCert framework execution layer by allowing it to represent the Solana blockchain, along with some of its fundamental concepts, and Solana contracts;

2. Extend the ConCert framework extraction layer, more exactly, develop a new Rust extraction tailored to the Solana blockchain, such that they can be deployed onto it;

3. Implement smart contracts using the new execution layer, make use of them to exhibit the newly created Coq to Solana extraction and the new features in the ConCert framework, and formally verify some contract-specific properties.

1.4. Contributions

The main contributions of this thesis are:

- An extension to the ConCert smart contract verification framework. More specifically, an extension of the already existing ConCert execution model to allow Solana programs to be written in Coq. Furthermore, this extension allows the verification of small correctness properties of the developed contracts.

- An extension to the ConCert Rust extraction. The current ConCert's Rust extraction is aimed at the Concordium blockchain and cannot be directly used for other chains. Our extraction extension allows for contracts developed in Coq to be printed in Rust that targets the Solana blockchain. The extracted code will then be deployed onto the chain.

- Two small case studies demonstrating that our extension is capable of developing and partly verifying contracts and extracting them to Rust.

All our code and case studies are available at https://github.com/siimplex/ConCert.

1.5. Dissertation Outline

This document is structured as follows. chap:back presents the background of blockchain and smart contracts, an analysis of the blockchain Solana and of the programming language Rust, and an overview of Coq and two frameworks MetaCoq and ConCert. In chap:relatedwork, an overview of existing works related to program extraction, execution of languages, smart contract verification, and verification in Solana. In chap:proposal, the pipeline of the proposed tool, and the developed execution model are presented and, lastly, the extraction to Solana and its details are discussed. In chap:evaluation, the execution model is evaluated, as two case studies of implemented and extracted contracts and a comparison with the framework that served as the basis for this tool. Finally, chap:conclusion concludes this document, including limitations and future work.

2. Background

2.1. Blockchain

A blockchain is a form of distributed ledger technology where the committed transactions are stored in a chain of blocks that is endlessly growing. This concept was first introduced in 1991 by Stuart Haber and W. Scott Stornetta [14] and later, in 2008, the Bitcoin model was proposed, but only in 2009, it was implemented by Satoshi Nakamoto. Bitcoin was the first decentralized blockchain implementation using a peer-to-peer network to solve the double-spending problem [20], i.e. when a single coin is spent simultaneously more than once.

With today's high computational capacity where a single computer can compute thousands and thousands of hashes per second using hashing as block security is just not enough to prevent tampering. As such, to mitigate this problem, blockchains use different consensus mechanisms like PoW and PoS to regulate the creation of blocks. These systems vary depending on the blockchain, e.g. Bitcoin's PoW makes each miner compute a hash until it finds a correct one by using CPU power, Cardano's PoS uses considerably less CPU power and the validating capability depends on the stake in the network.

2.2. Smart Contracts

Smart contracts were proposed in the 1990s by Nick Szabo [26, 27], long before the creation of Bitcoin, and made great progress after 2013 when the altcoin Ethereum [10] appeared. Smart contracts are just like traditional contracts in the way that they both can be seen as an agreement with specific terms between two or more entities. These digital contracts are programs that keep the agreement terms between the buyer and the seller directly into lines of code, they automatically execute only when set predefined conditions are met.

The contracts are stored on a blockchain-based platform which allows the enforcement of terms of an agreement between parties without the need for a trusted third party. Once contracts are on the chain they inherit the chain's immutable property, in the sense that once created and deployed it is not possible to modify or tamper with the code of the contract.

Ethereum was the first decentralized open-source blockchain to launch smart contract functionality. It is currently the most used altcoin and the second-best cryptocurrency only falling behind Bitcoin. This blockchain-based platform aims to give an alternative protocol to building distributed applications, or dApps, whilst providing different sets of trade-offs that emphasize development time, security, and the ability to efficiently interact with other dApps. To develop smart contracts and dApps, Ethereum provides a virtual general-purpose machine, EVM, where the smart contracts are executed in stack-based machine language. But, since low-level is not that convenient to the programmer, most smart contracts are written in higher-level languages, for example, Solidity.

Ethereum is the current leader in DeFi but, in the past few years, other promising altcoins appeared, one of them being Solana.

2.3. Solana Blockchain

Solana [31] just like its competitor Ethereum, is a blockchain-based platform that allows the development of smart contracts and dApps. Solana has been steadily growing, in the year 2021 SOL (Solana's token) went up more than 10000% according to CoinGecko[4].

[4]https://www.coingecko.com/en/coins/solana

```
init : Chain -> ContractCallContext -> Setup -> option State
```

Listing 1: `init` definition in ConCert.

```
receive : Chain -> ContractCallContext -> State -> option Msg -> option (State
↪  * ActionBody)
```

Listing 2: `reveice` definition in ConCert.

Solana claims to be the fastest-growing ecosystem in the crypto environment and the fastest blockchain on the planet. There are two main reasons for this, high TPS, which is owed to its PoS jointly with PoH and its asynchrony, and secondly, it has a really low cost per transaction compared to its competitors, e.g., gas fees in Ethereum. PoS is a consensus algorithm in which all the network stakeholders agree on the validity of the shared data and secure that data on the blockchain. The blockchain network can only move on to a new block of data once the previous one is secured. Solana, unlike Ethereum and Bitcoin's PoW, uses PoS with PoH. PoS can be seen as an evolution in consensus mechanisms because its less energy-intensive than PoW and provides a major increase in speed and efficiency.

Traditional EVM-based blockchains combine both logic and state into a single contract. Solana takes a slightly different path. Solana's smart contracts are read-only[31], meaning that they do not contain any state data, only contain program logic. Once a contract is deployed on-chain it can be accessed by external accounts. During these interactions, if allowed, the program may store information on the accounts that initiated the interaction.

Solana on-chain programs are compiled using the LLVM compiler infrastructure [15] to an ELF file containing a variation of the BPF. Hence smart contracts can be written in any programming language that can target the LLVM compiler, such as C, C++, and Rust. Using these programming languages it is possible to develop high-performance smart contracts, moreover, Rust solves issues of memory safety and thread concurrency.

2.4. ConCert Framework

ConCert[5] is a smart contract verification framework written in Coq for Coq. ConCert [6] allows the embedding of functional languages into Coq by using meta-programming. In ConCert, there are two ways of representing functional programs: as an AST, deep embedding, and as a Coq function, shallow embedding. Each representation has its advantages: deep embedding is fitting for meta-theoretical reasoning whilst shallow embedding is fit for proving the properties of concrete programs. To connect these two representations of functional programs ConCert uses some of MetaCoq's facilities.

The ConCert framework is split into three layers. The Embedding Layer, which features the embedding of smart contracts into Coq, λ_{smart}, together with its proof of soundness, and the Execution Layer and the Extraction Layer. The latter two are the layers most relevant to our work.

2.4.1 Execution Layer

The Execution Layer provides a model with which is possible to reason about contract execution traces, making it possible to state and prove the properties of interacting smart contracts. In this model smart contracts are comprised of two functions: `init` and `receive`. The `init` function is called after the smart contract is deployed onto the blockchain (Listing 1)

The first parameter of type Chain represents the blockchain, the second parameter ContractCallContext contains all the data about the call to the contract, i.e., who initiated the transaction, the address that sent the call, the address of the contract being called, the balance of the contract being called and the number of tokens being passed in the call. The third parameter of type Setup contains user-defined information to be used in the function. Lastly, this function returns None if it fails, and returns the initial state if it succeeds.

After being deployed, each time the contract gets called the `receive` function is executed (Listing 2)

The first two parameters are the same as the ones in the `init` function. The third parameter, with type State, contains the current state of the contract. Msg is a message type previously defined by the user. Finally, if the function succeeds it returns the resulting state with a list of actions to execute if any. The action can only be one of three types: transfers, calls to other contracts, or contract deployment.

2.4.2 Extraction Layer

Finally, the Extraction Layer [3] provides facilities to extract a smart contract written in Coq into a program in a FSCL. To extract to different FSCL, ConCert maps the abstractions of the Execution Layer to the corresponding abstractions in the target FSCL. This layer works as an interface between the smart contracts written in Coq and the extraction functionality. ConCert currently allows the extraction to Liquidity [4], CameLIGO [5] (both smart contract languages), Midlang, Elm (web programming), and Rust (a multi-paradigm language with a functional subset) [2]. To extract smart contracts they must go first through a process of erasure, and to achieve this ConCert uses MetaCoq erasure with extensions. MetaCoq's verified erasure procedure will erase any type of term into a box, and in specific cases, it can erase an entire term into a box. ConCert [6] erasure procedure extension generates type annotations, which are needed to help with the typing of the target languages. Without these type annotations, there would be ambiguities that cannot be solved by the type checker. Also in this first extension, the erasure procedure for type schemes allows for handling type aliases, i.e., Coq definitions that return a type, which is present in the standard library. The second extension of MetaCoq's verified erasure is *deboxing*, which consists of an optimization process that removes boxes that were left behind by the erasure step. After the erasure process, with extensions and the optimization process, the optimized code is pretty-printed to the target language.

3. Related Work

3.1. Extraction to Statically Typed Languages

Here extraction means getting source code in a target language, the source code must be accepted by the language compiler and must also be able of being integrated into the targeted systems. Coq [17, 18] is one of the several proof assistants that facilitates the extraction of functional languages such as Haskell and OCaml (from Coq proofs or programs). Agda is another proof assistant capable of extraction, it has mechanisms for defining custom backends, but the GHC backend is the most approved.

The ConCert framework [6] extends Coq's proof assistant extraction [3, 4] using MetaCoq's erasure [2]. The current ConCert extraction allows the extraction to a multi-paradigm general-purpose language (Rust), functional smart contract languages (Liquidity, Simplicity), and a functional language for web development (Elm). Coq proof assistant's current extraction is not

[5]https://github.com/AU-COBRA/ConCert

verified due to unverified optimisations that are done during the extraction process. This separation makes it difficult to compare it with the formal procedure given by Letouzey [17], on the other hand, ConCert's separation between erasure and optimisation facilitates such comparisons.

3.2. Execution of Dependently Typed Languages

Related works in this section are related to compiling code from a dependently-typed language to low-level code.

In [21] Nielsen and Spitters present a model specification of smart contracts in Coq, that features inter-contract communication and allows modeling both depth-first execution blockchains and bread-first execution blockchains. This work later served as the basis for the execution layer of ConCert in [6].

The CertiCoq project [1] is a verified compiler for Coq which closes the gap between certified high-level systems and compiled code in machine language. CertiCoq uses the MetaCoq project quoting capabilities, it uses Template-Coq at the first step and verified erasure at the first step. After a few more steps C code is produced and later compiled with CompCert certified compiler [16] to the machine target language.

3.3. Smart Contract Verification

It is essential to use a proof assistant like Coq, Isabelle/HOL [22] or Agda [9], to prove properties of smart contracts with theorems/lemmas. In [7] Arusoaie uses Coq proof assistant to formally verify financial derivatives (written in a purely declarative in a DSL - Findel) by developing an infrastructure that allows to formalise and prove properties that, if proved, will exclude several potential vulnerabilities. Mi-Cho-Coq [8], a Coq framework that formalises the Michelson language (used in the Tezos blockchain), implementing a Michelson interpreter and a way to encode language expressions. Furthermore, it uses the weakest precondition calculus defined as a proven correct function.

Chapman et al. [11] develop a System F, which is a typed λ-calculus that extends the simply-typed λ-calculus. Using Agda, Chapman et al. present one of the first System F that is intrinsically typed and formalised.

3.4. Solana Contract Verification

The related works in this category are concerned with discovering vulnerabilities and verifying smart contracts in Solana.

Pierro et al. [24] develop a web tool capable of verifying the ownership of smart contracts by comparing the source code of the contract, written in a high-level language, with the bytecode deployed in the Solana blockchain. This tool yields some interesting results, the program address advertised by the owner corresponds to the bytecode in the blockchain, thus increasing the trust of the users on the blockchain.

Tavu in his thesis [28] investigates real-smart contracts to discover common vulnerabilities. These vulnerabilities are generally known by Solana developers, and to automate this process verification tools can be used. The author uses different tools with different approaches and run they against real-world contracts, finding vulnerabilities in some of them.

4. Implementation

4.1. Pipeline

This thesis aims to provide a tool that can be used by Solana Smart Contracts developers' giving them the possibility to develop verified contracts. Hence, contributing to the overall safety of the Solana ecosystem.

The tool pipeline can be seen in Figure 1 where the contributions have been highlighted in italic bold and green.

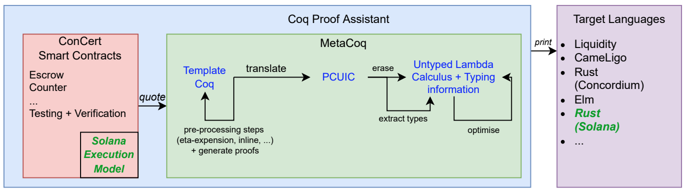

Figure 1: The tool pipeline, starting from a written contract in Gallina and ending with the extraction to the target language.

To start the pipeline, it is necessary to write a smart contract in Gallina using the new Execution Model. During this phase, it is possible to test and verify the properties of the written contracts. The remaining passes, i.e. quoting, translation, erasure, and optimisations, were all borrowed in their entirety from the original ConCert's pipeline. This denotes that MetaCoq [25] is also an integral part of this tool. Furthermore, the certifying passes and optimisations are still applied to mean that the new execution model did not affect this transformation/optimisation phase.

To finalize the pipeline process, the optimised code can be pretty-printed using the already existing pretty-printer developed directly in Coq. This tool uses a new pretty-printer to Rust (based on the already existing one) that has been fine-tuned to be able to print deployable contracts for the Solana Blockchain.

4.2. Solana Execution Model in Coq

4.2.1 Basic Implementation

To extend the existing ConCert execution model it is necessary to consider Solana core concepts[6] such as accounts, transactions, programs, rent, and calling between programs. Moreover, it is also necessary to consider Solana's Rust program support.

A Solana contract can be represented in Coq as a function that receives the blockchain information (type `Chain`), an array of accounts (type `SliceAccountInformation`), and instruction data (`option Msg`): `Chain` represents the view of the blockchain that a contract can access and interact with (e.g. current chain height, finalized height, etc);

`SliceAccountInformation` is a type alias for an array of `AccountInformation` records; and `option Msg` is the instruction data that has been deserialized into the `Msg` type. When this function is called it can either be successful or it can fail with an error type, `ProgramError`, which differs depending on the contract error.

The Coq record `AccountInformation` (Listing 3) is a direct replica of the Solana Rust struct `AccountInfo`, with the exception of the field `rent_epoch` which is not relevant to this model. Similar to ConCert, this representation allows dealing with the contracts in a generic manner. However, it makes reasoning harder, and for this reason, the `Contract` type is preferred to reason with, since it has no `SerializedValues` and has concrete types in their place.

In addition to accounts and their state, it needs to be possible to interact with other programs and accounts. It is possible to do so by using monads and helper functions that ease the use of actions throughout contract behavior.

In this design, the contracts can interact with the blockchain and other programs by transferring lamports, calling programs (`act_call`), deploying programs (`act_deploy`), and using special calls (`act_special_call`) (Listing 4).

[6]https://docs.solana.com/developers

```
Record AccountInformation :=
  build_account {
    account_address : Address;
    account_is_signer : bool;
    account_is_writable : bool;
    account_balance : Amount;
    account_state : SerializedValue;
    account_owner_address : Address;
    account_executable : bool;
  }.
```

Listing 3: Coq's AccountInformation definition.

```
Inductive ActionBody :=
  | act_transfer (to : Address) (amount : Amount)
  | act_call (to : Address) (msg : SerializedValue)
  | act_deploy (c : WeakContract)
  | act_special_call (to : Address) (body : SpecialCallBody)
    with SpecialCallBody :=
        | transfer_ownership (old_owner account new_owner : AccountInformation)
        | check_rent_exempt (account : AccountInformation)
        | check_token_owner (account : AccountInformation)
    with WeakContract :=
        | build_weak_contract
            (process :
               Chain -> (* chain *)
               SliceAccountInformation -> (* accounts *)
               option SerializedValue -> (* instruction data *)
               result unit ProgramError).
```

Listing 4: Solana Action Body definition in Coq.

Special calls is a newly added constructor that facilitates the execution of different context-specific functions. This way these context-specific actions do not clutter the `ActionBody` inductive. Moreover, it facilitates reasoning and extraction to Solana. In this design, there are three special actions: `transfer_ownership`, `check_rent_exempt`, and `check_token_owner`. Special action `transfer_ownership` will transfer ownership of one program, i.e. change the program's owner address. In addition to the destination address, it needs three accounts: the current contract owner, the contract that will transfer ownership, and the contract's new owner. Secondly, the special action `check_rent_exempt` checks if an account is rent exempt, i.e., if it has more than two years' worth of rent in lamports. Lastly, the special action `check_token_owner` is straightforward as it checks if the queried account's owner is the token program. And, just like the previous one, this special action does not modify the environment.

Unlike the original ConCert execution model where they obtained a clear separation between contracts and their interaction with the chain, this separation is no longer that clear with this design. Actions are called in the middle of the program and their effects take place when they are executed.

The `Action` type resembles what is usually considered a transaction, but just like the authors of ConCert, the `Action` and `Transaction` (respectively Listing 5 and Listing 6) types are different [21]. The two aforementioned types differ from each other because an `Action` is done by a user and modifies the blockchain state. The latter can be seen as a fully specified action and additional information.

Solana's transaction model allows for transactions to contain more than one instruction (or action) per transaction. To ease reasoning on transactions and their effects on the chain, our model restrained each transaction to a single instruction (or action). Moreover, the transaction format has been simplified from the Solana docs, specifically the transaction array of signatures which

```
Record Action :=
  build_act {
    act_origin : Address;
    act_from : Address;
    act_body : ActionBody; }.
```

Listing 5: Solana `Action` definition in Coq.

```
Record Tx :=
  build_tx {
      tx_origin : Address;
      tx_from : Address;
      tx_to : Address; !\label{lst:tx!
      tx_amount : Amount;
      tx_body : TxBody;
  }.
```

Listing 6: Solana `Transaction` definition in Coq.

```
Record Environment :=
  build_env {
    env_chain :> Chain;
    env_account_balances : Address -> Amount;
    env_account_owners : Address -> option Address;
    env_contracts : Address -> option WeakContract;
    env_contract_states : Address -> option SerializedValue;
  }.
```

Listing 7: Environment definition in Coq.

are used to check if a user authorized that transaction.

Finally, the Solana calling between programs concept can be achieved using the `act_call` in a program. Program-derived accounts are not explicitly implemented in the execution model like the other core concepts. However, they are implicitly used in the special action `transfer_ownership` when this action is converted into actual Rust code (Section 4.3) delves deeper into this concept and conversion from Coq inductive to Rust code).

In summary, the `Contract` and `WeakContract` definitions were modified. The record `AccountInformation` was added to represent the accounts. The record `Tx` remained the same, but the inductive type `TxBody` was modified according to the new actions added in the inductive `ActionBody`.

4.2.2 Semantics of the Extended Execution Layer

In this section, we start by looking at the `Environment` which contains all the information related to the blockchain and accounts, then we present the `ActionEvaluation` which is used to evaluate the actions and their effects on the chain. Finally, we present the *ChainHelpers* class that is used to aid contract implementation and extraction.

The `Chain` type mentioned previously, has some information about the blockchain but it is not enough to allow the blockchain to execute actions. It needs to be possible to look up deployed contract information. ConCert's original `Environment` definition allows the look up of the contract's functions and state. But, since Solana has some concepts that were not in the original design the environment was extended with look up functions for program ownership (`env_account_owners`) and program balance (`env_account_balances`) (Listing 7).

Like most of the concepts and definitions described in this section, action evaluation is also based on the original action evaluation in ConCert. When actions are executed it is necessary to evaluate the effects of the actions. This is defined as a "proof-relevant" relation `ActionEvaluation` in Coq, with type `Environment` → `Action` → `Environment` → `Type` [21].

This relation is defined by four cases: transfer tokens, contract deployment, contract calls, and special calls. In Listing 8, it is presented the details of the transfer case.

The special call case is evaluated as any of the other cases but has a specific evaluation relation `SpeciallCallBodyEvaluation` to evaluate the body of this call.

Now with the `Environment`, there is enough information to evaluate actions. ConCert further augments this type to keep track of actions to execute and also define the meaning of a step

```
Inductive ActionEvaluation
          (prev_env : Environment) (act : Action)
          (new_env : Environment) (new_acts : list Action) : Type :=
  | eval_transfer :
      forall (origin from_addr to_addr : Address)
             (origin_acc from_acc to_acc : AccountInformation)
             (amount : Amount),
        amount >= 0 ->
        amount <= env_account_balances prev_env from_addr ->
        account_address origin_acc = origin ->
        account_address from_acc = from_addr ->
        account_address to_acc = to_addr ->
        address_is_contract to_addr = false ->
        act = build_act origin from_addr (act_transfer to_addr amount) ->
        EnvironmentEquiv
          new_env
          (transfer_balance from_addr to_addr amount prev_env) ->
        new_acts = [] ->
        ActionEvaluation prev_env act new_env new_acts
  | eval_deploy :
      ...
  | eval_call :
      ...
  | eval_special_call :
      ...
.
```

Listing 8: Action Evaluation relation in Coq.

```
Class ChainHelpers :=
  build_helpers {
    next_account : SliceAccountInformation -> Z ->
                    result AccountInformation ProgramError;
    deser_data (A : Type) : SerializedValue -> result A ProgramError;
    deser_data_account (A : Type) : AccountInformation -> result A
    ↪ ProgramError;
    ser_data {A : Type} : A -> SerializedValue;
    ser_data_account {A : Type} : A -> AccountInformation ->
                                   result unit ProgramError;
    exec_act : WrappedActionBody -> result unit ProgramError;
  }.
```

Listing 9: `ChainHelpers` class and functions in Coq.

in a chain with three inference rules. This part of the execution model was not required to suffer any major changes and, as such, it was kept almost fully. The only exception was a few small changes to accommodate previously mentioned additions to the chain and some lemmas and theorems that needed to be adjusted to be accepted.

Finally, to help with the extraction process the class `ChainHelpers` was created. This class contains the function signatures that are to be used during contract implementation (Listing 9).

4.3. Solana Extraction from Coq

The extraction presented in this thesis is an extension of the extraction in ConCert. More specifically, the new extraction adds a new Rust pretty-printer that targets the Solana blockchain and makes some minor changes to the actual Rust extraction procedure.

4.3.1 Execution Model Helper Functions

To facilitate the extraction procedure to Solana, several helper functions and definitions were implemented. The aforementioned functions are needed because some of the existing functions from the execution model core cannot be used by the extraction due to visibility issues. Moreover, these functions are used as placeholders in contract development, and after extraction, they are replaced by calls to Rust functions that were previously developed. Most of them do not directly affect the chain/environment and as such do not have any lemmas or theorems to prove their effects.

There are two classes of helper functions that were created to facilitate the extraction process: `AccountGetters` and `ChainHelpers` (Listing 8). To ensure that both of these classes' functions are visible by the extraction, all of them are declared as

```
Definition WrappedActionBody_to_ActionBody (wact: WrappedActionBody) :
↪ ActionBody :=
  match wact with
  | wact_transfer from to amount => act_transfer (account_address to) amount
  | wact_call to msg             => act_call (account_address to) msg
  | wact_deploy contract         => act_deploy contract
  | wact_special_call to body    => act_special_call (account_address to) body
  end.
```

Listing 10: WrappedActionBody example in Coq.

global.

The `AccountGetters` consists of a total of seven getter functions that are used to get information from the `AccountInformation` record, one for each field.

The `ChainHelpers` class consists of a broad range of functions from serialization and deserialization functions to a function to simulate action execution.

`WrappedActionBody` is an inductive type like `ActionBody` however instead of each constructor receiving arguments with `Address` type, they receive arguments with `AccountInformation` (Listing 10).

4.3.2 Extraction Approach

The ConCert Framework extraction targets several languages, one of them being Rust for the Concordium blockchain. The extraction procedure is, as we have seen before, composed of transformations, translations, and optimizing passes. ConCert's contract extraction to Concordium builds upon this procedure with a pretty-printer made specifically so that the contracts are allowed on-chain. The new Solana extraction follows the same idea of having a pretty-printer precisely to print contracts that use Solana's crates/libraries.

The approach used can be split into three different parts: generated code from the contract, directly printed code, and remapped code.

Generated Code Code that is generated from contracts that are developed using the Execution Model. This code is generated using the Rust extraction present in the ConCert framework, which will convert all the Coq functions used in the contract into Rust methods.

Directly Printed Code The extraction presented in this project has many functions that consist of directly printed code, i.e. program preamble, contract entrypoint, and functions that convert actions into instructions.

The program preamble consists mainly of auxiliary functions that together with inductive remapping can be used by the contract. Moreover, this preamble contains the Rust crates imports needed for the proper functioning of the contract.

Solana's Rust program exports an entrypoint that is used by Solana runtime to call and invoke that program. In Solana Rust contract development one sets the entrypoint by using its macro and giving the function name as an argument. The given function (usually called `process_instruction`) must have three specific input arguments: program id, list of accounts, and an array of bytes with the instruction data.

To simulate the actions used in the Coq contracts it must be possible to convert each constructor of `ActionBody` into Rust code that has an equivalent behaviour. Listing 11 shows the Rust function that is extracted in order to convert actions and the instruction that are executed for each action. This conversion function however does not allow converting deploy or call actions.

```
fn convert_action(&'a self, act: &ActionBody<'a>)
        -> Result<(), ProgramError> {
    let cact = if let ActionBody::Transfer
        (donator_account, receiver_account, amount) = act {
        if **donator_account.try_borrow_mut_lamports()? >= *amount {
            **receiver_account.try_borrow_mut_lamports()? += amount;
            **donator_account.try_borrow_mut_lamports()? -= amount;
        } else {
            return Err(ProcessError::Error.into());
        };
    } else if let ActionBody::SpecialCall(to, body) = act {
        return self.convert_special_action(to, body);
    } else {
        return Err(ProcessError::ConvertActions.into());
    };
    Ok(())
}
```

Listing 11: Function that converts enum `ActionBody` into code.

```
Definition remap_blockchain_consts : list (kername * string) := [
  remap < @Address > "type ##name##<'a> = Pubkey;";
  remap < @SliceAccountInformation > "type ##name##<'a> =
                            &'a[AccountInfo<'a>];" ].
```

Listing 12: `Address` and `SliceAccountInformation`.

Moreover, if these appear in a contract its execution will result in an error due to their behaviour not being implemented in Rust.

Remmaped Code As mentioned before, remaps are also a relevant part of the extracted code. The execution model uses the type `Address` to identify the contracts. When extracting a contract this type must be remapped to the address type in the target programming language used by the target chain. Also, in the Execution Model, the contract's `Process` function receives a `SliceAccountInformation` type, which in Coq is represented as a list of accounts. Listing 12 presents the remap of these two types.

Both helpers classes' functions `ChainHelpers` and `AccountGetters` must be remapped to their equivalent code in Rust.

In order to use the correct actions in Rust it is necessary to remap the Coq inductive type `WrappedActionBody` into the Rust enum `ActionBody`. This is done using by remapping each Coq constructor to a Rust constructor (Listing 13).

When extracting a Coq contract, the extraction combines the three parts of the extracted code, function remaps and inductive remaps, directly printed code, and the generated code. This code is then printed onto a file, where after setting it up with cargo it can be compiled and deployed onto the Solana blockchain.

5. Evaluation

In this section, the developed extension is evaluated. First, we discuss the model expressiveness, i.e., what can and cannot be written using the extended framework. Afterward, we present which properties of the execution model have been proved. Then we will present one case study, an `Counter` contract where both implementation and extraction will be analyzed. Finally, in the last section, we will compare the proposed extension and the original framework.

```
Definition remap_WrappedActionBody : remapped_inductive :=
  {| re_ind_name := "ActionBody<'a>";
     re_ind_ctors := ["ActionBody::Transfer"; "ActionBody::Call";
                 "__Deploy__Is__Not__Supported"; "ActionBody::SpecialCall"];
     re_ind_match := None |}.
```

Listing 13: ActionBody remapped to Rust.

5.1. Model Expressiveness

The Execution Model allows the user to develop a multitude of contracts. These contracts can contain common actions like transferring tokens between accounts, calling other contracts, and creating new accounts (despite some of these actions cannot be extracted). Furthermore, the execution model allows for more Solana specific actions like verifying if an account is rent exempt and transferring ownership of an account with the help of the token program.

The developed model has most of Solana's core concepts identified in Section 4.2.1. *Rent* is one of the missing concepts since it is usually only used to verify if a contract is rent exempt or not. To do so we use a special call action that does this check. However, this check is most meaningful when the contract is extracted and the code is converted to the actual action of verifying the contract's balance.

Accounts were fully implemented having all fields except its rent related field. It is possible to access all the fields and modify them accordingly.

Transactions are implemented and can be used to analyze the chain, despite not being used when writing contracts. Each transaction has exactly one instruction, unlike Solana, as explained in Section 4.2. This differs from the Solana documentation but facilitates reasoning when analyzing a chain.

In this model *calling between programs* is highly abstracted and program derived accounts are not implemented explicitly. For instance, the special action used to transfer ownership would use program derived accounts if they were explicit (in fact, they are used but only in the extracted program).

In a contract, it is possible to iterate over the list of accounts and check each account's owner and address. For each account, it is also possible to check if it is writable, executable and if it is a signer of that call. Also, it is possible to deserialize and modify an account state and serialize it back to the account. Each contract can have many types of messages, depending on the received message the contract can have different behaviors. All of these features allow for a variety of contracts to be written and reasoned with.

On the other hand, it is not possible to implement contracts that require making complex calls to other programs and Solana's existing programs. For instance, contracts that require heavy interactions with the Solana ecosystem or with external programs that are not represented in the framework.

5.2. Model Properties Proved

The developed tool can be split into two parts: extraction and execution. As aforementioned, the new extraction builds upon the existing one by adding a new print printer that targets the Solana blockchain. Due to this, there are no new proofs or modified proofs in the extraction process.

On the other hand, with the new Execution Model being an extension of ConCert's Execution model, there is the need to update the existing proofs and, in a few cases, add new ones. However, in a few cases, it was not possible to finish the proofs or add new ones for the new concepts due to time constraints.

Helper Classes Proven Properties Since there was a need to have auxiliary functions such as the ones in the `AccountGetters` and `ChainHelpers` classes there are some lemmas and theorems that prove their behavior. For instance, for each account getter, there is a small lemma that ensures that the auxiliary getter returns the same value as using the record projection from the

```
Lemma undeployed_contract_no_out_queue contract state :
  reachable state ->
  address_is_contract contract = true ->
  env_contracts state contract = None ->
  Forall (fun act => (act_from act =? contract) = false) (chain_state_queue
  ↪  state).
```

Listing 14: Undeployed contracts do not have actions Lemma in Coq.

```
Record State := build_state { count : Z ; active : bool ; owner : Address }.
```

Listing 15: Counter program state.

record `AccountInformation`. The `ChainHelpers` class, on the other hand, is missing lemmas for every function to ensure their correct behavior when used in contract implementation due to time constraints.

Core Elements Proven Properties Regarding proofs of core elements of the Execution Model, all of them were kept and if needed were updated/completed. We proved that for any chain trace (i.e. list of chain states) the ending state will not have any actions from undeployed contracts (Listing 14). Furthermore, it was proven that undeployed contracts do not have both outgoing and incoming transactions or calls.

We also demonstrated two properties related to contracts states and addresses: if a state is reachable and a contract state is stored on an address then that address must also have some contract deployed to it; and if a state is reachable and a contract state is stored on an address then that address must be a contract address. Furthermore, it was proven that if a state has a contract state on some address then any other state reachable through the first state must also have the same contract on the same address.

That being said, there are many proofs that were not finished in time, i.e., the lemma is defined but the proof was not finished. The extension presented in this thesis overall removes some of the formal verification done by the original framework since a moderate amount of proofs were not finished in time.

5.3. Case Study A—Counter Contract

As an example of our extension, we consider the implementation of a counter contract. This type of contract allows arbitrary users to have a counter which they can increment or decrement. Contracts like this one are standard uses of smart contracts and appear in many blockchains. This particular implementation is an adaptation of the existing one in ConCert's repository.

Each contract must define its state structure and the permitted messages. The `Counter` contract is a simple contract, it has a state with three fields (Listing 15) and three possible instructions (Listing 16). The state fields contain an integer that holds the counter value, a boolean that tells if the count is active and the address of the owner of the program. The three possible instructions consist of an initializing function, a function to increment the counter, and a function to decrement it.

Each program instruction results in a new state (Listing 17) that is then returned and serialized into the counter account.

The contract is defined by an entry point function that receives a `Chain`, a list of accounts, and an instruction. The entry point function checks if there is an instruction, if not it will throw an

```
Inductive ContractInstruction :=
  | Init (i : Z)
  | Inc (i : Z)
  | Dec (i : Z).
```

Listing 16: Counter program instructions.

```
Definition counter_init (owner_address : Address) (init_value : Z) : State :=
    {| count := init_value ; active := false ; owner := owner_address |}.

Definition increment (n : Z) (st : State) : State :=
    {| count := st.(count) + n ; active := true ; owner := st.(owner) |}.
```

Listing 17: Counter program state init and increment.

```
Lemma counter_increment_correct {prev_state next_state i} :
    increment i prev_state = next_state ->
    0 < i ->
    prev_state.(count) < next_state.(count)
        /\ next_state.(count) = prev_state.(count) + i.
```

Listing 18: Counter program state init and increment.

error, otherwise, it executes the main functionality of the program. This main function iterates over the accounts and obtains the counter program state and, according to it and the received instruction will modify the state accordingly.

When the contract is fully defined in Coq it is possible to verify simple properties regarding its behaviour. For instance, in Listing 18 we verify the correctness of the increment function (can be seen in Listing 17).

At this stage, the Counter contract is ready to be extracted. Once the extraction is complete, a Rust program is generated as it was explained in Section 4.3.

All the previous contract functions are extracted and have both a curried and an uncurried version. This is done to close the gap between Coq and Rust partial applications disparity [3].

Finally, the whole program comes together and can be accessed and interacted with through the Solana contract entrypoint statically defined in the extraction. The program definition[7] and extracted code[8] can be seen in the repository.

5.4. Extension Comparison

The tool proposed in this thesis is an extension of the smart contract verification framework ConCert. In this section, we compare our extension to the original framework. More specifically, it is compared the number of contract examples and variety, i.e., what is possible to write. Moreover, the counter and escrow implementation and extraction will be compared, and the proofs accomplished by each model will be compared.

The implemented extension has two contract examples, a `Counter` contract and an `Escrow` contract which represents what is possible to implement using the extended Execution Model. It is, however, possible to implement additional contracts but they are limited to the existing actions. Since many Solana programs require calls to other programs (e.g. system program, token program, token swap program) it would be needed to develop the Execution model further. The ConCert framework, on the other hand, has a wide variety of contracts in their repository[9] implemented using their execution model. Furthermore, many of these contracts are partly or fully verified, whereas in our extension there is not a fully verified contract.

In Table 1 we present a comparison between the implemented contracts using the extension and the same contracts implemented in the original framework. The contracts prefixed with "Solana" are the contracts implemented with the extension and extracted to Rust (Solana) the ones without a prefix are implemented using ConCert and extracted to Rust (Concordium). The column "Coq Lines" contains roughly the amount of lines needed to implement the contract in Coq. The column "Extraction Time" contains

[7]https://github.com/siimplex/ConCert/blob/master/execution/examples/CounterSolana.v
[8]https://github.com/siimplex/ConCert/blob/master/extraction/examples/extracted-code/solana-extract/counter-extracted/src/lib.rs
[9]https://github.com/AU-COBRA/ConCert

the average time (in seconds) needed to extract each contract to Rust using the corresponding extractions (i.e. for new contracts the new Rust extraction was used). The column "Rust Extracted Lines" contains the number of lines of the Rust code generated by the extraction procedure.

Contract	Coq Lines	Extraction Time (s)	Rust Extracted Lines
ConCert's Counter	82	4.368	501
ConCert's Escrow	125	41.048	1496

Table 1: Contract implementation and extraction comparison.

Both `Counter` contracts are similar in the three aspects, the main difference is the number of lines needed to implement the contract and the additional time to extract it.

Regarding proofs achieved, since our model builds upon ConCert, the proofs in each model are, for the most part, the same. With the exception of the helper classes implemented which proofs were implemented. However, modifying the execution model invalidates some of the existing proofs, and as such those proofs need to be updated accordingly but, as previously mentioned in Section 5.2, it was not possible to complete some of the existing proofs within the time frame of this work. This means that whilst the ConCert model is verified our model is only partly verified since some of the proofs are still to be completed.

6. Conclusions

Blockchain technology and smart contracts are two quickly advancing research topics that have resulted in many platforms with a wide array of applications. Smart Contracts transactions can have huge amounts of cryptocurrency, and consequently, if a vulnerability or bug is found it could easily lead to a significant financial loss. There are previous works that allow smart contracts to be developed, verified, and extracted for blockchains like Ethereum, Concordium, Tezos, and others but, to the best of our knowledge, there is not any work that does this for Solana programs.

This thesis presents an extension to the existing smart contract verification framework in Coq, Concert. This extension aims at reducing the overall amount of bugs and vulnerabilities present in Solana smart contracts by allowing users to develop and verify contracts in Coq and then extract them to Rust to later be deployed onto the Solana Blockchain.

This extension is split into two parts, Execution and Extraction. The Execution allows users to develop and implement existing Solana contracts in Coq with basic contract actions (e.g. contract call, transfer, and deployment) and Solana specific actions (e.g. check account rent and transfer ownership). Furthermore, once a contract is implemented, it is possible to verify its behavior and other properties. The Extraction grants the user the ability to implement contracts using the Execution Model and extract them to Rust so they can be deployed onto the Solana Blockchain.

6.1. Limitations and Future Work

This dissertation proposes an extension to the existing smart contract verification framework ConCert to allow for Solana programs to be implemented using the Execution model and extracted using the Extraction. Although the current extension proves to be useful to address the problem, it can be further improved. This section will discuss the existing limitations together with suggestions to solve them, and suggestions for future work.

The first limitation relates to the Execution Model expressiveness. The presented model allows the implementation of a variety of contracts that are not complex and do not require interacting with other deployed or native Solana programs. However, complex contracts, like DeFi protocols, or contracts that interact heavily with the Solana ecosystem cannot be implemented using the current model.

The second limitation relates to the verification of the Execution Model. Due to time constraints, we were unable to complete all the proofs of the present model, and as such, the model is not fully verified. Verifying the entire model, like it has been done in ConCert [21, 6], would increase the reliability and trustworthiness of this extension. Furthermore, completing the proofs in the execution model would improve the trustworthiness of properties verified in smart contracts.

The third limitation relates to the Extraction aspect of the extension. The current extension can extract both contracts used for case studies, however, both extractions result in a Rust program that cannot be immediately compiled. Most of the compilation issues stem from Rust lifetimes, which require updating most of the lifetimes present in the extracted contract functions. Another error preventing compilation originates from Rust borrow checker however, it can be easily solved by following the compiler hints and error descriptions.

As future work is driven by the present work and existing limitations, the next step would be to optimize the extension, i.e. the Execution Model and Extraction. The existing Execution Model is limited in terms of the complexity of contracts it can represent. Further developing the model and optimizing the way Solana specific actions are implemented would increase the model's expressiveness. Fully verifying the model would be the vehement next step.

In the Extraction procedure, the evident next step would be to generate code that can be automatically compiled by the target language which would allow the generated program to be directly deployed on the target blockchain. Thus, preventing possible bugs or vulnerabilities introduced during the generated program debug.

An interesting path for future work would be the implementation of a DeFi protocol (fully or partially), verifying correctness and safety properties, and finally extracting it to Rust and deploying it onto the Solana blockchain. This would prove difficult with the current model and extraction procedure but it would result in a reliable and useful framework that could produce safe contracts.

Finally, even if the Execution Model is fully verified and the contracts implemented using it are fully verified there is no guarantee that these verified properties will hold after the extraction procedure. Thus, as future work, it would be interesting to verify the entire extraction procedure as it would allow users to develop and verify contracts and extract them directly to the blockchain with the guarantee that the deployed contract is bug and vulnerability free.

References

[1] A. Anand, A. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. S. Belanger, M. Sozeau, and M. Weaver. Certicoq: A verified compiler for coq. In *The third international workshop on Coq for programming languages (CoqPL)*, 2017.

[2] D. Annenkov, M. Milo, J. B. Nielsen, and B. Spitters. Extending metacoq erasure: Extraction to rust and elm. *The Coq Workshop 2021*, 2021.

[3] D. Annenkov, M. Milo, J. B. Nielsen, and B. Spitters. Extracting functional programs from coq, in coq, 2021.

[4] D. Annenkov, M. Milo, J. B. Nielsen, and B. Spitters. Extracting smart contracts tested and verified in coq. *Proceed-*

ings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Jan 2021.

[5] D. Annenkov, M. Milo, and B. Spitters. Code extraction from coq to ml-like languages. *ML Workshop*, 2021.

[6] D. Annenkov, J. B. Nielsen, and B. Spitters. Concert: a smart contract certification framework in coq. *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, Jan 2020.

[7] A. Arusoaie. Certifying findel derivatives for blockchain. *Journal of Logical and Algebraic Methods in Programming*, 121:100665, 2021.

[8] B. Bernardo, R. Cauderlier, Z. Hu, B. Pesin, and J. Tesson. Mi-cho-coq, a framework for certifying tezos smart contracts. In *International Symposium on Formal Methods*, pages 368–379. Springer, 2019.

[9] A. Bove, P. Dybjer, and U. Norell. A brief overview of agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[10] V. Buterin et al. Ethereum whitepaper, 2013.

[11] J. Chapman, R. Kireev, C. Nester, and P. Wadler. System f in agda, for fun and profit. In *International Conference on Mathematics of Program Construction*, pages 255–297. Springer, 2019.

[12] P. Daian. Analysis of the dao exploit, Jan 2016.

[13] C. Faife. Nomad crypto bridge loses $200 million in 'chaotic' hack, Aug 2022.

[14] S. Haber and W. S. Stornetta. How to time-stamp a digital document. In *Conference on the Theory and Application of Cryptography*, pages 437–455. Springer, 1990.

[15] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[16] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 42–54, 2006.

[17] P. Letouzey. A new extraction for coq. In *International Workshop on Types for Proofs and Programs*, pages 200–219. Springer, 2002.

[18] P. Letouzey. Extraction in coq: An overview. In *Conference on Computability in Europe*, pages 359–369. Springer, 2008.

[19] X. Li, Z. Shi, Q. Zhang, G. Wang, Y. Guan, and N. Han. Towards verifying ethereum smart contracts at intermediate language level. In *International Conference on Formal Engineering Methods*, pages 121–137. Springer, 2019.

[20] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[21] J. B. Nielsen and B. Spitters. Smart contract interactions in coq. In *International Symposium on Formal Methods*, pages 380–391. Springer, 2019.

[22] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[23] S. Palladino. The parity wallet hack explained, Jul 2017.

[24] G. A. Pierro and A. Amoordon. A tool to check the ownership of solana's smart contracts. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1197–1202. IEEE, 2022.

[25] M. Sozeau, A. Anand, S. Boulier, C. Cohen, Y. Forster, F. Kunze, G. Malecha, N. Tabareau, and T. Winterhalter. The metacoq project. *Journal of Automated Reasoning*, 2020.

[26] N. Szabo. Smart contracts. 1994. *Virtual School*, 1994.

[27] N. Szabo. Smart contracts: building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought,(16)*, 18(2), 1996.

[28] T. N. Tavu. *Automated Verification Techniques for Solana Smart Contracts*. PhD thesis, 2022.

[29] E. Team. Etherscan: The ethereum block explorer. *URL: https://etherscan. io*, 2017.

[30] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li. A survey of smart contract formal specification and verification. *ACM Computing Surveys (CSUR)*, 54(7):1–38, 2021.

[31] A. Yakovenko. Solana: A new architecture for a high performance blockchain v0. 8.13. *Whitepaper*, 2018.