SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

An extended compiler for safe TEAL smart contracts

Laurea Magistrale in Methematical Engineering - Ingegneria Matematica

Author: Francesco Albano

Advisor: Prof. Daniele Marazzina

Co-advisor: Prof. Andrea Bracciali

Academic year: 2021-2022

1. Introduction

Smart contracts are programs running on cryptocurrency blockchains. Their popularity arises from the possibility to perform financial transactions, such as payments and auctions, in a distributed environment with no need for any trusted third party. Unfortunately, the rise of these technologies has been marred by a series of costly bugs. Indeed recent attacks have pointed out the necessity of the design of formal verification techniques for smart contracts. This, however, requires rigorous semantic foundations and a formal characterisation of the expected security properties. In this framework, several studies have tried to understand deeply the models behind different blockchains. In [5] the first complete small-step semantics of EVM bytecode is presented, obtaining executable code that they successfully validate against the official Ethereum test suite. Similar path is followed by [6] regarding Bitcoin. They propose a symbolic verification theory for open script (i.e. smart contract), a verifier tool-kit, and illustrate examples of use on Bitcoin transactions. In [2] we find another innovative solution for this issue, where the authors develop a formal model for Algorand. Algorand [4] is a late-generation blockchain, pioneer in introducing important properties, like high-scalability and a no-forking consensus protocol based on Proof-of-Stake. Its smart contract layer (ASC1) aims at mitigating SC risks and adopts a non-Turing-complete programming model, some supporting atomic sets of transactions and user-defined assets. Algorand lacks of a mathematical model of contracts and transactions suitable for formal reasoning on their behaviour, and for the verification of their properties. Such a model is needed to develop techniques and tools to ensure that contracts are correct and secure. Moreover, even preliminary informal reasoning on non-trivial smart contracts can be challenging, as it may require to resort to experiments, or to direct inspect the platform source code. The formal model is high-level enough to simplify the design of Algorand smart contracts and enable formal reasoning about their security properties. The model embed Algorand contracts in a simple declarative language and provides a basis for the automatic verification of ASC1. The authors define also a new declarative language based on the formalised model. This work proposes a prototype tool that compiles smart contracts, written in our formal declarative language, into executable TEAL code. After this idea, it is called SecTeal, indeed Secure Teal. In the actual form, this application does not allow the implementation of every type of stateless smart contract and also misses of a verification procedure.

Our contribution to this work can be shared in two main phases. First, we complete the syntax of the new declarative language in order to make SecTeal able to compile stateless smart contract for every type of transactions according with the formal model [2] and Algorand. Then we update the syntax in order to better integrate with the new features of ASC1. To achieve this, we have enhanced the previous version of the compiler and we have substantially modified the architecture of the syntax adding three other transaction types. Also, we give some formal definitions for the concept of "Safe", "Correct" and "Wrong" in this smart contract setting. In the second part of the work then, we effectively extend the compiler implementing a verification algorithm (SecFun) which checks if the contract is safe and correct. In case an error is found, this compiler corrects the contract by means of a direct interaction with the user. The method for checking the correctness is based on the creation of various Gold Standard tables, one for each transaction type. These define the parameters which characterise a particular type of smart contract. They specify which parameters are necessary in the SC and for each parameter there is a list of associated value and type. This whole new system ensures that contracts are safe and correct. In the last part of our work, we detailed the main passage of Secfun algorithm and we performed various test with a wide range variety of types of stateless smart contracts in order to validate the tool. Eventually, the developed tool is able to compile Teal stateless smart contracts in an extreme secure way and it is ready to be deployed on Algorand blockchain.

2. Formal model ASC

The formal model [2] is faithful to the previous ASC1 implementation. It is developed so that it has a high-level architecture and it is simple to be understood. The model has to cover the most commonly used features and procedures of Algorand and support the specification and verification of non-trivial smart contracts. As TEAL code has the purpose of accepting or rejecting transactions, the model by [2] evaluates expressions as true or false. They also formalise different transaction types by classifing them merely on the base of their final goal.
The cornerstones of the model are the definition of Account and Transaction. An account is a deposit of one or more crypto-assets, as it is also in Algorand. It is modeled as terms $x[\sigma]$, where x is an address uniquely identifying the account, and σ is the balance. The transaction instead shows some differences with respect to the Algorand model. In [2], transactions are modelled as records with the structure in Table 1. For every possible type of transaction defined in the formal model, all the parameters associated to them are listed.

Type	Parameter
pay	snd,rcv, val, asst
close	snd,rcv, asst
gen	snd,rcv, val
optin	snd, asst
burn	asst
rvk	snd,rcv, val, asst
frz	snd, asst
unfrz	snd, asst
delegate	snd, asst, rcv

Table 1: Transaction types according to [2]

Observing Table 1, it is worth noticing that *pay* and *close* are now two separate transactions, while in Algorand they belong to the same transaction type and they are discerned only by the parameters. This follows the idea to create a clearer, simpler and safer model. In Algorand, the actual behaviour of a transaction may depend on both its type and other conditions, e.g., which optional fields are set. For instance, pay transactions may also close accounts if the *CloseRemainderTo* field is set [1].

The authors of [2] model the evolution of Algorand blockchain as a labelled transaction system. The fundamental concept is the state of the chain, indicated with Γ. At any time the transaction state is:

$$x_1[\sigma_1] \mid ... \mid x_n[\sigma_n] \mid r \mid \mathcal{T}_{lv} \mid f_{asst} \mid f_{lx} \mid f_{frz} \tag{1}$$

where all addresses x_i are distinct and "|" is commutative and associative. All the elements presents in Equation 1 describe the actual state

of the chain. For every operation on the chain, namely for every change of the parameters, the whole state of the chain changes. In the model we find $\Gamma \xrightarrow{t}_1 \Gamma'$ with the the following meaning: *if* the transaction t is performed in blockchain state Γ, then the blockchain evolves to state Γ'. The authors specify the transition relation $\rightarrow_1$ through a set of inference rules: each rule describes the effect of a transaction t in the state Γ of Equation 1. Using the same architecture, the model defines also the evolution for Atomic transaction. Consider $\mathcal{T} = t_1 \dots t_n$ sequence of transactions for Γ and all the transaction T_i to be performed in sequence, then we write the atomic execution of $\mathcal{T}$ in Γ as: $\Gamma \xrightarrow{\mathcal{T}} \Gamma_n$.
As explained before this model is developed to be able to write more secure smart contract. The model covers also the design of SCs and establishes their security. To do this, the authors assume the presence of an attacker model and consider a set of adversary strategies which can delay the transactions of the user. In the formal model, we take inspiration from the latter to abstract TEAL bytecode scripts. When firing transaction involving scripts, users can specify a sequence of arguments. Accordingly, the script language includes operators to know the number of arguments, and access them. Further, scripts include cryptographic operators to compute hashes and verify signatures. The script $\mathbf{e}$ evaluation function $[\![\mathbf{e}]\!]^{\mathcal{W}}_{\mathcal{T},i}$ evaluates $\mathbf{e}$ using 3 parameters: a sequence of arguments W, a sequence of transactions T forming an atomic group, and the index $i < |T|$ of the transaction containing $\mathbf{e}$. Lastly, we mention that the formal model satisfies also the three fundamental property of Algorand smart contracts, namely No double spending, Value preservation and Determinism.

3. Secteal

A further contribution in [2] is SecTeal [3]. In its actual implementation, this tool compiles into Teal smart contract written in the declarative language. This tool is born with the idea of testing the model and helping the user to implement smart contracts in more suitable ways. Since the style of the new programming language is similar to PyTeal, the transition from it to Teal is very intuitive. For instance, translating a transaction of the form:

$$type : close, snd : x, rcv : y, asst : \tau \qquad (2)$$

results in the concrete transaction in Table 2 and Table 3 (where we omit the irrelevant fields). In Equation 2 we implement a close transaction written in the declarative language based on the formal model. In particular the sender is x and the receiver is y (e.g. x, y represent two different address). The result is the concrete transaction reported respectively in Table 2 and Table 3. These show respectively a pay transaction (τ: Algos) and an asset transfer transaction (τ: Generic Asset) for Algorand.

Type	pay
snd	x
close	y
amt	0

Table 2: Pay, τ: Algo

Type	axfer
snd	x
asnd	x
aclose	y
xaid	τ
aamt	0

Table 3: Asset Transfer, τ: Asset

A transaction is indicated by *tx* and compiling the script *tx(n).f* depends on the field *f* that could be *type, snd, rcv* .... The compilation of *tx(n).snd* is in Teal equal to "*gntx n Sender*".

The principal components of SecTeal are The interface, the Parser and the Compiler. The second one is based on ANTLR [7]. In the Figure 1, a scheme of SecTeal is represented in the actual form.

4. Extended Compiler

SecTeal is a first building block toward a comprehensive IDE for the design, verification, and deployment of contracts on Algorand. To make sure that the compiler fullfills all these proprierties, we need to implement some support functions that check if the safeness of the contract is respected and adjust it in case of errors. Moreover, an extension of the actual syntax is needed

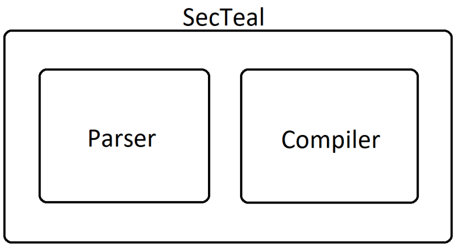

Figure 1: SecTeal

to design smart contract for every transaction type. At the end, to deployed the contract directly on Algorand testnet, also the type parameter must be translate in the corresponding Algorand transaction type. Our contribution follows this path implementing various extensions of this tool. After that, the extended SecTeal presents the architecture in Figure 2. Our work

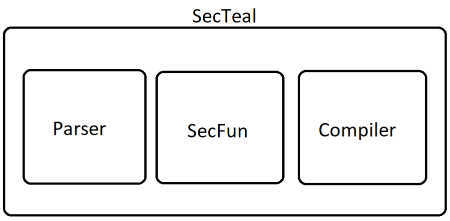

Figure 2: Extended SecTeal

modifies the structure of the compiler in two way: first, completing the compiler already implemented to be able to compile stateless smart contract for every type of transaction and to be updated with the new features of ASC1. Then we implement a verification algorithm to check that the contract is safe. This part allow Secteal to interact with the user and correct the input structure in order to have as output of the tool a Teal contract ready to be deployed without any operational risk for the Algorand user. At the beginning we define that *a contract is safe if is impossible using that contract to validate a transaction that could belong to a different type.* This definition can be adopted for SC that at least contains the type transaction parameter. This can be one of the ones listed in the Table 1 plus three others like *pay asset, close asset*, and *rekye*.

For each type of transaction, it is established a GS-Table (Gold-Standard), which describes precisely the required parameters, with associated type and value, needed to correctly implement a safe stateless smart contract once its intended type has been identified. A GS-Table for a *pay transaction* is the follow:

Param	Category	Type	Value
TYPE	$\top$	String	
SND	$\top$	Address	
RCV	$\top$	Address	
VAL	$\top$	Integer	
CRCV	**	Address	Null Address
REKEY	**	Address	Null Address
FV	$\perp$	Integer	
LV	$\perp$	Integer	
LX	$\perp$	Integer	
NOTE	$\perp$	String	
FEE	$\perp$	Integer	

Table 4: PAY

The parameters that belong to the Necessary category (**) must be present into the contract with the value indicated on the last column. If this condition is not respected then the contract in not safe. In this case Secteal reports the error to the user and adjusts the contract. In the others two categories, Required ($\top$) and Default ($\perp$), we find the parameters that belongs to this transaction type but are not necessary in determining the safeness of the contract. If a Required parameter value is not specified by the contract, then a Warning message is reported to the user. Instead no actions are taken if the condition of a Default parameter is not specified. For what concern Value or Type Error, an error message is reported independently from the parameter. The contract moves from the Parser to the Secure Function (SecFun) with the input P. The structure of it is

$$[op, exp, exp] \;\; or \;\; [op, exp]$$

where op are the canonical operator defined in the formal model. Recursively exp could have the same structure of P or it is a combination of variables and constants, belonging to the syntax of the declarative language, joined by operators. In this sense, P indeed has the same architecture of a Tree. For this reason we chose the top-down parsing strategy to parse P. If at a certain node the op is equal to AND or OR then there are

two links, otherwise it is an leaf node.
The parameters are collected in a set of T-tables

$$\mathcal{TT} := \{TT_1, TT_2, ...TT_k\}.$$

A T-table consists of a set of requirements of transaction parameters which must hold at the same time in order for the contract to successfully terminate and accept the transaction. T-tables are built according with the structure of the SecTeal logic expressions and its structure. For example the expression

$$[and, A, [or, C, B]]$$

will return the T-tables

$$[A_l, C_{lr}] \quad [A_l, B_{ll}]$$

where subsripts indicate associated condition in P, according to standard tree enumeration. Instead uppercase are the parameters. We indicate with $match(TT_j, GS_i)$ the function which compares the elements of TT_j and GS_i. Following our definition, TT_j and GS_i *suitable match if every parameter in the table* TT_j *is also in* GS_i *and they have the same type and value.* The outcome of each single $match(TT_j, GS_i)$ is said to be Positive, when the two tables *suitably match*. The output is negative if TT_j and GS_i do not *suitably match* (e.g. some parameter in TT_j does not respect condition in GS_i and/or some parameter in TT_j is not present in GS_i). We implement also a method which allows SecTeal to work in the same way explained above also when the type of the stateless smart contract is not specified. In that case the tool displays to the user all the possible type of contracts that *suitably match* the other parameters and it asks to choose one. In case the user does not selects any of these options, the process run without any other possible checks.
Our work presents also a series of tests of the extended compiler for different types of stateless smart contracts. The tests are performed on Python using the VS Code editor. All codes are accessible on Github.
Now, we report an example step by step to show how our extended compiler works with a stateless smart contract for a PAY transaction which does not respect all the conditions reported on Table 4. More precisely, in the following contract the *crcv* (close account) parameter is not specified. First, SecTeal notices that the contract is not safe and reports the error to the user (line 1). If the user decides to fix it (line 2), then the result is a Teal smart contract where the previously missing condition (CloseReminderTo) is now fulfilled (lines 30-33). Notice that in line 34 there is the proper operator in the correct place.

```
1  ERROR :  crcv non present
2  Do you want to add this condition? (Y/N)
       :  Y
3
4  #OUTPUT: TEAL  SMART CONTRACT - ADJUSTED
5
6  PAY                     <<<
7
8  #pragma version 2
9  gtxn 0 Type
10 int pay
11 ==
12 gtxn 0 Fee
13 int 1000
14 <
15 gtxn 0 Receiver
16 addr
       ZZAF5ARA4MEC5PVDOP64JM5O5MQST63Q2KOY
17     2FLYFLXXD3PFSNJJBYAFZM
18 ==
19 gtxn 0 Amount
20 int 100
21 ==
22 gtxn 0 Sender
23 addr 5ARA4MEC5PVDOP64JM5O5MQST63Q2KOY2FL
24      YFLXXD3JFSNJJCYAFZTSERT
25 ==
26 gtxn 0 rekey-to
27 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
28      AAAAAAAAAAAAAAAAAAAAAAY5HFKQ
29 ==
30 gtxn 0 CloseReminderTo
31 addr AAAAAAAAAAAAAAAAAAAAAAAAAA
32      AAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQ
33 ==
34 &&
35 &&
36 &&
37 &&
38 &&
39 &&
```

Otherwise, in the following script is reported the case in which the user decides no to adjust the contract. In this case the tool stops the process because the contract is not safe and returns an error message to the user.

```
ERROR :  crcv non present
Do you want to add this condition ? (Y/N
    ) : N

CLOSE                    <<<

#pragma version 2
['Contract IS NOT SAFE :  Condition on
    crcv non present']
```

5. Conclusions

In this work we have presented the idea and the realisation of an extended compiler suitable for smart contract on Algorand. This thesis builds on the work of [2]. The main contribution of this work is the implementation of several new features of the tool Sectal [3]. To do this, we had to introduce notation and formalize different concepts already introduced in [2].

We provided the compiler with several important extensions. First of all, we have updated the syntax and the architecture of the declarative language with respect to the new version of ASC1 always taking into account stateless smart contracts. We also had to reshape the architecture of this declarative language creating two new transactions with the aim of making SecTeal more user-friendly. For each types, we have associates a Gold Standard-Table describing in details the required parameters for the particular transaction type. Furthermore, we explicit the parameters that are used for testing the condition of safeness and therefore need to be added while writing the contract. The core of our work is the implementation of the Secure function algorithm.

As final result, Secteal is able to translate contracts for all types of transaction listed in [2] compiling it into safe Teal smart contracts. Our compiler checks the contract written by the user, adjusts it in case it is not Safe or if there are some other errors. In this way the output of our tool is a correct and safe stateless smart contract ready to be deployed on Algorand testnet. In the last part of our work we detailed the most important procedure of the verification algorithm (SecFun). The validity of the problem has been tested through the implementation of various experiments with some type of transactions.

The extended tool for Algorand smart contracts presented in this work can be expanded depending on the evolution of Algorand itself and its original model. Also, a similar analysis could be done for the declarative languages for stateful Algorand smart contracts. This would require also the update of the model underlying it. It would be necessary to embed the key-value store in contract accounts and extend the script language with key-value store updates. Moreover, our compiler works only for single transactions so it would be useful to complete it in order to allow the implementation of smart contract for atomic transactions.

References

[1] Algorand. Opcodes, 2021.

[2] Massimo Bartoletti, Andrea Bracciali, Cristian Lepore, Alceste Scalas, and Roberto Zunino. A formal model of algorand smart contracts. In *International Conference on Financial Cryptography and Data Security*, pages 93–114. Springer, 2021.

[3] Massimo Bartoletti, Andrea Bracciali, Cristian Lepore, Alceste Scalas, and Roberto Zunino. Secteal, 2021.

[4] Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theoretical Computer Science*, 777:155–183, 2019.

[5] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *International Conference on Principles of Security and Trust*, pages 243–269. Springer, 2018.

[6] Rick Klomp and Andrea Bracciali. On symbolic verification of bitcoin's script language. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 38–56. Springer, 2018.

[7] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.