



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

An extended compiler for safe TEAL smart contracts

TESI DI LAUREA MAGISTRALE IN
MATHEMATICAL ENGINEERING - INGEGNERIA MATEMATICA

Author: **Francesco Albano**

Student ID: 968874

Advisor: Prof. Daniele Marazzina

Co-advisors: Prof. Andrea Bracciali

Academic Year: 2021-2022

Abstract

Smart contracts (SCs) are powerful applications used by the most efficient blockchains. For this reason it is crucial to be able to write safe and correct smart contracts in a simple way. The Algorand blockchain writes SCs in the programming language Teal. Since Teal is a low level language difficult to be used, many errors can occur. Several researches have tried to solve this problem. Among all of them, an innovative solution is found in the alternative model in "A formal model for Algorand smart contract" [16]. The authors exploit this framework to prove fundamental properties of the Algorand blockchain, and to establish the security of some archetypal SCs. A declarative language for the implementation of smart contract is defined, based on the formal model [16]. This work includes also a tool called Secteal. It provides a compiler whose main function is translating the declarative language into Teal. The aim of our work is enhancing Secteal. Our new compiler is able to handle even complex SCs for every type of transaction. It checks the correctness of the smart contract provided by the user, ensuring a better safety of the whole process. Safeness is guaranteed by validation of necessary conditions for every type of stateless smart contract. The output of Secteal is a Teal contract ready to be deployed on Algorand. To achieve these goals we have implemented various Secure functions that guide the user to correct the SC in case of errors or missing safety conditions.

Keywords: Blockchain, Algorand, Smart Contract, Teal, Secteal, Safe smart contract, Compiler, Secure function.

Sommario

I contratti intelligenti (SC) sono applicazioni che hanno portato grandi miglioramenti alle più efficienti Blockchain. Per questa ragione è cruciale essere in grado di scrivere in modo semplice contratti intelligenti sicuri e corretti. Algorand utilizza il linguaggio di programmazione Teal per implementare i suoi contratti. Esso è un linguaggio di basso livello, difficile da utilizzare. Questo porta ad accrescere il numero di errori. Diversi studi sono stati presentati su questo tema per diverse tipologie di Blockchain. Una soluzione innovativa per Algorand è stata proposta in "A formal model for Algorand smart contract" [16]. Gli autori presentano un modello per provare le proprietà fondamentali di Algorand e per definire un metodo per la costruzione di contratti intelligenti sicuri. Nel loro lavoro gli autori introducono un linguaggio di programmazione basato su tale modello. Il lavoro include un'applicazione chiamata SecTeal. Questa si basa principalmente su un compilatore che ha come funzione principale quella di tradurre il nuovo linguaggio in Teal. L'obiettivo del nostro lavoro è quello di completare SecTeal. Il nostro compilatore sarà capace di gestire anche contratti intelligenti più articolati per ogni tipo di transazione controllando la correttezza del contratto in ingresso a SecTeal. Questo assicurerà maggior sicurezza all'intero processo. La sicurezza è garantita dalla presenza di condizioni necessarie per ogni tipo di SC. Il risultato finale della nostra applicazione è un contratto intelligente scritto in Teal pronto per essere distribuito su Algorand. Per raggiungere questi obiettivi verranno implementate diverse funzioni di sicurezza le quali analizzeranno e aggiusteranno il contratto intelligente in caso di errori o mancanza di condizioni necessarie per la sicurezza del contratto.

Parole chiave: Blockchain, Algorand, Contratto Intelligente, Teal, Secteal, Sicurezza, Compilatore, Funzioni di sicurezza.

Contents

Abstract	i
Sommario	iii
Contents	v
1 Introduction	1
2 Literature review	3
3 Background	5
3.1 Blockchain Structure	6
3.2 Consensus mechanism	7
3.3 Cryptography	12
3.4 Smart Contract	13
4 Algorand	15
4.1 Trilemma	15
4.2 Pure Proof of Stake (PPoS)	16
4.2.1 Security	18
4.2.2 Scalability	18
4.2.3 Decentralization	19
4.2.4 Non-Forkable	19
4.3 Principal Components	19
4.4 Algorand Smart Contract	21
4.5 Algorand Smart Signatures	23
4.6 Contract account	24
4.7 Delegated approval	25
4.8 Conclusion	25

5	Teal	27
5.1	Introduction Teal	27
5.2	Execution Modes	31
6	Formal model for ASC	35
6.1	Blockchain states	37
6.2	Executing single transactions	38
6.3	Atomic transaction	40
6.4	Executing smart contracts	40
6.5	Authorizing transactions, and user-blockchain interaction	41
6.6	Fundamental properties of ASC1	42
6.7	Designing secure smart contracts in Algorand	43
6.8	Conclusion	46
7	Secteal	49
7.1	From the formal model to concrete Algorand	49
7.2	SecTeal	50
7.3	Extended SecTeal	51
7.4	SecFun	52
7.4.1	Algorithm	56
7.4.2	Parsing procedure	57
7.4.3	Matching procedure	60
7.4.4	Correction procedure	61
7.4.5	Table without Type parameter	66
7.5	Conclusion	67
8	Test	69
8.0.1	Close account transaction smart contract	69
8.0.2	Pay account transaction smart contract	71
8.0.3	Pay Asset transaction smart contract	73
8.0.4	Rekey account transaction smart contract	75
8.0.5	Close without CloseRemainderTo	76
8.0.6	Close with parameter type error	79
8.0.7	Value type error	82
8.0.8	Necessary parameter value error	85
8.0.9	Parameter not in <i>GS-Table</i>	86
8.0.10	Smart contract without Type parameter	88
8.0.11	Contract with OR operator	90

8.0.12 Hash Time Lock Contract (HTLC)	93
8.0.13 Periodical payment	94
8.0.14 Oracle	95
9 Conclusions and future developments	99
Bibliography	101
A Appendix A	107
B Appendix B	111
B.1 Type Parameter	111
B.2 <i>GS</i> -Table	111
List of Figures	119
List of Tables	121

1 | Introduction

Transactions between parties in current systems are usually conducted in a centralised fashion, which requires the involvement of a trusted third party (e.g., a bank). However, this could result in security issues (e.g., single point of failure) and high transaction fees. Blockchain technology has emerged to tackle these issues by allowing untrusted entities to interact with each other in a distributed manner without the involvement of a trusted third party. Blockchain is a distributed database that records all transactions that have ever occurred in a network. It was originally introduced for Bitcoin (a peer-to-peer digital payment system), but then evolved to be used for developing a wide range of decentralised deployments. An appealing application which can be deployed on top of blockchain is Smart Contracts (SCs). They are executable codes that run on the blockchain to facilitate, execute and enforce the terms of an agreement between untrusted parties. It can be thought of as a system that releases digital assets to all or some of the involved parties once the pre-defined rules have been met. Given the wide applicability of SCs and their significant lower cost, several blockchain are nowadays employing them. Our work focuses on the blockchain *Algorand*. It was founded by Silvio Micali, with a vision to democratise finance and deliver on the blockchain promise. Unfortunately most of blockchain are incapable of satisfying simultaneously the following properties: security, scalability and decentralisation. This problem is known as *Trilemma*. A solution is found introducing a new Pure Proof of Stake (PPoS) consensus protocol, which is the one used in Algorand. Algorand Smart Contracts (ASC1) are separated into two main categories, smart contracts, and smart signatures. These types are also referred as stateful and stateless contracts, respectively. The type of contract that is written will determine when and how the logic of the program is evaluated. Both types of contracts are written in the Transaction Execution Approval Language (TEAL), which is an assembly-like language that is interpreted by the Algorand Virtual Machine (AVM) running within an Algorand node. Since Teal is a low level language and difficult to be used, several errors can occur. In the literature, many authors have tried to overcome this issue for various blockchains like Bitcoin and Ethereum. In this work we build on the approach introduced in [16]. The authors of [16] present an alternative model to the one underling

the Algorand architecture. They prove that the fundamental properties of Algorand still hold in this new framework. Moreover it is high-level enough to simplify the design of Algorand smart contracts and enable formal reasoning about their security properties providing also a basis for the automatic verification of ASC1s. Based on this model our contribution is to complete a prototype tool which compiles smart contracts (written in our formal declarative language introduced by [16]) into executable TEAL code. That tool is called *Secteal*. In its current form, *Secteal* supports experimentation with the formal model, and it is provided with a series of examples. Furthermore it allows private user to compile their own contracts.

After a deep study and analysis of the model [16], we managed to understand the key points in the Algorand blockchain underlying model for ensuring the necessary security. 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 get Secteal able to compile stateless smart contract for every type of transactions according with the formal model [16] and Algorand [8]. Then we update the syntax in order to consider some of the new features of ASC1 [4]. To do that we have modified the already implemented compiler and also the architecture of the syntax adding three other transaction types. Also, we give some definition in order to well define what a Safe, Correct, Wrong contract is. In a second moment we work on the extension of the compiler implementing a verification algorithm (SecFun) that checks if the contract is safe and correct. In case of error our extended tool is able to adjust the contract through the direct interaction with the user. This method is based on the creation of various Gold Standard tables, one for each transaction type, that define the parameters that characterise that type. These tables specify what parameters are necessary to be included into the contract, with associated value and type, in order to have a safe and correct contract. In the last part of our work we detailed the main passage of Secfun algorithm and various tests are performed with some types of stateless smart contracts in order to validate the extended SecTeal. All of these procedures are crucial in order to have a tool able to compile Teal smart contracts secure and ready to be deployed on Algorand blockchain.

The remainder of this thesis is organised as follows. In chapter 2 a survey of relevant literature is reported. In chapter 3 we detail some basic concepts about blockchains and smart contracts. In chapter 4 and chapter 5 we give an overview of what Algorand is and explain the main features of Teal. We highlight the architecture of the formal model for Algorand smart contract in chapter 6. Then in chapter 7 we detail our contribution to this work. Chapter 8 summarises all the experiments performed and finally chapter 9 presents the conclusions of the work.

2 | Literature review

This work is part of a wider research line on formal modelling of blockchain based on contracts. 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 expansion of these technologies has been marred by a series of costly bugs. These issues arise in several blockchain, from Bitcoin ([21],[38],[46]) to Ethereum ([20], [42], [32], [33], [34] [35]) and from Tezos [19] to Cardano [22]. They all tried to propose different methodologies to develop a more clear and secure SC. The authors of [38] present a symbolic verification theory for open script (i.e. smart contract), a verifier tool-kit, and illustrate examples of use on Bitcoin transactions. Two year later, [46] presents the first mechanized formal model of Bitcoin transactions and blockchain data structures including the formalization of the blockchain validation procedures. Their formal model includes regular and coinbase transactions, segregated witnesses, relative and absolute locktime. In this work, Bitcoin Script language expressions together with a denoted semantics, transaction fees and block rewards are included. They formally specify the details of validity checks performed when adding new blocks to the blockchain.

For the Ethereum blockchain, [32] overviews the existing approaches taken towards formal verification of Ethereum smart contracts and discusses EtherTrust, the first sound static analysis tool for EVM bytecode. Specifically, they present an illustrative presentation of the small-step semantics proposed by [33] with deep focus on the semantics of the bytecode instructions that allow for the initiation of internal transactions. The subtleties in the semantics of these transactions have shown to form an integral part of the attack surface in the context of Ethereum smart contracts. A review of an abstraction based on Horn clauses [27] for soundly over-approximating the small-step executions of Ethereum bytecode [31]. A demonstration of how relevant security properties can be over-approximated and automatically verified using the static analyzer EtherTrust [31] by the example of the single-entrancy property defined in [33]. Another work following a similar path is [32]. The authors introduce the first complete small-step semantics for EVM bytecode. A formalization of a large fragment of the semantics, which can serve as

a foundation for verification techniques based on encoding into their declarative language [20] as well as machine-checked proofs for other analysis techniques. They explicit the first formal definition of crucial security properties for smart contracts, for which we devise a dedicated proof technique, atomicity, and independence from miner controlled parameters. Interestingly enough, the formalization of these properties requires hyper-properties, while existing static analysis techniques for smart contracts rely on reachability properties and syntactic conditions. In this specific framework, the authors of [16] present a formal model of stateless smart contract providing a solid theoretical foundation to Algorand smart contracts. Such a model formalises Algorand accounts, transactions and smart contracts. In order to validate the model, all fundamental properties of Algorand state machine - like no double spending, determinism and value preservation - are formalised and proved. The analysis of Algorand contract design patterns is extremely important. It is based on several non-trivial contracts, covering both standard use cases, and new ones. Quite surprisingly, they show that stateless contracts are expressive enough to encode arbitrary finite state machines. These formal models are a necessary prerequisite to rigorously reason on the security of smart contracts, and they are the basis for automatic verification. Besides modelling the behaviour of transactions, in [16] the authors propose a model of attackers: this enables us to prove properties of smart contracts in the presence of adversaries, in the spirit of longstanding research in the cryptography area ([13], [14], [15], [18], [26], [39], [40]). Furthermore, they present a prototype tool called Secteal which compiles smart contracts (written in their formal declarative language) into executable TEAL code. This tool, based on the model, works only for stateless SCs. The idea is to create a tool able to help the user to write not naive smart contracts. They must be secure and correct in order to be ready deployed on Algorand testnet [3]. Our contribution goes in this direction, implementing an extended SecTeal that not only compile the smart contract but also checks if it is safe and correct. Our tool will be able to interact with the user and adjust the contract in base to the different type of stateless smart contracts and the various errors found. In order to do that we have modified and completed the already developed syntax of the declarative language and we have implemented a verification algorithm.

3 | Background

Distributed Ledger Technologies (DLT) are distributed ledger-based systems, i.e. systems in which all nodes in a network have the same copy of a database that can be read and modified independently of individual nodes.

DLT and the traditional Distributed Database differentiate for the way they share information within the network. In the latter indeed, every node own just a copy of the original document in the central node. Therefore every operation in the network has to pass through the central coordinator. In DLT instead, information is shared between every node. In this sense, every node is independent and can lead operation by his own with other nodes. These operations are regulated by means of some *consensus algorithm*. As self explained by their name, these algorithms make it possible to reach consensus between the various versions of the register, despite the fact that they are updated independently of the network participants. In addition to consensus algorithms, to maintain the security and immutability of the ledger, Distributed Ledger and Blockchain also make extensive use of cryptography. The type of network, the consensus mechanism and the structure of the register are the main three parameter that define the methods of updates of the register in the network. Depending on the type of network, we can distinguish between two systems:

- *Permissioned* - networks in which to access it is necessary to register and identify oneself and therefore be authorized by a central body or by the network itself;
- *Permissionless* - networks where anyone can access without permission.

In *Permissioned* systems the consensus mechanism is straightforward: whenever a node proposes the addition of a transaction, its validity is verified and the decision is made on the based of the result of a majority vote rule. On the other hand, in *Permissionless* systems consent mechanisms are more complex to prevent a malicious subject from creating numerous fictitious identities and influencing the process of editing the register.

Blockchains are a particular type of DLT. They have some extra features, namely the features Transfer and Asset.

From its early beginning in 2010, *Blockchain technology* has evolved to become a manage-

ment solution for all types of global industries. Nowadays, *Blockchain* technology provides transparency for a wide variety of industries like the food supply chain, securing health-care data and innovating gaming. Overall, they are the main responsible for the change in how we handle data and attribute ownership on a large scale. As explained above, *Blockchain* is a distributed database solution that maintains a continuously growing list of data records that are confirmed by the nodes participating in it. The data is recorded in a public ledger, including information of every transaction ever completed shared equally among all the nodes. This attribute makes the system more transparent than centralized transactions involving a third party. In addition, the nodes in *Blockchain* are all anonymous, which makes it more secure for other nodes to confirm the transactions. Moreover it helps reduce security risks, stamp out fraud and bring transparency in a scalable way. However, even though *Blockchain* seems to be a suitable solution for conducting transactions with cryptocurrencies, it has still some technical challenges and limitations that need to be studied and addressed. High integrity of transactions and security, as well as privacy of nodes, are needed to prevent attacks and attempts to disturb transactions in *Blockchain*. In addition, confirming transactions requires computational power.

3.1. Blockchain Structure

Blocks are data structures within the blockchain database, where transaction data in a cryptocurrency blockchain are permanently recorded. A *block* records some or all of the most recent transactions not yet validated by the network. Once the data are validated, the *block* is closed. Then, another one is created for new transactions to be entered into and validated. It is thus a permanent store of records that, once written, cannot be altered or removed. Each *block* has three basic elements: data, nonce and hash. Data stores information, nonce is instead a number used only once that is a 32-bit whole number that's randomly generated when a block is created, which then generates a block header hash. The hash is a 256-bit number permanently attached to the nonce. It must start with a huge number of zeroes (i.e., be extremely small). When the first block of a chain is created, a nonce generates the cryptographic hash. The data in the block is considered signed and forever bind to the nonce and hash unless it is mined.

There are many pieces of information included within a block, but it doesn't occupy a large amount of storage space. Blocks generally include these elements, but it might vary between different types:

- *Information number*: A number containing specific values that identify that block as part of a particular cryptocurrency's network.

- *Blocksize*: Sets the size limit on the block so that only a specific amount of information can be written in it.
- *Block header*: Contains information about the block.
- *Transaction counter*: A number that represents how many transactions are stored in the block.
- *Transactions*: A list of all of the transactions within a block.

The transaction element is the largest as it contains the most of information. The second largest in storage size is the block header, which includes these sub-elements:

- *Version*: The cryptocurrency version being used.
- *Previous block hash*: Contains a hash (encrypted number) of the previous block's header.
- *Hash Merkle root*: Hash of transactions in the Merkle tree of the current block.
- *Time*: A timestamp to place the block in the blockchain.
- *Bits*: The difficulty rating of the target hash, signifying the difficulty in solving the nonce.
- *Nonce*: The encrypted number that a miner must solve to verify the block and close it.

One 32-bit number in the header is called a nonce, the mining program uses random numbers to guess the nonce in the hash. When a nonce is verified, the hash is solved when the nonce, or a number less than it, is guessed. Then, the network closes that block, generates a new one with a header, and the process repeats. Different mechanisms are used to reach a consensus; the most popular for cryptocurrency is proof-of-work (PoW) [37], with proof-of-stake (PoS) [28] becoming more so because of the reduced energy consumption compared to PoW.

3.2. Consensus mechanism

As already mentioned, in any centralized system, a central administrator has the authority to maintain and update the database. Public blockchains that operate as decentralized, self-regulating systems work on a global scale without any single authority. They involve contributions from hundreds of thousands of participants who work on verification and authentication of transactions occurring in the blockchain and on the block mining ac-

tivities. In such a dynamically changing status of the blockchain, these publicly shared ledgers need an efficient, fair, real-time, functional and secure mechanism to ensure that all the transactions occurring on the network are safe and all participants agree on a consensus on the status of the ledger. This all-important task is performed by the consensus mechanism, which is a set of rules that decides on the legitimacy of contributions made by the various participants (i.e., nodes or transactors) of the blockchain.

There are different kinds of consensus mechanism algorithms, each of which works on different principles. The *proof of work (PoW)* is a common consensus algorithm used by the most popular cryptocurrency networks like bitcoin and litecoin. It requires a participant node to prove that the work done and submitted by them qualifies them to receive the right to add new transactions to the blockchain. However, this whole mining mechanism of bitcoin involves high energy consumption and a long processing time. The *proof of stake (PoS)* is another common consensus algorithm that evolved as a low-cost, low-energy consuming alternative to the PoW algorithm. It involves the allocation of responsibility in maintaining the public ledger to a participant node in proportion to the number of virtual currency tokens held by it. However, this comes with the drawback that it incentivizes cryptocurrency hoarding instead of spending. While PoW and PoS are by far the most prevalent in the blockchain space, there are other consensus algorithms like *Proof of Capacity (PoC)* [47] which allow sharing of memory space of the contributing nodes on the blockchain network. The more memory or hard disk space a node has, the more rights it is granted for maintaining the public ledger. *Proof of Activity (PoA)* [50], used on the Decred blockchain, is a hybrid that makes use of aspects of both PoW and PoS. *Proof of Burn (PoB)* [36] is another that requires transactors to send small amounts of cryptocurrency to inaccessible wallet addresses, in effect burning them out of existence. Another, called *Proof of History (PoH)* [49], developed by the Solana Project and similar to *Proof of Elapsed Time (PoET)* [25], encodes the passage of time itself cryptographically to achieve consensus without expending many resources.

Proof of work

PoW describes a system that requires a not-insignificant but feasible amount of effort in order to deter frivolous or malicious uses of computing power, such as sending spam emails or launching denial of service attacks. The concept was subsequently adapted to securing digital money by Hal Finney in 2004 through the idea of reusable proof of work using the SHA-256 hashing algorithm [44]. Following its introduction in 2009, Bitcoin became the first widely adopted application of Finney's PoW idea [45]. This explanation focuses on the way *proof of work* works in the Bitcoin network. Bitcoin is a digital

currency that is underpinned by a kind of distributed ledger containing a record of all bitcoin transactions. In order to prevent tampering, the ledger is public or distributed. An altered version would quickly be rejected by other users. *POWs* detect tampering through hashes. Given set of data through a hash function (bitcoin uses SHA-256), it will only ever generate one hash. Due to the avalanche effect, however, even a tiny change to any portion of the original data will result in a totally unrecognisable hash. Whatever the size of the original data set, the hash generated by a given function will have the same length. The hash is a one-way function: it cannot be used to obtain the original data, only to check that the data that generated the hash matches the original data. Generating just any hash for a set of bitcoin transactions would be trivial for a modern computer, so in order to turn the process into work, the bitcoin network sets a certain level of difficulty. This setting is adjusted so that a new block is mined, added to the blockchain by generating a valid hash, approximately every 10 minutes. Setting difficulty is accomplished by establishing a "target" for the hash: the lower the target, the smaller the set of valid hashes, and the harder it is to generate one. In practice, this means a hash that starts with a very long string of zeros.

Since a given set of data can only generate one hash, is interesting to understand how do miners make sure they generate a hash below the target. They alter the nonce. Once a valid hash is found, it is broadcast to the network, and the block is added to the blockchain. Mining is a competitive process, but it is more of a lottery than a race. On average, someone will generate acceptable proof of work every ten minutes, but who it will be is anyone's guess. Miners pool together to increase their chances of mining blocks, which generates transaction fees and, for a limited time, a reward of newly-created bitcoins. *Pow* makes it extremely difficult to alter any aspect of the blockchain, since such an alteration would require re-mining all subsequent blocks. It also makes it difficult for a user or pool of users to monopolise the network's computing power, since the machinery and power required to complete the hash functions are expensive. Miners earn coins by verifying transactions and blocks. However, they pay their operating expenses like electricity and rent with fiat currency. What really happens then is that miners are exchanging energy for cryptocurrency. The amount of energy required to mine proof-of-work cryptocurrency profoundly affects the market dynamics of pricing and profitability. There are also environmental aspects to consider since PoW mining uses as much energy as a small country.

Proof-of-stake

PoS is a cryptocurrency consensus mechanism for processing transactions and creating new blocks in a blockchain. *Proof-of-stake* reduces the amount of computational work needed to verify blocks and transactions that keep the blockchain - and thus a cryptocurrency - secure. *PoS* changes the way blocks are verified using the machines of coin owners. The owners offer their coins as collateral for the chance to validate blocks. Coin owners with staked coins become *Validators*. They are then selected randomly to mine, or validate the block. This system randomises whom gets to mine rather than using a competition-based mechanism like *proof-of-work*. To become a validator, a coin owner must stake a specific amount of coins. For instance, Ethereum [48] will require 32 ETH to be staked before a user can become a validator. Blocks are validated by more than one validator, and when a specific number of the validators verify that the block is accurate, it is finalized and closed. Different proof-of-stake mechanisms may use different methods for validating blocks; when Ethereum transitions to PoS, it will use shards for transaction submissions. A validator will verify the transactions and add them to a shard block, which requires at least 128 validators to be attested. Both consensus mechanisms help blockchains synchronize data, validate information, and process transactions. Each method has proven to be successful at maintaining a blockchain, although there are pros and cons to each. However, the two algorithms have very differing approaches. Under PoS, block creators are called validators. Under PoW, the creators are called miners. Miners solve complex mathematical problems to verify transactions. To buy into the position of becoming a block creator, investors need only to purchase the sufficient limit of coins or tokens required to become a validator for a PoS blockchain. For PoW, miners must invest in processing equipment and incur heavy energy charges to power the machines attempting to solve the computations. If PoW should attempt expensive energy cost limiting access to mining and strengthening the security of the blockchain, PoS blockchains often allow for more scalability due to their energy efficiency. These aspects do that proof of stake is more environmental sustainability instead proof of work is more competitive to verifying transactions. The PoS mechanism seeks to solve these problems by effectively substituting staking for computational power, whereby an individual's mining ability is randomized by the network. This means there should be a drastic reduction in energy consumption since miners can no longer rely on massive farms of single-purpose hardware to gain an advantage. A problem regarding PoS is when the 51% attack is used, but it is very unlikely. A 51% attack is when someone controls 51% of a cryptocurrency and uses that majority to alter the blockchain. In PoS, a group or individual would have to own 51% of the staked cryptocurrency. It is not only very expensive to have 51% of

the staked cryptocurrency but also staked currency is collateral for the privilege to mine. The miner(s) that attempt to revert a block through a 51% attack would lose all of their staked coins. This creates an incentive for miners to act in good faith for the benefit of the cryptocurrency and the network. Most other security features of PoS are not advertised, as this might create an opportunity to circumvent security measures. However, most PoS systems have extra security features in place that add to the inherent security behind blockchains and the PoS mechanisms.

The tables below sum up the main differences between the two consensus mechanism.

Proof of Stake
Block creators are called validators
Participants must buy coins or tokens to become a validator
Energy efficiency
Allows for more scalability
Network control can be bought
Validators receive transactions fees as rewards

Table 3.1: PoS

Proof of Work
Block creators are called miners
Participants must buy equipment and energy to become a miner
Not energy efficient
Does not allow for more scalability
Robust security due to expensive upfront requirement
Miners receive block rewards

Table 3.2: PoW

3.3. Cryptography

Each cryptographic primitive that is used with blockchain technology has a distinct role. Hash functions and digital signatures are two concepts of cryptography that are extensively used with blockchain. Cryptography is mainly used in the consensus and application layers of the blockchain. A hashing algorithm is mainly used to create block identity, ensure the integrity of the blockchain, and also acts as a key ingredient of consensus algorithms, such as in Proof of Work. The digital signature, on the other hand, deals with the application layer, where it is used to validate events by embedding them into transactions.

The fundamental objective of cryptography is to enable two parties to communicate over an insecure network. This is achieved by encrypting a Plain Text from the sender to form a Cipher Text that can only be decrypted by the receiver, with whom the sender shares a secret. Third parties may intercept the channel which is used to transport the Cipher Text, but the text does not have any meaning to it, so it does not matter whether the channel is secure or not. Cryptography is also used in many consensus algorithms, which basically exploits the hashing power of the computing systems that form the blockchain network. Digital signatures are used to sign and verify events such as transactions. Asymmetric key cryptography [23] is a core concept in blockchain applications that gives identity to the participants of the network or can prove the ownership of assets.

In order to understand how cryptography is used within Blockchain Technology, it is important to gain a broad understanding of encryption and how it has evolved over the years. In a classical encryption scenario, a Plain text message may be Encrypted using a secret key that is shared with another user or third party over a secure channel. The party that wants to read the text will decrypt the cipher text using the secret key, which will return the original Plain Text. The key is private, and the encryption and decryption algorithms are made public as it is impossible to decrypt the cipher text without the key. Two types of operation are used to transform plaintext to ciphertext: substitution and transposition. Both of these techniques ensure that the operation is reversible, and therefore they could be used in encryption algorithms.

The corner stone of cryptography in cryptocurrency are Public and Private keys. They make decentralization possible. Private keys are used to initiate a transaction. The second ones are used to verify transactions. Public key cryptography (PKC) is also known as asymmetric encryption or trapdoor functions where information is encrypted and decrypted. The idea behind PKC is that it takes an immeasurable amount of time to solve the mathematical function, but it is so easy to calculate and verify it. Private key Used to digitally sign cryptocurrency transactions, for this reason is extremely important

to kept It private and not shared with anyone whatsoever. If someone has your private key, he/she can use it to access your crypto wallet. Public keys are used to encrypt a transaction before it occurs and then to sign the transaction after it is verified. Only after the transaction is signed, it can be added to the blockchain and let others know that the blockchain has been updated. Public keys can be shared without being vulnerable. People cannot access your assets through your public keys.

3.4. Smart Contract

One of the most important application on blockchain is Smart contract. Smart contracts were introduced in the 90s but their idea can be traced back to the 70s in relation to the need to manage the activation or deactivation of a software license according to certain conditions. Recently, smart contracts have become the focus of numerous debates on digital transformation, due to the numerous contexts in which they can be applied and because they represent one of the many dimensions of the growing blockchain phenomenon. Smart contracts are defined by our regulation – *in Legislative Decree no. 135 of 14 December 2018, converted into law by Law no. 12 of 11 February 2019, art. 8-ter* – as

Definition 3.4.1. *A computer program that operates on blockchain technologies and whose execution automatically binds two or more parts on the basis of effects predefined by them" They meet the requirement of the written form after computer identification of the interested parties*

This concept refers to a contract, containing conditions that must be met in order for the operational definitions to be fulfilled. The logic that is respected is that of "if-this-then-that", or "if this happens then it happens". It follows that legal support is therefore useful in the drafting of the smart contract, but not in the verification and activation phase, which takes place automatically. The first step is the conclusion of a traditional contract. Then the two parties transfer the clauses into a smart contract and subsequently, the smart contract is recorded in the blockchain. This means that the transaction cannot be changed. At this point the block will have to be evaluated by the users of the blockchain through the consensus mechanism. The contract will then be monitored by a third-party agent (which can also be an app for smartphones or tablets). When the app sends the signal to the blockchain that one or more conditions have been fulfilled, then the app itself will make sure that the conditions expressed in the smart contract occur automatically. Within a smart contract, there can be as many clauses as are necessary to satisfy the participants and ensure that the operation is completed satisfactorily. To establish the

terms, participants must determine how transactions and their data are represented on the blockchain, agree on the rules governing those transactions, explore all possible exceptions, and define a framework for dispute resolution. Highlight the main differences between smart contracts and contracts governed by the Civil Code. In normal contracts trust is guaranteed by a third party, which can be that of a notary or a lawyer. In the smart contract, the use of a third party figure is lost. However, it is clear that some guarantees must always be respected: the code must not be modifiable, the databases and data sources must be certified and reliable and the methods of reading and controlling the data sources must be certified; Instead in smart contracts there is no room for violation of the conditions signed, since among their intrinsic characteristics there are precisely the automatic execution and inalterability.

There can be several advantages deriving from the use of smart contracts. *Independence* from intermediaries, such as notaries and lawyers, in the phase of verification and approval of the contract; *Immutability* of the code, which excludes the need for third parties who examine the lawfulness and validity of an agreement; Economic savings, largely due to the exclusion of intermediaries in the verification and approval phases; *Greater precision* and *reduction of errors*, since the smart contract, automatically, when the established conditions occur, causes certain actions to occur; *Generically*, simplification of bargaining operations. In addition to several advantages, some elements also emerge to be monitored carefully: *Coded language*: it is necessary that the parties rely on the one hand on a computer expert able to translate the text of the agreement into code, on the other hand to intermediary figures suitable for the correct transmission of the will of the parties to the IT figure, in order to avoid misunderstandings or misunderstandings, which would compromise the real will of the parties; The *interpretation of the contract* (relating to the intention of the contractors, overall interpretation of the clauses, interpretation of good faith, etc.). *Automatic execution of the service*: it does not leave the possibility for the contractors to carry out actions contrary to or different from those provided for in the contractual clauses, limiting the discretionary power of the parties. It can therefore create difficulties with regard to the irrevocability of the agreement and therefore the management of institutions such as: withdrawal, nullity, termination.

4 | Algorand

Algorand [24] was founded by Silvio Micali, a Turing award winner, co-inventor of zero-knowledge proofs, and a world-renowned leader in the field of cryptography and information security. He founded Algorand with the vision of democratising finance based on the blockchain promises. Up to this point, we have been discussing the blockchain technology and enumerating the benefits it brings to applications that transfer value. While listing the definitions in chapter 3, we were making always assuming to deal with particular types of blockchians. However, what happens in reality is that not all the blockchains are implemented in the same way and therefore they provide with different properties. In this section we highlight some of the main categories to take into consideration when choosing a blockchain and we explain how *Algorand* fares in each of these categories.

In this chapter we overview the problem of *Trilemma*. Then we point out the structure and main component of Algorand. In conclusion we explain how smart contract are defined on this blockchain.

4.1. Trilemma

One of the main issue with many blockchains is that they are not able to satisfy simultaneously the key properties of *security*, *scalability*, and *decentralization*. This problem is known as the blockchain *Trilemma*. Micali and his team solved the blockchain trilemma by inventing a new Pure *Proof of Stake (PPoS)* consensus protocol, which is the indeed the one employed by the Algorand blockchain.

Algorand's consensus protocol works by selecting a block proposer and a set of voting committees at each block round, to propose a block and validate the proposal, respectively. The proposer and committees are randomly chosen from the pool of all token holders (the accounts that hold algos), and the likelihood of being chosen is proportional to the stake of the account in the network (i.e. how many algos it has compared to the whole). In this process is involved a wide variety of the finest cryptographic algorithms, like *verifiable random functions* [30] and *cryptographic sortition* [29], which ensure that the vote is fair, that no one can collude, and that the overall system is highly secure. As

already explained, *Trilemma* is a situation where there are three options, but at most only two of them are possible at the same time. If one of the three properties of the *Trilemma* is not met, the system has to face severe consequences. Without *decentralization*, we essentially remain in the same system that already exists today: exclusive and secretive. These systems are usually employed when a small number of delegates is involved, but as a matter of fact they do not have a proper decentralisation. Without *security*, transactions can effectively disappear. Someone could have received money from somebody and given them a product or service, but a while later you realize you never actually got the money in the first place.

Lastly, without *scalability*, the network would be slow and prone to getting clogged. This is not favourable as we want to build a global instant network. We will better discuss these properties later.

Recall that a blockchain essentially has two necessities. First, they have need to make themselves tamper proof and traceable. This is done by one-way hashing techniques and including the hash of the last block within the latest block. Pretty much all blockchains out there do this and are not very different in this case. Second, they have to be able to efficiently generate new blocks. How to choose which block to happen is a tricky question and different blockchains use different consensus algorithms to determine this. Solving the trilemma comes down to developing a consensus algorithm which can do all those three things simultaneously. In the previous chapter different consensus mechanism have been described. In the next section we present more in detail the consensus mechanism used by Algorand for dealing with the *Trilemma*.

4.2. Pure Proof of Stake (PPoS)

The solution implemented in Algorand for these problems is their PPoS algorithm. PPoS does not require users to stake any money in a bond, but simply requires them to have it in the first place. The goal here is not trying to ensure by means of threatening fines that users behave honestly, but rather it makes cheating by a minority of the money impossible, and " cheating by a majority of the money not profitable [41]. This algorithm is secure when most of the money is in honest hands.

See now how the blockchain implements the *Pure Proof of Stake*.

At a high level, Algorand constructs a new block in two phases. In the first phase (*Phase 1*), a single token is randomly selected, and the owner of that token is selected to propose the next block. In *Phase 2*, a few thousand tokens are selected randomly from all tokens in the network. The owners of these tokens are selected to form a *phase 2* committee

which then validate and approve the block proposed in *phase 1*. Since are the tokens to be selected randomly and not the owners, some members may be chosen $k > 1$ times and have k votes in the committee.

In order to understand the security of this process we show the following example from [6]. The core assumption is that in any society there is a minority of bad actors, somewhat around 1% or 2%. If one is in a particularly dangerous society, maybe even 10% or 20%. There are no society where the majority is made of bad actors, as such society could not in fact being called a society and exist. Now, think of the Algorand network as a society. Let us assume the worst case scenario, namely consider 20% of the tokens on the network belong to malicious actors. Then, 1/5 times the token chosen for *Phase 1* will be owned by a bad actor. Assume also that they try to cause disagreement in the society telling some users about the block being "X" while they tell other users about the block being "Y". The number of tokens on the Algorand network is a very big number, close to 2^{256} - even it varies slightly depending on the total voting stake. The probability of a token being selected to vote on a block is approximately $2990/2^{256}$. Hence, the committee which votes on the blocks has size approximated by a Poisson distribution with mean 2990. The threshold for reaching consensus is 2267 votes. Since the adversary holds 20% of the stake, the expected number of votes going to bad actors are roughly $2990/5 = 598$ (the actual number is based on a Poisson distribution), and the expected number of honest voters is roughly $2990 * (4/5) = 2239$. The adversary would win if they are able to get more than the threshold amount of votes for two different values, thereby partitioning the chain and causing disagreement on the network.

As an example ([6]), let us examine what happens in the average case where the number of honest votes are exactly the expectation 2239.

For the adversary to win, they would need $[(2 * 2267) - 2239] / 2$ votes to win — or 1147.5 votes. Since number of votes are integers, they need ≥ 1148 votes to win. The expected value of the votes going to the adversary however, as described above, is 598. Calculating the probability of getting ≥ 1148 votes in a Poisson distribution with $E(X) = 598$ is roughly $(5/10^{87})$ which is very, very small. A really intriguing question of this algorithm regards the committee. In particular, is very important to understand who chooses this committee and how users are selected to be part of that. If the committee is chosen by Algorand (the company) themselves, that would be a highly centralized solution and hold the trilemma. If the users discuss among themselves until they decide who the members are, that's a super slow system as how do you determine you trust someone else, and may never lead to a selection. This is where the interesting part comes in. The committee members choose themselves. To belong to the committee, the nodes run a

cryptographically verifiable lottery over all their accounts, and if any of their coins win the lottery, they are selected to be part of the committee. This lottery is run in isolation with no communication with other nodes on the network. Since it's cryptographically verifiable, nobody can alter their chances to win the lottery either, not even with an enormous computational power. When a user runs the lottery, there are two possible scenarios. Either none of their tokens win the lottery or Some $k \geq 1$ tokens win the lottery. In the first case their opinion about the block is ignored. In the second one the user gets a winning ticket (a short cryptographic proof) to prove they won the lottery. Then, they propagate this ticket along with their opinion about the block to the rest of the network.

Now that we have more clear how consensus works in Algorand, we do a step back to the *Trilemma* analyzing in more details the three important properties.

4.2.1. Security

Assume a powerful malicious adversary would like to corrupt the committee members and influence their votes about the next block. Let's even assume they could do this if they knew who the committee members were. That's where the caveat comes in, they don't know who the committee members are. Since the lottery is run in isolation, only the members alone know if they're selected or not until the time they propagate their proof and opinion about the block to the rest of the network. Once they do propagate their opinion, other nodes know who the committee members are, but at this point it's too late to corrupt them. They've already said what they had to say, and their opinion is already being broadcasted to the network. There is no guarantee when they will be a committee member again. Thus, the malicious adversary can't do anything now to silence them. So essentially, Algorand is secure because firstly, the adversary doesn't know who the committee members are. And when they do find out, it's too late to corrupt them. Additionally, since nobody knows who the committee members are initially, they cannot attack their nodes using methods like DDoS either.

4.2.2. Scalability

Running the lottery only takes a microsecond for any user, no matter how many tokens they have. Also, since all lotteries are run independently of each other, nodes do not need to wait for other nodes to finish doing something first. Once selected, the members propagate a single short message to the rest of the network. So no matter how many users are on the network, only a few thousand messages need to be propagated across the

network. This is highly scalable.

4.2.3. Decentralization

There are not a few users deciding on what the next block will be. Nor is there a fixed committee which makes this decision every time. The committee is chosen randomly and securely, and does not require much computational power at all. This allows everyone on the network to have a chance of being in the committee and voting on the next block.

4.2.4. Non-Forkable

In Algorand, only one block at a time can have the required threshold of committee votes. Accordingly, all transactions are final as soon as they are added to a block. Once a block appears, you can count on it to be there forever and transactions are thus instantly final.

4.3. Principal Components

Now we give an overview on the main characteristics of the blockchain Algorand. First of all each blockchain has its own *native currency* which incentivize the good network behavior. The native currency of Algorand is called the *Algo*. Anybody who owns some *Algo* can register to participate in consensus, which means that you will participate in the process of proposing and voting on new blocks. The *Algo* also acts as a utility token. When someone is building an application, they need the native coin to pay transaction fees and to keep a minimum balance deposits which is a necessary condition for storing data on the blockchain. The cost of these fees and minimum balances is very low, fractions of a penny in most cases. Fees are calculated based on the size of the transaction and a user can choose to augment a fee to help prioritising acceptance into a block when network traffic is high and blocks are consistently full. There is no concept of "gas fees" on Algorand. The minimum fee for a transaction is only 1,000 microAlgos or 0.001 Algos.

Earlier, we have compared a blockchain ledger that is distributed, to a traditional ledger that is owned by a single entity. Technically, a blockchain ledger could be owned and operated by just a few entities, but this wouldn't be a very good blockchain since such a centralized set of nodes could easily manipulate the state of the blockchain. Algorand is completely open and permissionless. This system of fee is also helpful to encourage participation of people. If all the people who are running nodes are from the same company or set of companies then we find ourselves in a similar situation where we are not much better off than just having a central database controlled by a selected minority. On

Algorand, since the protocol is open and permissionless, nodes can and do exist all over the world. Moreover the fact that on Algorand all of the codes are open source, give to the blockchain a lot of transparency. In this way anyone can review it and contribute to it.

Another common problem in almost every blockchains is the management of hard fork.

Forking is when a blockchain diverges into two separate paths. Sometimes it can be intentional, like when a significant part of the community wants to change the fundamentals of the protocol. Other times it is accidental and occurs when two miners find a block at almost the same time. Eventually, one of the paths will be abandoned, which means that all transactions that occurred since that fork on the abandoned path (the orphaned chain) will be invalid. This has important implications for transaction finality, which will be discussed later in details. Since Algorand is pure proof-of-stake and uses a voting mechanism to validate blocks, forking is impossible. In a worst case scenario, if the committee is taking longer to reach agreement, the blockchain will slow down or temporarily stall.

A quality which allows Algorand to be very competitive respect the other blockchain is he speed at which blocks are produced. Indeed this, the amount of transactions that can fit into a block and when those transactions are considered final are important factors to consider when choosing a blockchain. However, even it the performance is very high, for Algorand this issue is and will always be a key focus area for the core development team.

On Algorand, blocks are produced every 4.5 seconds and can hold up to 5,000 transactions, which results in a throughput of about 1,000 transactions per second (1000 TPS) [9]. In proof-of-work blockchains, since forking is a possibility, transactions cannot be considered final until a certain amount of time passes and the likelihood of the transaction being on an orphaned chain is practically zero. This means that the actual throughput of this type of blockchain is caveated by a delay in finality. Downstream processes in an application must take this into account to avoid compounding issues if a transaction ends up being invalid. As we mentioned earlier, Algorand does not have forking so transactions are final as soon as they are confirmed in a block. A throughput of 1,000 TPS then actually means 1,000 finalized transactions per second.

Algorand makes it easy to tokenize, transfer, and program conditions on any instrument of value. Create fungible tokens, NFTs, and security tokens with a single transaction (no smart contract code required). In the same way, programming sophisticated decentralized applications (dApps) with Algorand smart contracts is easy. Developers can write smart contracts in Python or Reach (a Javascript-like language) and can use one of four SDKs (Python, JavaScript, Golang, Java) to connect to on-chain assets or applications.

Algorand Smart Contracts (ASC1) [4] are small programs which serve various functions on the blockchain and operate on layer-1. Layer 1 refers to a basic network, such as Bitcoin, BNB Chain or Ethereum and its underlying infrastructure. Layer 1 blockchains can validate and finalize transactions without the need for another network. Such protocols also have their own native token, which is used to pay transaction fees.

Smart contracts are separated into two main categories, smart contracts, and smart signatures. These types are also referred to as stateful and stateless contracts respectively. The type of contract that is written will determine when and how the logic of the program is evaluated. Both types of contracts are written in the Transaction Execution Approval Language (TEAL), which is an assembly-like language that is interpreted by the Algorand Virtual Machine (AVM) running within an Algorand node. TEAL programs can be written by hand or by using the Python language with the PyTEAL compiler.

4.4. Algorand Smart Contract

Smart contracts are contracts that, once deployed, are remotely callable from any node in the Algorand blockchain [5]. Once deployed, the on-chain instantiation of the contract is referred to as an Application and assigned an Application Id. These applications are triggered by a specific type of transaction called an Application Call transaction. These on-chain applications handle the primary decentralized logic of a dApp.

Applications can modify state associated with the application (global state) or a per application & account (local state) basis. Applications can access on-chain values, such as account balances, asset configuration parameters, or the latest block time. Applications can execute transactions as part of the execution of the logic. One type of transaction they can perform, as of AVM 1.1, is an Application Call transaction, which allows one application to call another. This ability to call other applications enables composability between applications. Applications have an associated Application Account that can hold Algos or ASAs balances and can be used as on-chain escrow accounts. To provide a standard method for exposing an API and encoding/decoding data types from application call transactions, the ABI should be used.

Algorand smart contracts are pieces of logic that reside on the Algorand blockchain and are remotely callable. These contracts are primarily responsible for implementing the logic associated with a distributed application. Smart contracts are referred to as stateful smart contracts or applications in the Algorand documentation. They can generate asset and payment transactions allowing them to function as Escrow accounts on the Algorand

blockchain. Smart contracts can also store values on the blockchain. This storage can be either global or local. Local storage refers to storing values in an accounts balance record if that account participates in the contract. Global storage is data that is specifically stored on the blockchain for the contract globally.

Smart contracts are implemented using two programs. The *ApprovalProgram* is responsible for processing all application calls to the contract, with the exception of the clear call. This program is responsible for implementing most of the logic of an application. Like smart signatures, this program will succeed only if one nonzero value is left on the stack upon program completion or the return opcode is called with a positive value on the top of the stack. The *ClearStateProgram* is used to handle accounts using the clear call to remove the smart contract from their balance record. This program will pass or fail the same way the *ApprovalProgram* does. In either program, if a global or local state variable is modified and the program fails, the state changes will not be applied.

Having two programs allows an account to clear the contract from its state, whether the logic passes or not. When the clear call is made to the contract, whether the logic passes or fails, the contract will be removed from the account's balance record. Note the similarity to the *CloseOut* transaction call which can fail to remove the contract from the account, which is described below. Calls to smart contracts are implemented using *ApplicationCall* transactions. These transaction types are as follows [10]:

NoOp - Generic application calls to execute the *ApprovalProgram*.

OptIn - Accounts use this transaction to begin participating in a smart contract. Participation enables local storage usage.

DeleteApplication - Transaction to delete the application. *UpdateApplication* - Transaction to update TEAL Programs for a contract.

CloseOut - Accounts use this transaction to close out their participation in the contract. This call can fail based on the TEAL logic, preventing the account from removing the contract from its balance record.

ClearState is similar to *CloseOut*, but the transaction will always clear a contract from the account's balance record whether the program succeeds or fails.

The *ClearStateProgram* handles the *ClearState* transaction and the *ApprovalProgram* handles all other *ApplicationCall* transactions.

These transaction types can be created with either goal or the SDKs.

4.5. Algorand Smart Signatures

Smart signatures contain logic that is used to sign transactions, primarily for signature delegation. The logic of the smart signature is submitted with a transaction. While the logic in the smart signature is stored on the chain as part of resolving the transaction, the logic is not remotely callable. Any new transaction that relies on the same smart signature would resubmit the logic. When the logic is submitted to a node the AVM evaluates the logic, where it either fails or succeeds. If the logic of a smart signature fails when executed by the AVM, the associated transaction will not be executed.

Smart signatures can be used in two different modes. When compiled, smart signatures produce an Algorand account that functions similar to any other account on the blockchain. These accounts can hold Algos or assets. These funds are only allowed to leave the account if a transaction occurs from the account that successfully executes the logic within the smart signature. This is similar in functionality to a smart contract escrow, but the logic must be submitted for every transaction from the account. Smart signatures can also be used to delegate some portion of authority to another account. In this case, an account can sign the smart signature which can then be used at a later time to sign a transaction from the original signer's account. This is referred to as account delegation. See the modes of use documentation for more details on these two types of smart signatures.

Once a transaction that is signed with a smart signature, is submitted it is evaluated by an Algorand node using the Algorand Virtual Machine. These contracts only have access to a few global variables, some temporary scratch space, and the properties of the transaction(s) they are submitted with. Smart signatures can also be used as escrow or contract accounts, but in most cases it is preferable to use a smart contract when an escrow is required. Most Algorand transactions are authorized by a signature from a single account or a multisignature account. Algorand's smart signatures allow for a third type of signature using a Transaction Execution Approval Language (TEAL) program, called a logic signature (LogicSig). Smart signatures provide two modes for TEAL logic to operate as a LogicSig, to create a contract account that functions similar to an escrow or to delegate signature authority to another account. These modes are used to approve transactions in different ways which are described below. Both modes make use of Logic Signatures. While using smart signatures for contract accounts is possible, it is now possible to create a contract account using a smart contract. Logic Signatures, referenced as LogicSig, are structures that contain the following four parts.

- Logic: Raw Program Bytes

- Sig: Signature of Program Bytes
- Msig: Multi-Signature of Program Bytes
- Args: Array of Bytes String Passed to the Program

Before a LogicSig can be used with a transaction, it first must be a valid Logic Signature. The LogicSig is considered valid if one of the following scenarios is true.

- Sig contains a valid Signature of the program from the account that is sending the Transaction
- Msig contains a valid Multi-Signature of the program from the Multi-Signature account sending the Transaction
- The hash of the program is equal to the Sender's Address

The first two cases are examples of delegation. An account owner can declare that on their behalf the signed logic can authorize transactions. These accounts can be either single or multi-signature accounts. Account delegation is described in further detail below. The third case is an account wholly governed by the program. The program cannot be changed. Once Algos or assets have been sent to that account, Algos or assets only leave when there is a transaction that approves it. This usage case is considered a contract account which is described below.

4.6. Contract account

For each unique compiled smart signature program there exists a single corresponding Algorand address, output by goal clerk compile. To use a TEAL program as a contract account, send Algos to its address to turn it into an account on Algorand with a balance. Outwardly, this account does not look different from any other Algorand account and anyone can send it Algos or Algorand Standard Assets to increase its balance. The account differs in how it authenticates spends from it, in that the logic determines if the transaction is approved. To spend from a contract account, create a transaction that will evaluate to True against the TEAL logic, then add the compiled TEAL code as its logic signature. It is worth noting that anyone can create and submit the transaction that spends from a contract account as long as they have the compiled TEAL contract to add as a logic signature.

4.7. Delegated approval

Smart signatures can also be used to delegate signature authority, which means that a private key can sign a TEAL program and the resulting output can be used as a signature in transactions on behalf of the account associated with the private key. The owner of the delegated account can share this logic signature, allowing anyone to spend funds from his or her account according to the logic within the TEAL program. For example, if user A wants to set up a recurring payment with her utility company for up to 200 Algos every 50000 rounds, it creates a TEAL contract that encodes this logic, signs it with her private key, and gives it to the utility company. The utility company uses that logic signature in the transaction they submit every 50000 rounds to collect payment from A. The logic signature can be produced from either a single or multi-signature account.

4.8. Conclusion

Algorand is an innovative blockchain that has as crucial point to find a solution to the Trilemma. It has his own coin and the PPoS allow to reach excellent result in term of performance. In this chapter we have presented the main aspect of Algorand. We have seen what it is, how it works and what are its goals. During the following chapter we point out our attention on Teal. In particular we study in details how use this programming language to build smart contract. This will be useful for a proper understanding of the chapter 6 and chapter 7.

5 | Teal

In this chapter is given a detailed overview of the language used by Algorand to develop smart contract on its blockchain. It is called TEAL. It is important for our purpose to introduce it because our work focuses on a compiler that translates smart contract written in a high-level language into Teal. First is introduced this language highlighting the main properties. After, some criticalities are pointed out.

5.1. Introduction Teal

Smart contracts and smart signatures are written in Transaction Execution Approval Language (TEAL) [12]. These contracts can be written directly or with Python using the PyTeal library.

TEAL is an assembly-like language and is processed by the Algorand Virtual Machine (AVM) [7]. The language is a Turing-complete language that supports looping and subroutines, but limits the amount of time the contract has to execute using a dynamic opcode cost evaluation algorithm. TEAL programs are processed one line at a time pushing and popping values on and off the stack. These stack values are either unsigned 64 bit integers or byte strings. TEAL provides a set of operators that operate on the values within the stack. It also allows arguments to be passed into the program from a transaction, a scratch space to temporarily store values for use later in the program, access to grouped or single transaction properties, global values, a couple of pseudo operators, constants and flow control functions like `bnz` for branching and `callsub` for calling subroutines. Smart contracts can read and write global storage for the contract and local storage for accounts that opt-in to the contract. SCs also have the ability to generate both asset and payment transactions within the logic. Using this ability, they can function as escrow accounts.

Some of the opcodes [8] in TEAL are only valid for a specific contract type. These are denoted in the TEAL Opcodes documentation with a Mode attribute. This attribute will be set to Signature for smart signatures and Application for smart contracts. For example, reading account assets or Algo balances is only available in smart contracts. In the Figure 5.1 we have a Teal architecture overview. Moving from left to right we notice first

the Teal algorithm and the last two columns contain respectively the transaction fields and the global parameters. The first are information about the current transaction being evaluated, the last one refers to the current state of the blockchain.

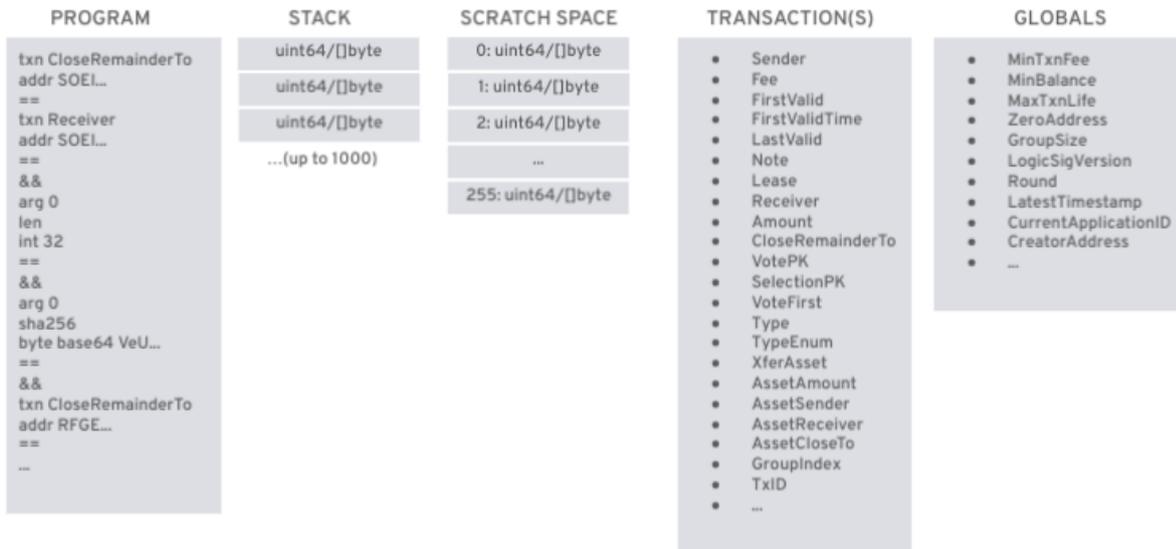


Figure 5.1: TEAL Architecture Overview, from Algorand Developer Portal [12]

The primary purpose of a TEAL program is to return either true or false. When the program completes, if there is a non-zero value on the stack then it returns true. If there is a zero value or the stack is empty, it will return false. If the stack has more than one value the program also returns false unless the return opcode is used. The following diagram illustrates how the stack machine processes the program.

Program line number 1: Figure 5.2

The program uses the `txn` to reference the current transaction's list of properties. Grouped transaction properties are referenced using `gtxn` and `gtxns`. The number of transactions in a grouped transaction is available in the global variable `GroupSize`. To get the first transaction's receiver use `gtxn 0 Receiver`.

The TEAL specification provides several pseudo opcodes for convenience. [8]

The `addr` pseudo opcode converts Algorand addresses to a byte constant and pushes the result to the stack. (Figure 5.3)

TEAL provides operators to work with data that is on the stack. For example, the `==` operator evaluates if the last two values on the stack are equal and pushes either a 1 or 0 depending on the result (Figure 5.4). The number of values used by an operator will depend on the operator. The TEAL Opcodes documentation explains arguments and

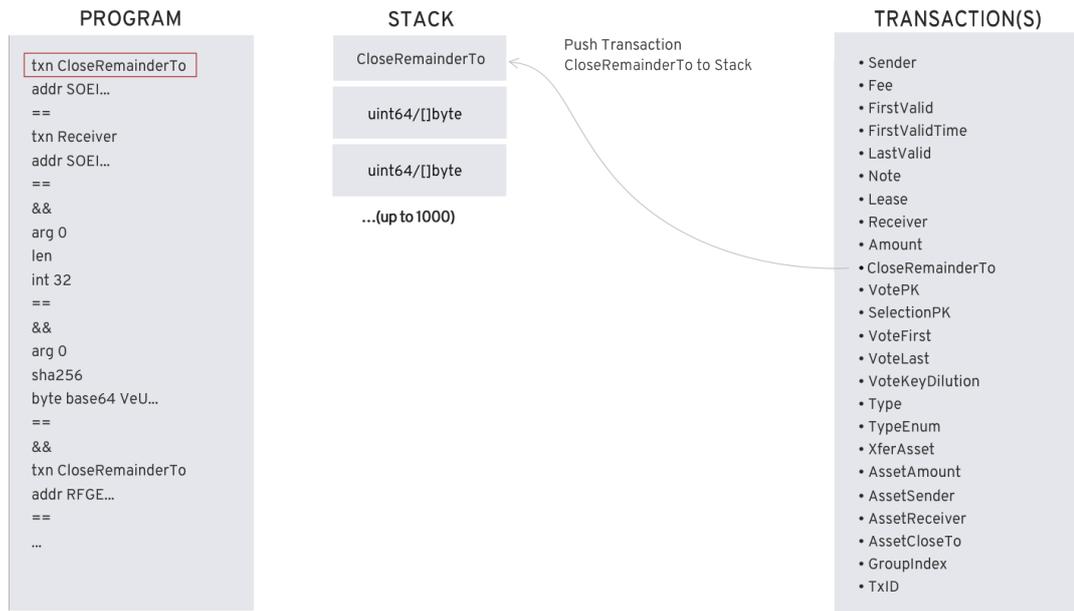


Figure 5.2: Getting Transaction Properties, from Algorand Developer Portal

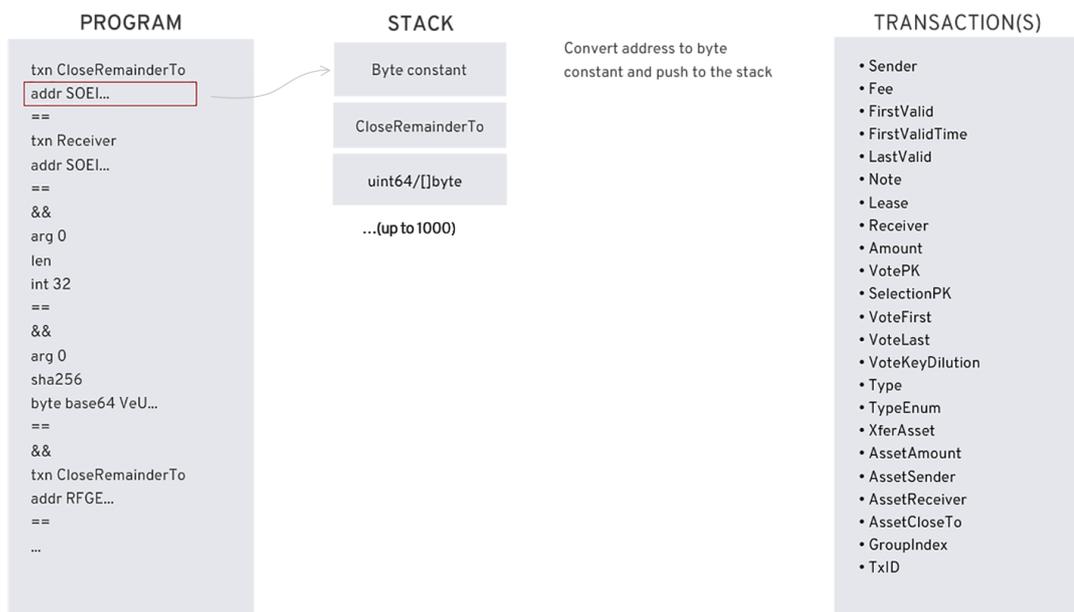


Figure 5.3: Pseudo Opcodes, from Algorand Developer Portal

return values.

Smart signature are limited to 1000 bytes in size. Size encompasses the compiled program plus arguments. Smart contracts are limited to 2KB total for the compiled approval and

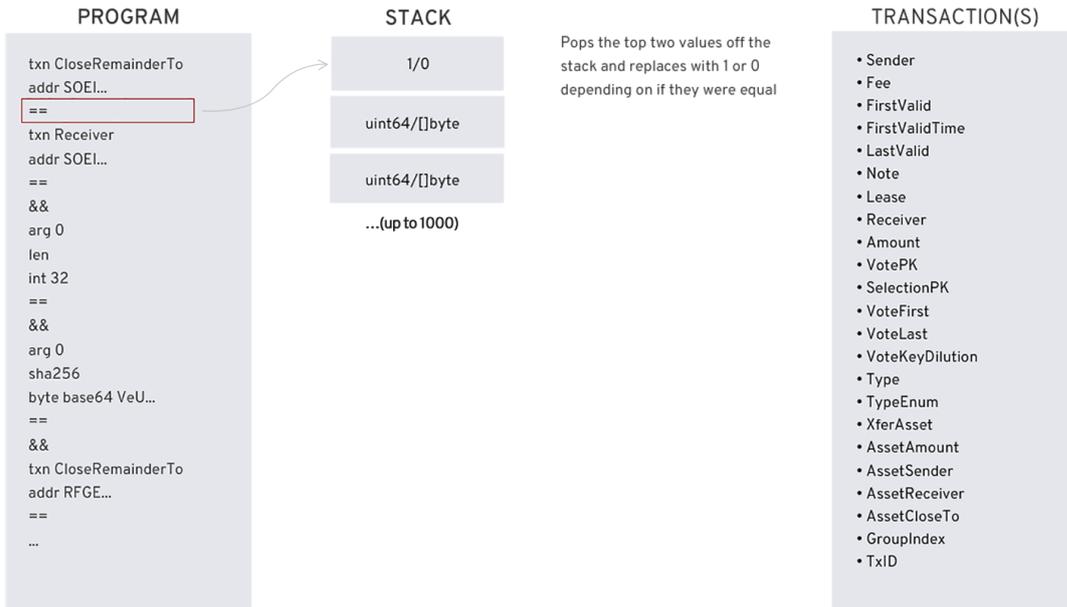


Figure 5.4: Operators, from Algorand Developer Portal

clear programs. This size can be increased in 2KB increments, up to an 8KB limit for both programs. For optimal performance, smart contracts and smart signatures are also limited in opcode cost. This cost is evaluated when a smart contract runs and is representative of its computational expense. Every opcode executed by the AVM has a numeric value that represents its computational cost. Most opcodes have a computational cost of 1. Some, such as SHA256 (cost 35) or `ed25519verify` (cost 1900) have substantially larger computational costs. Smart signatures are limited to 20,000 for total computational cost. Smart contracts invoked by a single application transaction are limited to 700 for either of the programs associated with the contract. However, if the smart contract is invoked via a group of application transactions, the computational budget is considered pooled. The total opcode budget will be 700 multiplied by the number of application transactions within the group. So if the maximum transaction group size is used and all are application transactions, the computational budget would be $700 \times 16 = 11200$. The TEAL Opcodes reference lists the opcode cost for every opcode.

The stack starts empty and can contain values of either `uint64` or byte-arrays (byte-arrays may not exceed 4096 bytes in length). Most operations act on the stack, popping arguments from it and pushing results to it. Some operations have immediate arguments that are encoded directly into the instruction, rather than coming from the stack.

The maximum stack depth is 1000. If the stack depth is exceeded or if a byte-array element exceeded 4096 bytes, the program fails.

In addition to the stack there are 256 positions of scratch space. Like stack values, scratch locations may be uint64s or byte-arrays. Scratch locations are initialized as uint64 zero. Scratch space is accessed by the load and store opcodes which move data from or to scratch space, respectively.

5.2. Execution Modes

Starting from v2, the AVM can run programs in two modes: *LogicSig* or *stateless mode*, used to execute *Smart Signatures* and *Application* or *stateful mode*, used to execute *Smart Contracts*.

Differences between modes include [11]:

1. Max program length (consensus parameters *LogicSigMaxSize*, *MaxAppTotalProgramLen* & *MaxExtraAppProgramPages*)
2. Max program cost (consensus parameters *LogicSigMaxCost*, *MaxAppProgramCost*)
3. Opcode availability. Refer to opcodes document for details [8].
4. Some global values, such as *LatestTimestamp*, are only available in stateful mode.
5. Only Applications can observe transaction effects, such as Logs or IDs allocated to ASAs or new Applications.

Smart Signatures execute as part of testing a proposed transaction to see if it is valid and authorized to be committed into a block. If an authorized program executes and finishes with a single non-zero uint64 value on the stack then that program has validated the transaction it is attached to.

The program has access to data from the transaction it is attached to (txn op), any transactions in a transaction group it is part of (gtxn op), and a few global values like consensus parameters (global op). Some "Args" may be attached to a transaction being validated by a program. Args are an array of byte strings. A common pattern would be to have the key to unlock some contract as an Arg. Be aware that Smart Signature Args are recorded on the blockchain and publicly visible when the transaction is submitted to the network, even before the transaction has been included in a block. These Args are not part of the transaction ID nor of the TxGroup hash. They also cannot be read from other programs in the group of transactions.

A program can either authorize some delegated action on a normal signature-based or multisignature-based account or be wholly in charge of a contract account.

If the account has signed the program (by providing a valid ed25519 signature or valid multisignature for the authorizer address on the string "Program" concatenated with the program bytecode) then: if the program returns true the transaction is authorized as if the account had signed it. This allows an account to hand out a signed program so that other users can carry out delegated actions which are approved by the program. Note that Smart Signature Args are not signed.

If the *SHA512_256* hash of the program (prefixed by "Program") is equal to authorizer address of the transaction sender then this is a contract account wholly controlled by the program. No other signature is necessary or possible. The only way to execute a transaction against the contract account is for the program to approve it.

The bytecode plus the length of all Args must add up to no more than 1000 bytes (consensus parameter *LogicSigMaxSize*). Each opcode has an associated cost and the program cost must total no more than 20,000 (consensus parameter *LogicSigMaxCost*). Most opcodes have a cost of 1, but a few slow cryptographic operations have a much higher cost. Prior to v4, the program's cost was estimated as the static sum of all the opcode costs in the program (whether they were actually executed or not). Beginning with v4, the program's cost is tracked dynamically, while being evaluated. If the program exceeds its budget, it fails.

Smart Contracts are executed in *ApplicationCall* transactions. Like Smart Signatures, contracts indicate success by leaving a single non-zero integer on the stack. A failed Smart Contract call is not a valid transaction, thus not written to the blockchain. Nodes maintain a list of transactions that would succeed, given the current state of the blockchain, called the transaction pool. Nodes draw from the pool if they are called upon to propose a block.

Smart Contracts have access to everything a Smart Signature may access (see previous section), as well as the ability to examine blockchain state such as balances and contract state (their own state and the state of other contracts). They also have access to some global values that are not visible to Smart Signatures because the values change over time. Since smart contracts access changing state, nodes must rerun their code to determine if the *ApplicationCall* transactions in their pool would still succeed each time a block is added to the blockchain.

Smart contracts have limits on their execution cost (700, consensus parameter *MaxAppProgramCost*). Before v4, this was a static limit on the cost of all the instructions in the program. Since then, the cost is tracked dynamically during execution and must not exceed *MaxAppProgramCost*. Beginning with v5, programs costs are pooled and tracked

dynamically across app executions in a group. If n application invocations appear in a group, then the total execution cost of such calls must not exceed $n * \text{MaxAppProgramCost}$. In v6, inner application calls become possible, and each such call increases the pooled budget by MaxAppProgramCost .

Executions of the `ClearStateProgram` are more stringent, in order to ensure that applications may be closed out, but that applications also are assured a chance to clean up their internal state. At the beginning of the execution of a `ClearStateProgram`, the pooled budget available must be MaxAppProgramCost or higher. If it is not, the containing transaction group fails without clearing the app's state. During the execution of the `ClearStateProgram`, no more than MaxAppProgramCost may be drawn. If further execution is attempted, the `ClearStateProgram` fails, and the app's state is cleared.

Smart contracts have limits on the amount of blockchain state they may examine. Opcodes may only access blockchain resources such as `Accounts`, `Assets`, and contract state if the given resource is available.

A resource in the "foreign array" fields of the `ApplicationCall` transaction (`txn.Accounts`, `txn.ForeignAssets`, and `txn.ForeignApplications`) is available.

The `txn.Sender`, global `CurrentApplicationID`, and global `CurrentApplicationAddress` are available.

Prior to v4, all assets were considered available to the `asset_holding_get` opcode, and all applications were available to the `app_local_get_ex` opcode.

Since v6, any asset or contract that was created earlier in the same transaction group (whether by a top-level or inner transaction) is available. In addition, any account that is the associated account of a contract that was created earlier in the group is available.

Since v7, the account associated with any contract present in the `txn.ForeignApplications` field is available.

6 | Formal model for ASC

In this chapter we present the main features of a formal model of Algorand smart contracts from [16]. All the theorems and definitions presented in this chapter are explained in [16]. The authors focus on TEAL ASC1, and develop a mathematical model for Algorand contracts and transactions suitable for formal reasoning on their behaviour, and for the verification of their properties. The general goal is to develop techniques and tools to ensure that contracts are correct and secure. Even informal reasoning on non-trivial smart contracts can be challenging, and it may require to resort to experimental validation or direct inspection of the platform source code. Therefore, authors formulate a new model that is high-level enough to simplify the design of Algorand smart contracts and enable formal reasoning about their security and correctness properties. Such a model should also allow programmers to express Algorand contracts in a simple declarative language, inspired by PyTeal, and provide a basis for the automated verification of Algorand smart contracts.

The formal model is faithful to the actual ASC1 implementation; it strives at being high-level and simple to understand, while covering the most commonly used primitives and mechanisms of Algorand, and supporting the specification and verification of non-trivial smart contracts. To achieve these objectives, high-level abstractions over low-level details are introduced: e.g., since TEAL code has the purpose of accepting or rejecting transactions, they are modelled using expressions that evaluate to true or false (similarly to PyTEAL). Also, different transaction types are formalised by focusing on their function, rather than their implementation.

We first discuss the fundamental concepts of the formal model, and then describe how an Account and a Transaction are defined. Our presentation follows the one in [16].

The user is indicated with **a**, **b**. As explained into chapter 3 a fundamental concept in blockchain are the public/private key pairs (k_a^p, k_a^s) . Users interact with Algorand through pseudonymous identities, obtained as a function of their public keys. Hereafter, they freely use **a** to refer to the public or the private key of **a**, or to the user associated with them, relying on the context to resolve the ambiguity. The purpose of Algorand is

to allow users to exchange assets τ, τ', \dots as well as the Algorand native cryptocurrency Algo. The authors adopt the following notational convention:

- lowercase letters for single entities (e.g., a user a);
- uppercase letters for sets of entities (e.g., a set of users A);
- calligraphic uppercase letters for sequences of entities (e.g., list of users A).

Given a sequence \mathcal{L} , the authors write $|\mathcal{L}|$ for its length, $\text{set}(\mathcal{L})$ for the set of its elements, and $\mathcal{L}.i$ for its i^{th} element ($i \in 1 \dots |\mathcal{L}|$); ϵ denotes the empty sequence. They also define the following functions:

- $\{x \rightarrow v\}$ for the function mapping x to v , and having domain equal to x ;
- $f\{x \rightarrow v\}$ for the function mapping x to v , and y to $f(y)$ if $y \neq x$;
- $f\{x \rightarrow \perp\}$ for the function undefined at x , and mapping y to $f(y)$ if $y \neq x$.

All the notation are summarised on the Table A.1.

An account is a deposit of one or more crypto-assets. Accounts are modelled as terms $x[\sigma]$, where x is an address uniquely identifying the account, and σ is a balance, i.e., a finite map from assets to non-negative 64-bit integers. In the concrete Algorand, an address is a 58-characters word; for mathematical elegance, in the model the authors represent an address as either:

- a single user \mathbf{a} . Performing transactions on $\mathbf{a}[\sigma]$ requires a 's authorization;
- a pair (\mathcal{A}, n) , where \mathcal{A} is a sequence of users, and $1 \leq n \leq |\mathcal{A}|$, are multisig (multi-signature) addresses. Performing transactions on $(\mathcal{A}, n)[\sigma]$ requires that at least n users out of those in \mathcal{A} grant their authorization;
- a script \mathbf{e} (i.e. smart contract). Performing transactions on $\mathbf{e}[\sigma]$ requires \mathbf{e} to evaluate to true

Each balance is required to own *Algos*, have at least 100000 *micro-Algos* for each owned asset, and cannot control more than 1000 assets. Formally, the authors say that σ is a valid balance (in symbols, $\models \sigma$) when:

$$\text{Algo} \in \text{dom}(\sigma) \wedge \sigma(\text{Algo}) \geq 100000 \cdot |\text{dom}(\sigma)| \wedge |\text{dom}(\sigma)| \leq 1001$$

Accounts can append various kinds of transactions to the blockchain, in order to, e.g., alter their balance or set their usage policies. Transactions are modelled as records with the structure in Table A.2.

Each transaction has a *type*, which determines which of the other fields are relevant. The field *snd* usually refers to the subject of the transaction (e.g., the *sender* in an *assets* transfer), while *rcv* refers to the receiver in an *assets* transfer. The fields *asst* and *val* refer, respectively, to the affected *assets*, and to its *amount*. The fields *fv* (“first valid”), *lv* (“last valid”) and *lx* (“lease”) are used to impose time constraints.

Algorand groups transactions into *rounds* $r = 1, 2, \dots$. To establish when a transaction t is valid, we must consider both the current round r , and a lease *map* f_{lx} binding pairs (address, lease identifier) to rounds: this is used to enforce mutual exclusion between two or more transactions. Formally, the authors define the *temporal validity of a transaction* t by the predicate $f_{lx}, r \models t$, which holds whenever:

$$t.fv \leq r \leq t.lv \text{ and } (t.lv - t.fv) \leq \Delta_{max}$$

and

$$(t.lx = 0 \text{ or } (t.snd, t.lx) \notin \text{dom}(f_{lx}) \text{ or } r \geq f_{lx}(t.snd, t.lx))$$

.

First, the current round must lie between $t.fv$ and $t.lv$, whose distance cannot exceed Δ_{max} rounds. Second, t must have a null lease identifier, or the identifier has not been seen before (i.e., $f_{lx}(t.snd, t.lx)$ is undefined), or the lease has expired (i.e., $r > f_{lx}(t.snd, t.lx)$). When performed, a transaction with non-null lease identifier acquires the lease on $(t.snd, t.lx)$, which is set to $t.lv$.

6.1. Blockchain states

Now is defined how the evolution of the Algorand blockchain is modelled as a labelled transition system.

A blockchain state Γ has the form:

$$x_1[\sigma_1] \mid \dots \mid x_n[\sigma_n] \mid r \mid \mathcal{T}_{lv} \mid f_{asst} \mid f_{lx} \mid f_{frz} \quad (6.1)$$

where all addresses x_i are distinct, \mid is commutative and associative, and:

- r is the current round;
- \mathcal{T}_{lv} is the set of performed transactions whose “last valid” time lv has not expired. This set is used to avoid double spending ;
- f_{asst} maps each asset to the addresses of its manager and creator;

- f_{lx} is the lease map (from pairs (address, integer) to integers), used to ensure mutual exclusion between transactions;
- f_{frz} is a map from addresses to sets of assets, used to freeze assets.

The *initial state* Γ_0 is defined as:

$$a_0 [\{Algo \rightarrow v_0\}] \mid 0 \mid \emptyset \mid f_{asst} \mid f_{lx} \mid f_{frz} \quad (6.2)$$

where $dom(f_{asst}) = dom(f_{lx}) = dom(f_{frz}) = \emptyset$, a_0 is the initial user address, and $v_0 = 1016$ (which is the total supply of 10 billions Algos).

Follow now the formalization of the ASC1 state machine, by defining how it evolves by single transactions, and then including atomic groups of transactions, smart contracts, and the authorization of transactions.

6.2. Executing single transactions

The authors write $\Gamma \xrightarrow{t}_1 \Gamma'$ to mean: *if* the transaction t is performed in blockchain state Γ , then the blockchain evolves to state Γ' . 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 6.1. We now illustrate all cases, depending on the transaction type ($t.type$).

When $\tau \in dom(\sigma)$, the authors use the shorthand $\sigma + v : \tau$ to update balance σ by adding v units to token τ ; similarly, $\sigma - v : \tau$ is written to decrease τ by v units:

$$\sigma + v : \tau \equiv \sigma\{\tau \rightarrow \sigma(\tau) + v\} \quad (6.3)$$

$$\sigma - v : \tau \equiv \sigma\{\tau \rightarrow \sigma(\tau) - v\} \quad (6.4)$$

Pay to a new account.

Let $t.snd = x_i$ for some $i \in 1 \cdots n$, let $t.rcv = y \in x_1, \dots, x_n$ (i.e., the sender account \mathbf{x} is already in the state, while the receiver \mathbf{y} is not), and let $t.val = v$. The rule has the following preconditions:

- c1.** t does not cause double-spending ($t \in T_{lv}$);
- c2.** the time interval of the transaction, and its lease, are respected ($f_{lx}, r \models t$);
- c3.** the updated balance of \mathbf{x}_i is valid ($\models \sigma_i - v : Algo$);
- c4.** the balance of the new account at address y is valid ($\models \{Algo \rightarrow v\}$)

If these conditions are satisfied, the new state Γ' is the following:

$$x_i [\sigma_i - v : Algo] \mid y [\{Algo \rightarrow v\}] \mid \cdots \mid r \mid T_{lv} \cup \{t\} \mid f_{asst} \mid upd(f_{lx}, t) \mid f_{frz} \quad (6.5)$$

In the new state, the Algo balance of \mathbf{x}_i is decreased by \mathbf{v} units, and a new account at \mathbf{y} is created, containing exactly the \mathbf{v} units taken from \mathbf{x}_i . The balances of the other accounts are unchanged. The updated lease mapping is:

$$upd(f_{lx}, t) = \begin{cases} f_{lx}(t.snd, t.lx) \rightarrow t.lv & \text{if } t.lx \neq 0 \\ f_{lx} & \text{otherwise} \end{cases} \quad (6.6)$$

Note that all transaction types check conditions **c1** and **c2** above; further, all transactions check that updated account balances are valid (as in **c3** and **c4**).

Pay to an existing account.

Let $t.snd = x_i$, $t.rcv = x_j$, $t.val = v$, and $t.asst = \tau$.

Besides the common checks, performing t requires that x_j has “opted in” τ (formally, $\tau \in dom(\sigma_j)$), and τ must not be frozen in accounts x_i and x_j (formally, $\tau \in f_{frz}(x_i) \cup f_{frz}(x_j)$). If $x_i = x_j$, then in the new state the balance of τ in x_i is decreased by v units, and that of τ in x_j is increased by v units:

$$x_i [\sigma_i - v : \tau] \mid x_j [\sigma_j + v : \tau] \mid \cdots \mid r \mid T_{lv} \cup \{t\} \mid f_{asst} \mid upd(f_{lx}, t) \mid f_{frz} \quad (6.7)$$

where all accounts but \mathbf{x}_i and x_j are unchanged. Otherwise, if $x_i = x_j$, then the balance of x_i is unchanged, and the other parts of the state are as above.

Close.

Let $t.snd = x_i$, $t.rcv = x_j = x_i$, and $t.asst = \tau$.

Performing t has two possible outcomes, depending on whether τ is Algo or a user-defined asset. If $\tau = Algo$, we must check that σ_i contains only Algos.

If so, the new state is:

$$x_j [\sigma_j + \sigma_i(Algo) : Algo] \mid \cdots \mid r \mid T_{lv} \cup \{t\} \mid f_{asst} \mid upd(f_{lx}, t) \mid f_{frz} \quad (6.8)$$

where the new state no longer contains the account x_i , and all the Algos in x_i are transferred to x_j . Instead, if $\tau = Algo$, performing \mathbf{t} requires to check only that x_i actually contains τ , and that x_j has “opted in” τ . Further, τ must not be frozen for addresses x_i

and x_j , i.e. $\tau \in f_{frz}(x_i) \cup f_{frz}(x_j)$.

The new state is:

$$x_i [\sigma_i\{\tau \rightarrow \perp\}] \mid x_j [\sigma_j + \sigma_i(\tau) : \tau] \mid \cdots \mid r \mid T_{lv} \cup \{t\} \mid f_{asst} \mid upd(f_{lx}, t) \mid f_{frz} \quad (6.9)$$

where τ is removed from x_i , and all the units of τ in x_i are transferred to x_j .

6.3. Atomic transaction

Atomic transfers allow state transitions to atomically perform sequences of transactions. To atomically perform a sequence $\mathcal{T} = t_1, \dots, t_n$ from a state Γ , we must check that all the transactions t_i can be performed in sequence, i.e. the following precondition must hold (for some $\Gamma_1, \dots, \Gamma_n$):

$$\Gamma \xrightarrow{t_1} \Gamma_1 \cdots \Gamma_{n-1} \xrightarrow{t_n} \Gamma_n \quad (6.10)$$

If so, the state Γ can take a single-step transition labelled \mathcal{T} . Denoting the new transition relation with \rightarrow , then the atomic execution of \mathcal{T} in Γ is defined as follows:

$$\Gamma \xrightarrow{\mathcal{T}} \Gamma_n \quad (6.11)$$

6.4. Executing smart contracts

In Algorand, custom authorization policies can be defined with a smart contract language called TEAL. As already described into chapter 5 TEAL is a bytecode-based stack language, with an official programming interface for Python (called PyTeal): in the formal model, the authors take inspiration from the latter to abstract TEAL bytecode scripts as terms, with the syntax in Table A.3.

Besides standard arithmetic-logical operators, TEAL includes operators to count and index all transactions in the current atomic group, and to access their id and fields.

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 evaluation function $\llbracket \mathbf{e} \rrbracket_{\mathcal{T}, i}^{\mathcal{W}}$ (Table A.4) evaluates \mathbf{e} using 3 parameters: a sequence of arguments \mathcal{W} , a sequence of transactions \mathcal{T} forming an atomic group, and the index $i < |\mathcal{T}|$ of the transaction containing \mathbf{e} . The script $tx(n).f$ evaluates to the field f of the n^{th} transaction in group \mathcal{T} . The size of \mathcal{T} is given by $txlen$, while $txpos$ returns the index \mathbf{i} of the transaction containing the script being evaluated. The script

$arg(n)$ returns the n^{th} argument in \mathcal{W} . The script $H(e)$ applies a public hash function H to the evaluation of e . The script $versig(e_1, e_2, e_3)$ verifies a signature e_2 on the message obtained by concatenating the enclosing script and e_1 , using public key e_3 . All operators in Table A.4 are strict: they fail if the evaluation of any operand fails.

6.5. Authorizing transactions, and user-blockchain interaction

As noted before, the mere existence of a step $\Gamma \xrightarrow{t}_1 \Gamma'$ does not imply that t can actually be issued. For this to be possible, users must provide a sequence \mathcal{W} of witnesses, satisfying the authorization predicate associated with t ; such a predicate is uniquely determined by the authorizer address of t , written $auth(t, f_{asst})$.

For transaction types *close*, *pay*, *gen*, *optin* the authorizer address is $t.snd$; for *burn*, *rvk*, *frz* and *unfrz* on an asset τ it is the *asset manager* $f_{asst}(\tau)$.

Intuitively, if $auth(t, f_{asst}) = x$, then \mathcal{W} authorizes t iff:

1. if x is a multisig address (\mathcal{A}, n) , then \mathcal{W} contains at least n signatures of t , made by users in \mathcal{A} ; (if \mathbf{x} is a single-user address \mathbf{a})
2. if \mathbf{x} is a script e , then e evaluates to true under the arguments \mathcal{W} .

Formalize now the intuition above. Since the evaluation of scripts depends on a whole group of transactions \mathcal{T} , and on the index i of the current transaction within \mathcal{T} , in the related paper the authorization predicate is defined as $\mathcal{W} \models \mathcal{T}, i$ (read: “ \mathcal{W} authorizes the i^{th} transaction in \mathcal{T} ”). Let $sig_{\mathcal{A}}(m)$ stand for the set of signatures containing $sig_a(m)$ for all $a \in \mathcal{A}$; then, $\mathcal{W} \models \mathcal{T}, i$ holds whenever:

$$\text{if } auth(\mathcal{T}.i, f_{asst}) = (\mathcal{A}, n), \text{ then } |set(\mathcal{W}) \cap sig_{set(\mathcal{A})}(\mathcal{T}, i)| \geq n \quad (6.12)$$

$$\text{if } auth(\mathcal{T}.i, f_{asst}) = e, \text{ then } \llbracket e \rrbracket_{\mathcal{T}, i}^{\mathcal{W}} = true \quad (6.13)$$

Note that, in general, the sequence of witnesses \mathcal{W} is not unique, i.e., it may happen that $\mathcal{W} \models \mathcal{T}, i$ and $\mathcal{W}' \models \mathcal{T}, i$ for $\mathcal{W} \neq \mathcal{W}'$. For instance, the **Oracle** contract accepts transactions with witnesses of the form 0s or 1s', where the first element of the sequence represents the oracle's choice, and the second element is the oracle's signature.

Given a *sequence of sequences* of witnesses $\mathbf{W} = \mathcal{W}_0 \cdots \mathcal{W}_{n-1}$ with $n = |\mathcal{T}|$, the *group authorisation predicate* $\mathbf{W} \models \mathcal{T}$ holds iff $\mathcal{W}_i \models \mathcal{T}, i \quad \forall i \in 0 \cdots n-1$.

User-blockchain interaction.

Model the interaction of users with the blockchain as a transition system. Its states are pairs (Γ, \mathbf{K}) , where Γ is a blockchain state, while \mathbf{K} is the set of authorisation bitstrings currently known by users. The transition relation \xRightarrow{l} (with $l \in \{w, \checkmark, \mathbf{W} : \mathcal{T}\}$) is given by the rules:

$$\frac{}{(\Gamma, \mathbf{K}) \xRightarrow{\mathbf{w}} (\Gamma, \mathbf{K} \cup \{\mathbf{w}\})} \quad \frac{\Gamma \xrightarrow{\checkmark} \Gamma'}{(\Gamma, \mathbf{K}) \xRightarrow{\checkmark} (\Gamma', \mathbf{K})} \quad \frac{\Gamma \xrightarrow{\mathcal{T}} \Gamma' \quad \text{set}(\mathbf{W}) \subseteq \mathbf{K} \quad \mathbf{W} \models \mathcal{T}}{(\Gamma, \mathbf{K}) \xRightarrow{\mathbf{W}:\mathcal{T}} (\Gamma', \mathbf{K})}$$

With the first two rules, users can broadcast a witness \mathbf{w} , or advance to the next round. The last rule gathers from \mathbf{K} a sequence of witnesses \mathbf{W} , and lets the blockchain perform an atomic group of transactions \mathcal{T} if authorized by \mathbf{W} .

6.6. Fundamental properties of ASC1

We now exploit the formal model to establish some fundamental properties of ASC1. Theorem 6.1 states that the same transaction t cannot be issued more than once, i.e., there is no *double-spending*. In the statement, the authors use $\rightarrow^* \xrightarrow{\mathcal{T}} \rightarrow^*$ to denote an arbitrarily long series of steps including a group of transactions \mathcal{T} .

Theorem 6.1. (No double-spending) [16]

Let $\Gamma_0 \rightarrow^* \xrightarrow{\mathcal{T}} \rightarrow^* \Gamma \xrightarrow{\mathcal{T}'} \Gamma'$. Then, no transaction occurs more than once in $\mathcal{T}\mathcal{T}'$.

Define the value of an asset τ in a state $\Gamma = x_1[\sigma_1] \mid \dots \mid x_n[\sigma_n] \mid r \mid \dots$ as the sum of the balances of τ in all accounts in Γ :

$$\text{val}_\tau(\Gamma) = \sum_{i=1}^n \text{val}_\tau(\sigma_i) \quad \text{where} \quad \text{val}_\tau(\sigma) = \begin{cases} \sigma(\tau) & \text{if } \tau \in \text{dom}(\sigma) \\ 0 & \text{otherwise} \end{cases} \quad (6.14)$$

Equation 6.2 states that, once an asset is minted, its value remains constant, until the asset is eventually burnt. In particular, since Algos cannot be burnt (nor minted, unlike in Bitcoin and Ethereum), their amount remains constant.

Theorem 6.2. (Value preservation)[16].

Let $\Gamma_0 \rightarrow^* \Gamma \rightarrow^* \Gamma'$.

$$\text{Then : } \quad \text{val}_\tau(\Gamma') = \begin{cases} \text{val}_\tau(\Gamma) & \text{if } \tau \text{ occurs in } \Gamma \text{ and it is not burnt in } \Gamma \rightarrow^* \Gamma' \\ 0 & \text{otherwise} \end{cases} \quad (6.15)$$

Theorem 6.3 establishes that the transition systems \rightarrow and \Rightarrow are deterministic: crucially, this allows reconstructing the blockchain state from the transition log.

Notably, by item 3 of Theorem 6.3, witnesses only determine whether a state transition happens or not, but they do not affect the new state. This is unlike Ethereum, where arguments of function calls in transactions may affect the state.

Theorem 6.3. Theorem 3 (Determinism) [16]

For all $\lambda \in \{\surd, \mathcal{T}\}$ and $l \in \{\surd, w\}$:

1. if $\Gamma \xrightarrow{\lambda} \Gamma'$ and $\Gamma \xrightarrow{\lambda} \Gamma''$, then $\Gamma' = \Gamma''$;
2. if $(\Gamma, \mathbf{K}) \xrightarrow{l} (\Gamma', \mathbf{K}')$ and $(\Gamma, \mathbf{K}) \xrightarrow{l} (\Gamma'', \mathbf{K}'')$, then $(\Gamma', \mathbf{K}') = (\Gamma'', \mathbf{K}'')$;
3. if $(\Gamma, \mathbf{K}) \xrightarrow{\mathbf{W}:\mathcal{T}} (\Gamma', \mathbf{K}')$ and $(\Gamma, \mathbf{K}) \xrightarrow{\mathbf{W}':\mathcal{T}} (\Gamma'', \mathbf{K}'')$,
then $\Gamma' = \Gamma''$ and $\mathbf{K}' = \mathbf{K}'' = \mathbf{K}$;

6.7. Designing secure smart contracts in Algorand

What is important on a blockchain is the capacity to develop very efficient and secure smart contract. Exploit now the formal model to design some archetypal smart contracts, and establish their security. In this model all smart contracts presented from now on are stateless smart contracts.

Before starting is important to introduce the attacker model.

Attacker model

Assume that cryptographic primitives are secure, i.e., hashes are collision resistant and signatures cannot be forged (except with negligible probability). A run \mathcal{R} is a (possibly infinite) sequence of labels $l_1 l_2 \dots$ such that $(\Gamma_0, \mathcal{K}_0) \xrightarrow{l_1} (\Gamma_1, \mathcal{K}_1) \xrightarrow{l_2} \dots$, where Γ_0 is the initial state, and $\mathbf{K}_0 = \emptyset$ is the initial (empty) knowledge; hence, each label l_i in a run \mathcal{R} can be either w (broadcast of a witness bitstring w), $\mathbf{W} : \mathcal{T}$ (atomic group of transactions

\mathcal{T} authorized by \mathbf{W}), or \surd (advance to next round).

Consider a setting where:

- each user \mathbf{a} has a *strategy* Σ , i.e. a PPTIME algorithm to select which label to perform among those permitted by the ASC1 transition system. A strategy takes as input a finite run \mathcal{R} (the past history) and outputs a single enabled label l . Strategies are *stateful*: users can read and write a private unbounded tape to maintain their own state throughout the run. The initial state of \mathbf{a} 's tape contains \mathbf{a} 's private key, and the public keys of all users;
- an *adversary* \mathbf{Adv} who controls the scheduling with her stateful *adversarial strategy* Σ_{Adv} : a PPTIME algorithm taking as input the current run \mathcal{R} and the labels output by the strategies of users (i.e., the steps that users are trying to make). The output of Σ_{Adv} is a single label l , that is appended to the current run. They assume the adversarial strategy Σ_{Adv} can delay users' transactions by at most δ_{Adv} rounds, where δ_{Adv} is a given natural number.

A set Σ of strategies of users and \mathbf{Adv} induces a distribution of runs; the authors say that run \mathcal{R} is conformant to Σ if \mathcal{R} is sampled from such a distribution. Assume that infinite runs contain infinitely many \surd : this *non-Zeno condition* ensures that neither users nor \mathbf{Adv} can perform infinitely many transactions in a round.

Smart Contracts

See now how exploit the model to specify some archetypal ASC1 contracts, and reason about their security. To simplify the presentation, the authors assume $\delta_{Adv} = 0$, i.e., the adversary \mathbf{Adv} can start a new round (performing \surd) only if all users agree. The Table 6.1 summarises the selection of smart contracts, highlighting the design patterns they implement.

Analyse only some of the contacts presented.

Oracle. Start by designing a contract which allows either \mathbf{a} or \mathbf{b} to withdraw all the *Algos* in the contract, depending on the outcome of a certain boolean event. Let \mathbf{o} be an oracle who certifies such an outcome, by signing the value 1 or 0. Model the contract as the following script:

Oracle \triangleq $tx.type = close$ and $tx.asst = Algo$ and $((tx.fv > rmax$ and $tx.rcv = a)$
 or $(arg(0) = 0$ and $versig(arg(0), arg(1), o)$ and $tx.rcv = a)$
 or $(arg(0) = 1$ and $versig(arg(0), arg(1), o)$ and $tx.rcv = b)$)

Use case / Pattern	Signed witness	Timeouts	Commit/reveal	State machine	Atomic transfer	Time windows
Oracle	✓	✓				
HTLC		✓	✓			
Mutual HTLC		✓	✓		✓	
$O(n^2)$ -collateral lottery		✓	✓		✓	
0-collateral lottery		✓	✓	✓	✓	
Periodic payment						✓
Escrow	✓			✓		
Two-phase authorization		✓		✓		✓
Limit order		✓			✓	
Escrow		✓			✓	

Table 6.1: selection of smart contracts and the design patterns implemented in [16]

Once created, the contract accepts only close transactions, using two arguments as witnesses. The argument $arg(0)$ contains the outcome, while $arg(1)$ is \mathbf{o} 's signature on $(Oracle, arg(0))$, i.e., the concatenation between the script and the first argument. The user \mathbf{b} can collect the funds in *Oracle* if \mathbf{o} certifies the outcome 1, while \mathbf{a} can collect the funds if the outcome is 0, or after round r_{max} . Theorem 6.4 below proves that Oracle works as intended. To state it, T_p is defined as the set of transactions allowing a user \mathbf{p} to withdraw the contract funds:

$$T_p = \{t \mid t.type = close, t.snd = Oracle, t.rcv = p, t.asst = Algo\}$$

The theorem considers the following strategies for \mathbf{a} , \mathbf{b} , and \mathbf{o} :

- Σ_a : wait for $s = sig_o(Oracle, 0)$; if s arrives at round $r \leq r_{max}$, then immediately send a transaction $t \in T_a$ with $t.fv = r$ and witness 0 s ; otherwise, at round $r_{max} + 1$, send a transaction $t \in T_a$ with $t.fv = r_{max} + 1$;
- Σ_b : wait for $s' = sig_o(Oracle, 1)$; if s' arrives at round r , immediately send a transaction $t \in T_b$ with $t.fv = r$ and witness 1 s' ;
- Σ_o : do one of the following: (a) send \mathbf{o} 's signature on $(Oracle, 0)$ at any time, or (b) send \mathbf{o} 's signature on $(Oracle, 1)$ at any time, or (c) do nothing.

Theorem 6.4. *Let \mathcal{R} be a run conforming to some set of strategies Σ , such that: (i) $\Sigma_o \in \Sigma$; (ii) \mathcal{R} reaches, at some round before r_{max} , a state $Oracle[\sigma] \parallel \dots$; (iii) \mathcal{R} reaches the round $r_{max} + 2$. Then, with overwhelming probability:*

1. if $\Sigma_a \in \Sigma$ and \mathbf{o} has not sent a signature on (Oracle , 1), then \mathcal{R} contains a transaction in T_a , transferring at least $\sigma(\text{Algo})$ to \mathbf{a} ;
2. if $\Sigma_b \in \Sigma$ and \mathbf{o} has sent \mathbf{a} signature on (Oracle , 1) at round $r \leq r_{max}$, then \mathcal{R} contains a transaction in T_b , transferring at least $\sigma(\text{Algo})$ to \mathbf{b} .

Notice that in item 1, the only assumption that the authors have done is that \mathbf{a} and \mathbf{o} use the strategies Σ_a and Σ_o , while \mathbf{b} and Adv can use any strategy (and possibly collude). Similarly, in item 2 we are only assuming \mathbf{b} 's and \mathbf{o} 's strategies.

6.8. Conclusion

What has been presented in this chapter is a formal model to develop secure and efficient smart contract. Besides modelling the behaviour of transactions has been proposed a model of attackers: this enables to prove properties of smart contracts in the presence of adversaries, in the spirit of longstanding research in the cryptography area.

Besides not modelling the consensus protocol, to keep the formalization simple, the authors or the model chose to abstract from some aspects of ASC1, which do not appear to be relevant to the development of (the majority of) smart contracts. First, some transaction fields are not modelled: among them, they have omitted the fee field, used to specify an amount of Algos to be paid to nodes, and the note field, used to embed arbitrary bitstrings into transactions. They associate a single manager to assets, while Algorand uses different managers for different operations (e.g., the freeze manager for frz /unfrz and the clawback manager for rvk). It is used two different transactions types, pay and close, to perform asset transfers and account closures: in Algorand, a single pay transaction can perform both. Note that the same effect can be achieved by performing the pay and close transactions within the same atomic group. Although Algorand relies on 7 transaction types, the behaviour of some transactions needs to be further qualified by the combination of other fields (e.g., freeze and unfreeze are obtained by transactions with the same type afrz, but with a different value of the AssetFrozen field). While this is useful as an implementation detail, the presented model simplifies reasoning about different behaviours by explicitly exposing them in the transaction type. In the same spirit, while Algorand uses different transaction types to represent actions with similar functionality (e.g., transferring Algos and user-defined assets are rendered with different transaction types, pay and axfer), now it is used the same transaction type (e.g., pay) for such actions. The model does not encompass some advanced features of Algorand, e.g.: rekeying accounts, key registration transactions (keyreg), some kinds of asset configuration transaction (e.g.,

decimals, default frozen, different managers), and application call transactions. The script language substantially covers TEAL opcodes with *LogicSigVersion* = 1, but for a few exceptions, e.g. bitwise operations, different hash functions, jumps.

7 | Secteal

In this chapter we present our contributed extension to Secteal. First, we describe how to translate a smart contract from the formalized language to TEAL. The core of this section is the detailed explanation of the extension of the compiler used to construct safe stateless smart contracts. To reach this goal we have implemented Secure Functions that check if the contract is *safe* and in case adjust it interacting directly with the user.

7.1. From the formal model to concrete Algorand

Before presenting the compiler, we detail the procedure to translate transactions and scripts (i.e. smart contract, defined in chapter 6) in previous presented model to concrete Algorand as explained in [16]. We first sketch how to compile our scripts into TEAL. The compilation of most constructs is straightforward. For instance, a script $e + e'$ is compiled by using the opcode `+` and recursively compiling the two expressions e and e' . Analogously it is done for the other arithmetic and comparison operators, and for the cryptographic primitives. The logic operators *and*, *or* are compiled via the opcode `&&` and `||`. The *not* operator is compiled via the opcode `!`. The operator $txid(n)$ is compiled as `gtxn n TxID`, $txlen$ is compiled as `global GroupSize`, $txpos$ is compiled as `txn GroupIndex`, and $arg(n)$ as `arg n`. Finally, compiling the script $tx(n).f$ depends on the field f . If f is *fv*, *lv*, or *lx*, then the compilation is `gtxn n i`, where i is, respectively, *FirstValid*, *LastValid*, or *Lease*. For the other cases of f , the compilation of $tx(n).f$ generates a TEAL script which computes f by decoding the concrete Algorand transaction fields, and making them available in the scratch space. This decoding is detailed in Table 2 Appendix C of [16]. From the same table we can also infer how to translate transactions in the model to concrete Algorand transactions. For instance, translating a transaction of the form:

$$type : close, snd : x, rcv : y, asst : \tau \tag{7.1}$$

results in the concrete transaction (where we omit the irrelevant fields).

Type	pay
snd	x
close	y
amt	0

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

Table 7.1: Left table τ : Algo, Right table τ : Asset

7.2. SecTeal

The modelling approach is supported by a just implemented prototype tool, called **Secteal** (secure TEAL), and accessible via a web interface at:

<http://bitly.ws/ueJK>

In its current form, SecTeal supports experimentation with the proposed model, and it is provided with a series of examples. Users can also compile their own SecTeal contracts, paving the way to a declarative approach to contract design and development. SecTeal is a first building block toward a comprehensive IDE for the design, verification, and deployment of contracts on Algorand.

The actual architecture of Secteal is represented in Figure 7.1.

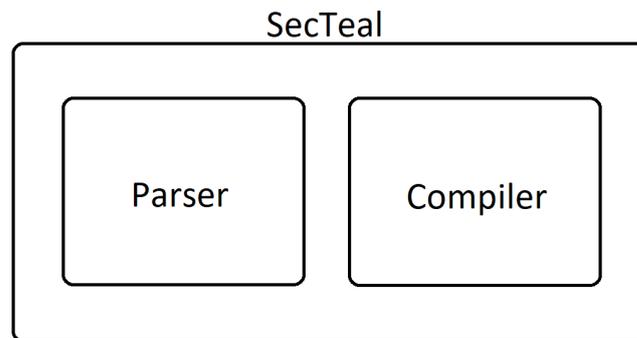


Figure 7.1: SecTeal Architecture

To reach these goals we need to implement some support functions to check the safeness of the contract and adjust it in case of error. 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 [16] 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 contract 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 tests 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 contract in an extreme secure way and it is ready to be deployed on the Algorand blockchain. The SecTeal architecture after our contribution is represented in Figure 7.2.

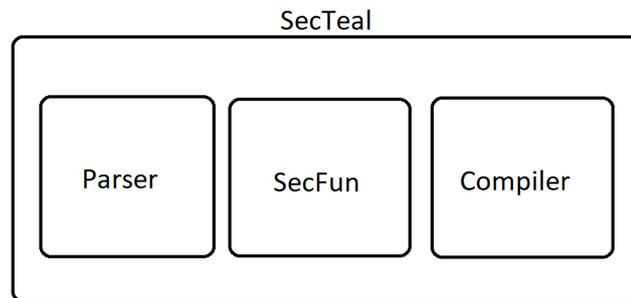


Figure 7.2: Extended SecTeal Architecture

7.3. Extended SecTeal

The Parser block remains unchanged from the original version. This part has been implemented using ANTLR [43]. ANTLR (ANOther Tool for Language Recognition) is a tool for processing structured text. It does this by giving us access to language processing primitives like lexers, grammars, and parsers as well as the runtime to process text against them. In our case the Parser returns a tree-like structure represented as a list P (Definition 7.4.2).

The compiler instead, has been modified in order to allow it to work for all the stateless

smart contracts defined by the formal model [16] and Algorand [8]. First, we extend the syntax and we slightly modify the implemented code according to our updated syntax. Also, the compiler is not able to catch the type of the input contract in Secteal and translate it in the corresponding Teal one. To solve this problem a lot of work has been done in section 7.4 then we adapt the compiler in order to works properly. Moreover, to simplify the solution of that problem we take in account the following transaction types:

$$\text{Type} : \{ \text{Pay}, \text{PayAsset} (\text{PayA}), \text{Close}, \text{Close Asset} (\text{CloseA}), \text{Gen}, \text{Optin}, \\ \text{Revoke}, \text{Freeze}, \text{Unfreeze}, \text{Burn}, \text{Delegate}, \text{Rekey} \}$$

Each transaction type has a proper structure that uniquely identified itself (It is described in section 7.4). From now, every transaction type considered in this section belongs to *Type*. We make also a further division in the *Type* class with respect to the transactions class of the formal model in Table A.2. (*Pay* with *Pay asset* and *Close* with *Close asset*). This gives more clarity to the model and increase the safeness of the contract. Considering this slight change the classification of transaction type is more similar to the Algorand one. Moreover, we have introduced *Rekey* transaction type and also the *rekey* parameter according to the new development of Algorand. The new block presents in Figure 7.2 is called *SecFun*. Here we implement a validation mechanism with the followings three main purposes: check the correctness and safeness of the contract and adjust it in the right way.

7.4. SecFun

The first concept which need to be defined is the safeness of smart contracts. In this work we define a Safe smart contract as follows.

Definition 7.4.1. Safe Smart Contract Consider a transaction with a specific type. A Smart Contract is Safe if is impossible using that contract to validate transaction that could belong to a different type.

Note that this notion of Safe contract holds only if the parameter type is present. Consider an example on the Algorand blockchain. Suppose that user **A** sends a 5 Algos to user **B**. It is possible to occur in an operational error and send a close account transaction instead

of a payment transaction.¹ A Teal smart contract in order to be safe must specify that the type of the transaction should be equal to pay, the *CloseReaminderTo* and *RekeyTo* parameter must be equal to the *Null Address*. These conditions ensure that the only possible transaction accepted by the contract is the Algo payment one.

As explained in Figure 7.2, the procedure is sequential, the output of the block in the left is the input of the next one. For this reason the input structure that should be analysed is not the original contract written by the user but the output of the Parser P , which is defined as follow.

Definition 7.4.2. P is a tree-like structure represented as a list defined by $[\dots]$ P can contain three kind of elements:

- *op*: operator (e.g. $+$, $=$, *not*, ...)
- *exp*: expression (e.g. 0 , 1 ...)
- *nested bracketed expressions* (P)

Expression is a combination of variables and constants (Table A.7), joined by operators. The expression, the operator and the grammar of SecTeal language are defined in Table A.6 and Table A.5. Also in Table A.8 and Table A.9 are described all the fields (e.g. *snd*, *rcv*...) and the corresponding Teal translations of all the elements of SecTeal. The collection of *exp* and *op* defines the lexicon of *Extended Secteal*.

Before we state our definitions of what we meant for '*Correct*' and '*Wrong*' contracts, it is important to introduce the *Gold Standard Tables*

Gold Standard Tables

For every type of transactions there is a GS-Table (Gold-Standard), each of which describes precisely required parameters (e.g. FIELD). As already explained, such parameters are needed to correctly form a safe stateless smart contract once its intended type has been identified. Each table has the following columns :

1. **Param**: main parameters that characterise a stateless smart contract with a specific type.
2. **Category**: parameters belonging to these classes are either:

¹This is possible on Algorand because payment and close transaction belongs to the same type transaction class. For this reason using the CLI *goal* for the transaction, the difference between these two transactions is only the presence of a precise parameter.

- (a) Necessary (**): parameters/conditions that must be specified to have a safe contract. They should be added in a second moment in case they are present.
- (b) Not Necessary(*) : parameters/conditions that are not necessary to have a safe contract.

If a parameter of this class misses, a *warning message* is sent to the user. But the user could also decide not to add them.

This category contains a further partition: *Required*(\top) and *Default*(\perp).

The *Default* parameters are common to every transaction instead the others are specific for each type. For example freeze manager parameter characterises the asset freeze transaction but not the payment transaction. Fee instead is valid for all transactions. Moreover, the parameters in the *Default* class have just various constraints, assigned by the syntax of Algorand, that limit their possible values. Instead this is not the same for the *Required* parameters. For example, in that case a possible error is to send money to the wrong account. This error is not possible to be checked automatically by Algorand network. The only way is to specify this condition in the smart contract. For this reason if one of the *Required* parameter misses a *warning message* is reported to user. If one of the *Default* parameter is not explicitly expressed nothing is done.

3. **Condition:** In this column, we find the values or type conditions that parameters must assume to have a safe contract. This class presents a further division: *Type* and *Value Condition*. *Type Condition* could be either *String* or *Integer* or *Address*.

Definition 7.4.3. Address We implement the Class Address with the following characteristics:

- *Alphanumeric string*
- *58 Characters*
- *Capital characters*

Value Condition is instead the specific value assigned to the parameter.

If these conditions are not respected we have different errors, *Type* and *Value error*. We now state the following definitions.

Definition 7.4.4. Type Error A *Type Error* is an error that we get if the type associated to the parameter is different to the one defined by the model (all the type parameters are

presented in the Table B.1) or if the parameter type is not one indicated by the corresponding GS-Table.

In the Example 7.4.1 we show two different type errors. The first is wrong because the type associated to the value of the *fee* is an address instead of an integer (Table B.1). The error in the example (2) is about the type of the value assigned to the parameter *fee*. It indicates an address instead this parameter can be assigned only to the integer value.

Example 7.4.1. Type error

Consider the transaction $tx(0)$ and the field *fee*.

(1): $tx(0).fee = addr\ 100$

(2): $tx(0).fee = int\ Null\ Address$

Definition 7.4.5. Value Error We incur in a Value Error if the parameter does not respect the specific values assigned by the corresponding GS-Table.

Example 7.4.2. Value error

Consider the close transaction $tx(0)$ and the field *crcv* (close receiver). The condition specified by the GS-Table B.4 for this type of transaction is that the necessary parameter *crcv* must be different to the Null Address. A value error occur if we write the following condition on the contract: $tx(0).crcv = addr\ Null\ Address$

In this case the contract is not safe.

Below are some of the GS-Table used in the Secteal model, in particular the table for a *Pay* and *Pay asset transaction*. For the complete list of tables see Appendix B.

PAY:

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 7.2: PAY GS-TABLE

 PAY ASSET:

Param	Category	Type	Value
TYPE	\top	String	
ASSTRCV	\top	Address	
ASST	\top	Integer	
SND	**	Address	!= Null Address
ASSTSND	**	Address	Null Address
ASSTVAL	**	Integer	!= Null
CASSTRCV	**	Address	Null Address
FV	\perp	Integer	
LV	\perp	Integer	
LX	\perp	Integer	
NOTE	\perp	String	
FEE	\perp	Integer	

Table 7.3: PAY A *GS-TABLE*

7.4.1. Algorithm

In this section we explain in detail the implementation of *SecFun* algorithm. It is based on the following assumptions. First of all it works only for stateless smart contracts. The algorithm is compatible with all the transaction type showed in the presented model [16] and also including `payA`, `closeA` and `rekey`. Lastly, we consider only single (non-atomic) transactions.

The function `SECFUN` takes in input a SecTeal parsed program P . The algorithm returns either:

1. a series of "corrected" tables according to the *GS-tables*, one for each type of transaction identified in P ;
2. warnings and errors to the users, prompting for corrections and re-compiling;
3. a statement where P is declared *unsatisfiable*

The algorithm output can be thought as the equivalent of the intended and corrected meaning of the smart contract, in form of a set of tables, one representing each transaction identified in the program, and corrected according to *GSs* (when possible). This can be thought as a canonic form of the contract in terms of a *sum of products*. The sum is the possible alternative transactions that can be accepted using that contract and the products are all the conditions that *need* to be fulfilled for each accepted form of transaction.

More in details, the algorithm is composed of four main parts executed in sequence. First we have the **Parse** of P for collecting information, the **Match** and **Correction** for checking the correctness of the contract and in case, fix the errors. Finally the **Compile** part for the final Teal translation. In the next section we describe in details each of these methods.

7.4.2. Parsing procedure

At the beginning, **Parse** checks the whole input structure and collects information related to the transaction, as listed in the Table A.8. The algorithm parses P as a Tree. Each node of this tree contains the following structure e.g.

1. $[op, [exp], [exp]]$ or
2. $[op, [exp]]$ or
3. $[exp]$

Where:

1. is a propagation node if $op = and$ or $op = or$
 - There are two links: left and right.

otherwise it is an leaf node

 - then no link starts from this node.
2. is an leaf node
3. is an leaf node.

Parsing the tree, the algorithm collects all the transaction parameter presented in an leaf node.

We now show an example to clarify the whole processes.

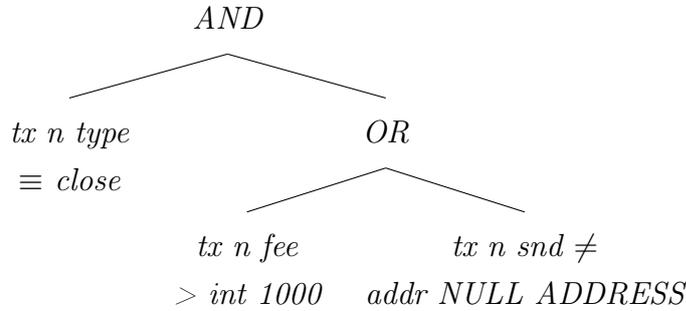
Example 7.4.3. Consider the following P :

$$[and, [=, [tx, n, type], [int, close]], [or, [>, [txn, n, fee], [int, 1000]],$$

$$[\neq, [tx, n, snd], [addr, Null Address^2]]]$$

²Is an address that represents a blank byte array. It is used when you leave an address field in a transaction blank.

This can be represented using a Tree:



In order to parse the Tree (e.g. the P) we chose the **top-down parsing strategy**.

Starting from the top (in this case 'AND'), we then move to the left :

- (1) if it is an leaf node then, we pick the parameter and inspect the right node.
- (2) otherwise, we move deep to the left child node.

The function continues with (1) and (2) procedures until all nodes have been inspected.

Once the algorithm parses P and picks the parameters, they are collected in the list of tables \mathcal{TT} .

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

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

Example 7.4.4. For example, the parameter of the following P structure

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

will be collected in the T-tables

$$[A_c, C_e] \quad [A_c, B_i]$$

where subscripts are addresses of the associated condition in P , according to standard tree enumeration. Instead uppercase are the parameters.

Each node contains a subset of \mathcal{TT} e.g. \mathcal{TT}_k and the corresponding P structure, as displayed in Example 7.4.3.

Let us follow now the standard enumeration of a Tree and suppose that at non-leaf node k^{th} we have $\mathcal{TT}_k = \mathcal{TT} = \{TT_i\}$ as current list of tables. The two possible outcomes are based on the value of the operator op .

Case 1 If $op = or$ then the algorithm copies all the parameters collected in TT_i into $TT_{i_{right}}$ and $TT_{i_{left}}$. From now on, information below this node are separated into $TT_{i_{right}}$ and $TT_{i_{left}}$ according to the left and right sub-tree.
 \mathcal{TT} from now on is

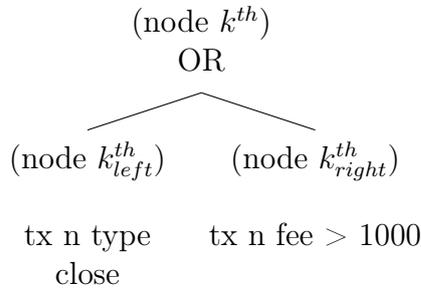
$$\mathcal{TT} = \{TT_{i_{left}}, TT_{i_{right}}\}$$

Case 2 If $op = and$ then the algorithm continues to propagate all information that will be collected from the left and right sub-tree into TT_i while \mathcal{TT} remains the same.

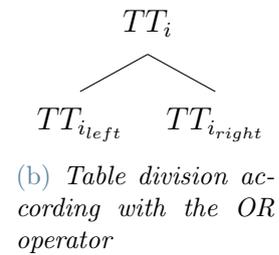
Consider these simple examples:

Example 7.4.5. Consider that the P structure corresponding to the node K^{th} is the following: $[OR, [=, [tx, n, type], [int, close]], [>, [tx, n, fee], [int, 100]]]$.

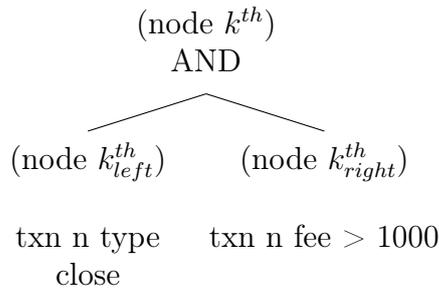
We show the Tree representations of the P structure at node k^{th} and the tables division procedure, according with the OR and AND operators. Moreover, the tables on the right indicates the table associated to each node.



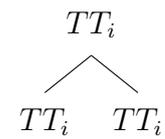
(a) Tree representation of P in node k^{th}



(b) Table division according with the OR operator



(a) Tree representation of P in node k^{th}



(b) Table division according with the AND operator

In the more complex case instead, namely when the list of tables at node k^{th} is of the form

$$\mathcal{TT}_k = \mathcal{TT} = \{TT_i, TT_j, \dots, TT_n\}$$

where $N = \{i, \dots, n\}$ is a finite set s.t. $\max_{i \in N} i < \infty$ and $i \neq j \forall i, j \in N$. In this case the same procedure explained above should be replicated for all the tables in \mathcal{TT} . The final result will be:

Case 1 $\mathcal{TT} = \{TT_{i_{left}}, TT_{i_{right}}, \dots, TT_{n_{left}}, TT_{n_{right}}\}$

Case 2 $\mathcal{TT} = \{TT_i, TT_j, \dots, TT_n\}$

In case, at node k^{th} , $\mathcal{TT}_k \subsetneq \mathcal{TT}$ then the same procedures hold only for the tables in \mathcal{TT}_k .

Consider now the leaf node k_{left}^{th} . If in this node there is some transaction information then it is collected in $TT_j \forall j$ s.t. $TT_j \in \mathcal{TT}_{k_{left}}$. The same procedure holds for the right side.

Continue to consider the previous example 7.4.5. Consider the situation where $\mathcal{TT}_k = \{TT_i\}$ and suppose that TT_i is empty. Once the algorithm has parsed the whole sub-tree of node k^{th} , we have the following tables respectively in OR and AND cases of example 7.4.5.

<table border="1" style="border-collapse: collapse; width: 100px; height: 30px;"> <tr><td colspan="2" style="text-align: center;">$TT_{i_{left}}$</td></tr> <tr><td style="width: 50%;">Type</td><td style="width: 50%;">close</td></tr> </table>	$TT_{i_{left}}$		Type	close	<table border="1" style="border-collapse: collapse; width: 100px; height: 30px;"> <tr><td colspan="2" style="text-align: center;">$TT_{i_{right}}$</td></tr> <tr><td style="width: 50%;">fee</td><td style="width: 50%;">1000</td></tr> </table>	$TT_{i_{right}}$		fee	1000	<table border="1" style="border-collapse: collapse; width: 100px; height: 30px;"> <tr><td colspan="2" style="text-align: center;">TT_i</td></tr> <tr><td style="width: 50%;">Type</td><td style="width: 50%;">close</td></tr> <tr><td style="width: 50%;">fee</td><td style="width: 50%;">1000</td></tr> </table>	TT_i		Type	close	fee	1000
$TT_{i_{left}}$																
Type	close															
$TT_{i_{right}}$																
fee	1000															
TT_i																
Type	close															
fee	1000															
(a) <i>Case OR</i>	(b) <i>Case OR</i>	(c) <i>Case AND</i>														

Figure 7.3: Table representation of OR and AND Tree of example

These set of tables contain all information in the sub-tree under the node k^{th} . For this reason in case of OR operator each new table is independent from the other. In this sense every table in \mathcal{TT} represents an independent set of conditions that must hold at the same time and we should check the correctness of each of them.

7.4.3. Matching procedure

Once the tables are completed we want to check if they are 'Correct'. Before studying the algorithm, we formalise the following definitions.

Definition 7.4.6. Suitable match Let $\mathcal{TT} = \{TT_n, \dots, TT_i, \dots, TT_m\}$ be a list of tables and consider two tables TT_i from \mathcal{TT} and GT_j from GS respectively. TT_i and GT_j suitably match if every parameter in TT_i is also present in GT_j and all the type and value conditions

are satisfied.

Definition 7.4.7. Wrong Table Let $\mathcal{TT} = \{TT_n, \dots, TT_i, \dots, TT_m\}$ be a list of tables and consider the table TT_i from the list \mathcal{TT} .

TT_i is Wrong if it does not suitably match with any table GS_j of \mathcal{GS}

Definition 7.4.8. Incomplete Table Let $\mathcal{TT} = \{TT_n, \dots, TT_i, \dots, TT_m\}$ be a list of tables and consider a table TT_i from this list \mathcal{TT} . TT_i is Incomplete if $\exists j$ s.t.

- $GS_j \in \mathcal{GS}$
- GS_j and TT_i suitably match
- \exists at least one Necessary parameter in GS_j that is not present in TT_i .

Definition 7.4.9. Correct Table Let $\mathcal{TT} = \{TT_n, \dots, TT_i, \dots, TT_m\}$ be a list of tables and consider the table TT_i from this list. TT_i is correct if it is not Wrong and it is not Incomplete.

Once we have completely parsed P and collected all information into \mathcal{TT} , we have to make sure that all of the tables are correct and safe. To do so, we simply match each table in \mathcal{TT} with our benchmark for a safe stateless smart contract (GS -Table).

We denote the match between TT_i and GS_j as $match(TT_j, GS_i)$. The *match* process is the comparison of types and values of the parameters of the two tables.

While matching TT_j in \mathcal{TT} with each GS – Table, we follow two different procedures depending on whether *Type* parameter belongs to TT_j .

First, consider $Type \in TT_j$. If a matching exists, it is *univocally determined* (e.g. $\forall j \exists!$ i s.t. TT_j and GS_i suitably match). In this case the outcome of the $match(TT_j, GS_i)$ is positive. This means that all the parameters respect the type condition and the value condition. Otherwise, if TT_j and GS_i do not match, the output is negative and the branch of the contract is *Wrong*. This happens either if some parameter in TT_j does not respect a type or value condition in GS_i or some parameter in TT_j is missing in GS_i . In the first case the output will be an Error message, while for the second case the output is an Warning message. The reason behind this is that the former corresponds to an *type* or *value error*, while the latter does not.

7.4.4. Correction procedure

In case the two tables *suitably match* then TT_j is either *Correct* or *Incomplete*. If some necessary condition misses then we must add it to the table and also modify P accordingly.

In order to really adjust the contract we should directly modify P . Note that from the table we know which conditions are missing on each branch of the contract. We add them to P by modifying its corresponding tree structure. We scan the tree from the root with the aim of adding a node, which means substituting a node with a propagation node, namely adding a sibling to the node with the new condition. Since the new condition and the rest of the tree need to be fulfilled simultaneously, with use an *AND* link.

We show now this procedure in details with the Example 7.4.6 and Example 7.4.7.

Example 7.4.6. Let P be the following input structure:

$$[and, [=, [tx, n, type], [int, pay]], [or, [=, [txn, n, crcv], [addr, Null Address]], [[=, [txn, n, rekey], [addr, Null Address]]]$$

We can recreate the representation of the tree and the tables. To each node of the tree is associated the P parameter and the enumeration of the node.

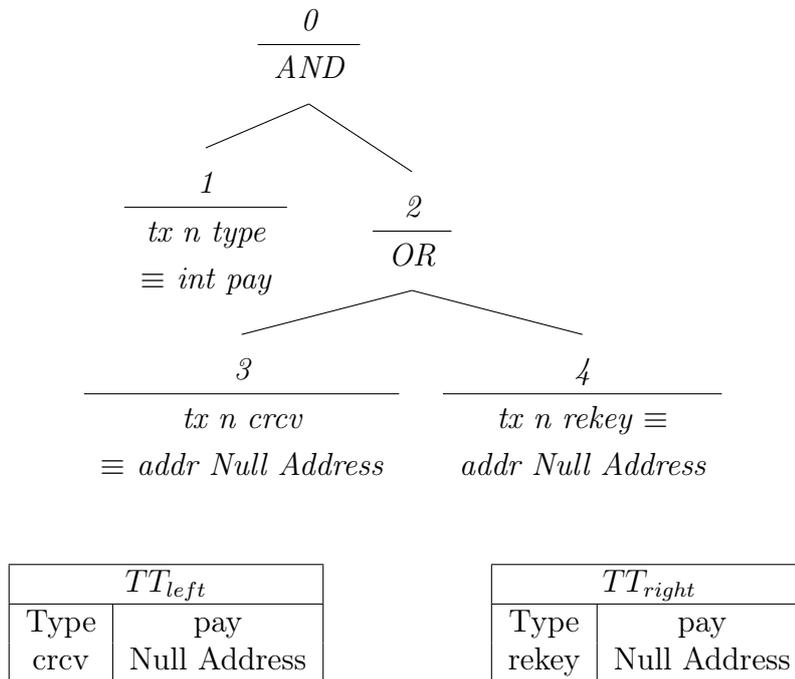


Figure 7.4: Table representation of P parameters

According to the Table B.2, TT_{left} and TT_{right} are both correct but not safe because the left table does not contain the rekey parameter and the right one does not contain the crcv parameter that is instead listed in Table B.2. To fix the TT_{left} table, we should add the rekey parameter. This cannot be added in the node 0 because it would also change the other table. The same reasoning applies to node 1 and 2. Therefore, we add the condition

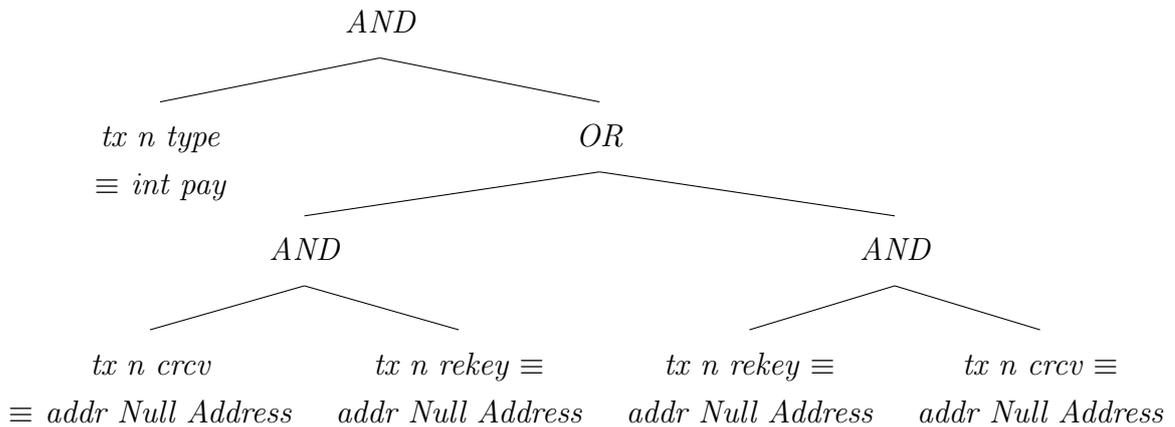
at the node 3 as it belongs only to the left branch (e.g. table). The same procedure is followed also for the *crvc* parameter in the right branch.

Finally, in Example 7.4.7 we show the final result of Example 7.4.6.

Example 7.4.7. Consider *P-Adjusted*:

$[and, [=, [tx, n, type], [int, pay]], [or, [and, [=, [txn, n, crcv],$
 $[addr, Null Address]], [=, [txn, n, rekey], [addr, Null Address]]],$
 $[and, [=, [txn, n, rekey], [addr, Null Address]], [=, [txn, n, crcv], [addr, Null Address]]]$

The tree representation and the tables are the following:



TT_{left}	
Type	pay
crvc	Null Address
rekey	Null Address

TT_{right}	
Type	pay
rekey	Null Address
crvc	Null Address

Figure 7.5: Table representation of *P* parameters

In this example, the missing conditions are added at the leaf node containing the parameter that is contained only in one table.

It is important to highlight that there is always a node where will be good to adjust the contract. Indeed each table has at least a parameter that belongs only there and the node that contains it is called *Unique node*. We want also have the possibility to adjust the contract before visiting the *Unique node*.

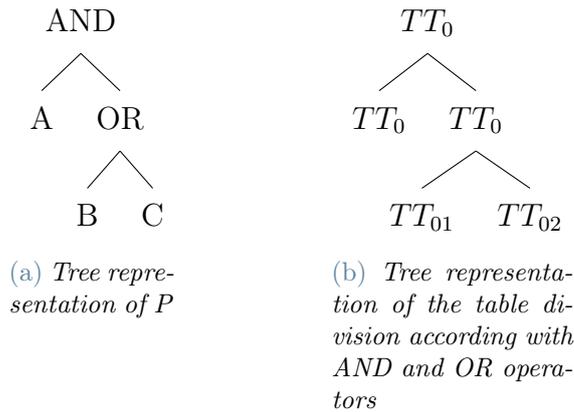
Before explain how it is possible , it is important to introduce a further feature of the table. Every table has a code which is a sequence containing all the nodes visited by the

algorithm and the architecture of the path followed before the initialisation of that table. This code is unique for each table and we call it *ID Code*. Suppose to choose to add the condition at a node k^{th} , this code give us the information about which branches and tables will be influenced by this change.

We present now Example 7.4.8 to help the understanding of how the ID Code is built.

Example 7.4.8. Consider $P: [and, [A], [or, [B], [C]]]$

In the tree 7.6a we have the tree representation of P . Following the procedure for the parse and the construction of the tables, we visit the leaves in the order A - B - C . The tree 7.6b show the associated table to each node. In this example, a table corresponds to each node. The subscripts represent the ID Codes. Until we inspect the node containing the parameter OR , we consider the same table TT_0 , as already described in 7.4.5. Here we should split it in two new tables TT_{01}, TT_{02} . The list of tables at the end of the parsing is $\mathcal{TT} = \{T_{01}, T_{02}\}$.



As showed in the example the encoding of the *ID Code* is the following. When we split the table the new two tables inherit the ID Code of the previous one. Then a new different digit is added to the both of them. In this way all the tables in \mathcal{TT} have an unique code.

Let us now explain more in detail how the algorithm works when it comes to the correction phase. As already explained before, we parsed P and collect all information in a list of tables \mathcal{TT} . Then, we parse again P in the **same order** of the first passage and reconstruct the tables in the same way. Recall that in every node there is the input structure P and the associated table identified by a code. While the algorithm visits each node, it takes the *ID Code* of the associated table which we call *IDNC* (e.g. ID Node Code) to differentiate it from the *ID Code* of the already built tables of \mathcal{TT} . From \mathcal{TT} , we select the tables which *ID Code* contains the IDNC (e.g. $0 \subseteq 01$). We call this sub-list of \mathcal{TT} , *Selected list* \mathcal{SL} . For each of the table in \mathcal{SL} we start the matching procedure detailed in subsection 7.4.3.

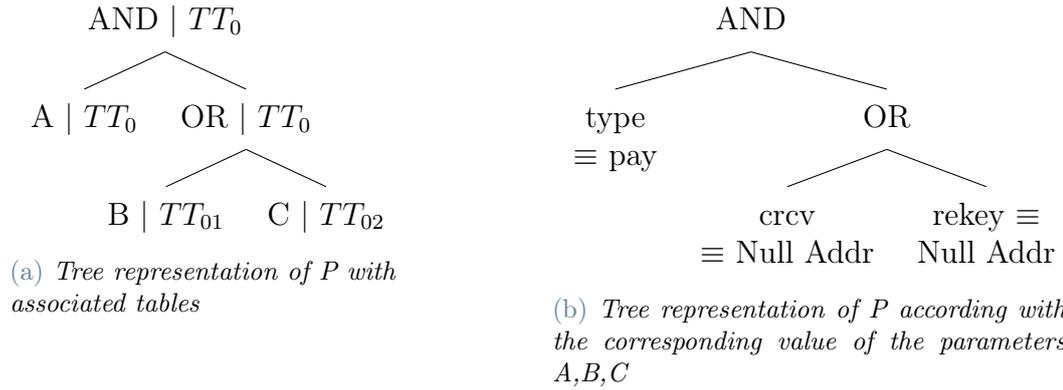
In case that table is incomplete we collect the missing necessary conditions. For each collected condition, we check if it is a missing condition for all the selected tables. In that case we add the condition to all the tables in \mathcal{SL} and also we modify P . Otherwise this condition is not added and we pass to the other one. At the end we follow the same procedure for all the nodes.

At the end all the *Incomplete* contracts became *Correct* and *Safe*.

Consider the following example to better understand how the algorithm works.

Example 7.4.9. Consider, as in the previous example, $P: [and, [A], [or, [B], [C]]]$

Following the same procedure, for parsing and construction of the table, of the example 7.4.8. Then we visit in order the leaf nodes A, B, C and the list of tables at the end of the parsing is $\mathcal{TT} = \{T_{01}, T_{02}\}$, where the subscripts are the ID Code of each table. Moreover, in the right tree we assign the parameters to the nodes as in the tree of Example 7.4.6.



On the left we have the representation tree of P with the associated table for each node. On the right tree each node has the explicit value of the parameters A, B, C . The tables TT_{01} and TT_{02} are the same of TT_{left} and TT_{right} of the Example 7.4.6.

As explained above, repeating the previous procedure, the first node that we visit is A . We see from tree 7.6a that the associated table has IDNC equal to 0. This IDNC is contained in the sequence of every table in \mathcal{TT} , so $\mathcal{SL} = \{TT_{01}, TT_{02}\}$. At this point, we match TT_{01} with $GS - Table$. We check that this table is *Incomplete* then we collect the *rekey* condition. Now we check if it is also a missing condition for TT_{02} and it is not. In this case the condition is not added to this node. We repeat the same procedure for TT_{02} and for the same reason we does not add the *crcv* condition to the first node. Following the same procedure for all the node with that associated table TT_0 we will have the same result. We visit then the node B . In this case $\mathcal{SL} = \{TT_{01}\}$. As already explained we match the table and collect the missing condition. Check if this condition is needed also for the other

tables, in this case there is only one, so we can add this condition to this node and to all the tables in \mathcal{SL} . Same procedure is followed for the node C . At the end we will have the same adjusted P and \mathcal{TT} of the example Example 7.4.7.

This tool is also able to modify instantaneously the *Type Errors* in the contract. The procedure in this case is less complex. While the algorithm parses P , if it finds that the type associated to the parameter is wrong (Example 7.4.1), it automatically fixes it according with the Table B.1. If the value associated to some parameter is not correct (Example 7.4.1), SecTeal asks the user to modify it with the right one.

7.4.5. Table without Type parameter

It is also possible that not all the tables have the parameter *Type*. In this case the procedure is slightly different. First we should formalise a new definition of *suitably match*.

Definition 7.4.10. Suitably match' Let $\mathcal{TT} = \{TT_n, \dots, TT_i, \dots, TT_m\}$ be a list of tables and consider two tables TT_i from \mathcal{TT} and GT_j from GS respectively. TT_i and GT_j *suitably match'* if every parameter in TT_i is also present in GT_j and all the type conditions are satisfied.

Also in this case we consider $match(TT_j, GT_i)$ positive if the tables *suitably match'*.

We introduce the variable Num to represent the number of GS_i which *suitably match'* with TT_j . There exist three possible situations:

If $Num = 1$ then there is an unique match. In this case the same procedure above ($Type \in TT_j$) holds with positive outcome.

If $Num = 0$ then $\forall i$ the output of $match(TT_j, GT_i)$ is negative, then TT_j is an "*undefined type branch*" of the contract. In that case *SecTeal* reports a warning message to the user who will decide to proceed without further checks or if it wants to add this condition.

The last possible case is when $Num > 1$, then the outcome is a list of all the possible GS_i at which this branch of the the contact TT_j can *suitably match'* with.

The user can choose one among them or None. If the user selects None, then the check and adjustment procedure are stopped for this branch of the contract. Otherwise we follow the same procedure used for the branches which have the parameter type.

If at the end of the *match e correction* procedures the contract is safe, we can start the last phase of SecTeal. The compiler receives a P structure adjusted in case of errors or with the necessary conditions. The compiler returns a Teal safe smart contract. Its implementation

is based on some recursive function on *P-adjusted* which collects the parameters, translates them according with the architecture and lexicon of Teal. Otherwise it displays the error to be solved before compiling.

7.5. Conclusion

The Compiler is an important tool in order to really implement smart contracts written in the new language. In this chapter we have detailed the syntax used to write a smart contract and the basic rules to respect in order to pass from Secteal to Teal. As explained in the previous section the model point out the basement for the implementation of safe smart contract. We have given the precise definition of what a Safe smart contract is and we have given an example to make it more clear. Using Algorand it is possible to send a transaction with type pay intending to pay 5 Algos to another user and wrongly close the account. It is not possible instead that this transaction is sent using the formal model. Moreover, writing a contract using Teal or Pyteal and deploying it directly on Algorand testnet there is the possibility that this contract accepts the transaction, instead using Secteal to build your smart contract this kind of error is impossible. To reach this purpose Secure functions are crucial. They indeed check if the contract is secure and works properly adjusting it interacting with the user.

8 | Test

In this chapter we test our extended compiler using various smart contract examples. First, we see how it works on smart contract for various type of transactions and then test the Secteal tool for more complex stateless contracts.

As detailed previously, we have various transaction types each one with different structure. The stateless smart contract for them must respect the conditions detailed in the *GS-Table*. Now we show how the compiler works for some of them. For the sake of comprehension, in these exmples we omit most of the parameters of the *Default* class (section B.2).

All the tests of this chapter are implemented in Python [1] using the Visual Studio Code editor [2]. All codes are in the Github repository of the author. In particular, the file *Steal.c* contains the compiler of [16], the file *compiler* contains the extended compiler; *Goldstand* contains all the GS-Tables and in the file *Test* there are all the stateless smart contract examples described in this chapter.

8.0.1. Close account transaction smart contract

In the case of close account transactions, as previously explained, a safe contract must respect all conditions and parameters presented in Table B.4.

Below, we display a *safe* smart contract for the indicated type. The *Secteal* contract is the input of the compiler and the output is the same contract written in *TEAL*.

In this contract there are all the necessary conditions and required parameters detailed in Table B.4. Notice that the contract specifies that the parameters CloseRemainderTO (crcv) and RekeyTO (rekey) are respectively equal and different from the *Null Address* (line 41-43 and 50-52) ¹. The condition for the val parameter is reported in line 22 for the Secteal contract and 47-49 for the Teal one. Moreover the Type (line 32-34) is not CLOSE (line 4) but PAY accordingly with Algorand transactions model. The required parameter are reported in lines 4,10,18 for SecTeal smart contract and in lines 38-40, 44-46 for the

¹Null Address is an address that represents a blank byte array. It is used when you leave an address field in a transaction blank.

Teal smart contract. Also we notice that all the operators are properly listed at the end of the Teal contract. In this way all the conditions of the contract must be respected from the transaction at the same time.

```

1 #INPUT: SECTEAL SMART CONTRACT
2
3 (
4   (tx(0).type = int close),
5     and
6   (
7     (tx(0).fee < int 1000),
8       and
9     (
10      (tx(0).rcv = addr
11        ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJB YAFZM)
12        and
13      (
14        (tx(0).crcv = addr
15          ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJB YAFZM)
16          and
17        (
18          (tx(0).snd = addr
19            LPHG7ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJB YKOY2)
20            and
21          (
22            (tx(0).val = int 0)
23              and
24            (tx(0).rekey = addr NULL ADDRESS)))))))))
25
26 #Output: TEAL SMART CONTRACT
27
28 CLOSE          <<<<
29
30
31 #pragma version 2
32 gtxn 0 Type
33 int pay
34 ==
35 gtxn 0 Fee
36 int 1000
37 <
38 gtxn 0 Receiver
39 addr ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJB YAFZM
40 ==

```

```

41 gtxn 0 CloseRemainderTo
42 addr ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBYAFZM
43 ==
44 gtxn 0 Sender
45 addr LPHG7ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBYKOY2
46 ==
47 gtxn 0 Amount
48 int 0
49 ==
50 gtxn 0 RekeyTo
51 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQ
52 ==
53 &&
54 &&
55 &&
56 &&
57 &&
58 &&

```

Now we show few other interesting examples of smart contract for various transactions.

8.0.2. Pay account transaction smart contract

PAY is a contract able to ensure that the pay transaction sent by an Algorand user to another one is correct. An example of Safe smart contract is the following.

From the script below it is possible to notice the presence of all the necessary conditions in the input SecTeal contract (lines 21 and 23). These conditions are correctly translated in Teal (lines 49-54). The required parameter are reported in the SecTeal contract in lines 10,14,17 with the respective Teal translation in lines 40-48. Also in this case all the operators are listed at the end of the Teal contract.

```

1 #INPUT: SECTEAL SMART CONTRACT
2
3 (
4   ( tx (0) . type = int pay )
5     and
6   (
7     ( tx (0) . fee < int 1000)
8       and
9     (
10      ( tx (0) . rcv = addr
11        ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBYAFZM )

```



```

56 &&
57 &&
58 &&
59 &&
60 &&

```

8.0.3. Pay Asset transaction smart contract

PAY Asset is a contract able to ensure that the pay asset transaction sent by an Algorand user is correct. In that case we are sending an asset, so we expect that the asset ID parameter is specified. From the script below is possible to notice the presence of all the necessary conditions in the input SecTeal contract (lines 17-27). These conditions are correctly translated in Teal (lines 49-60). The required parameter are reported in the SecTeal contract in lines 10,14 with the respective Teal translation in lines 43-48. Also in this case all the operators are listed at the end of the Teal contract.

```

1 # INPUT : SECTEAL SMART CONTRACT
2
3 (
4 (tx(0).type = int pay_ax )
5     and
6 (
7 (tx(0).fee < int 1000)
8     and
9 (
10 (tx(0).asstrcv = addr
11     ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBAYAFZM )
12     and
13 (
14 (tx(0).asst = int 185061)
15     and
16 (
17 (tx(0).snd = addr
18     5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3JFSNJJCAYAFZTSERT )
19     and
20 (
21 (tx(0).asstsnd = addr NULL ADD )
22     and
23 (
24 (tx(0).asstval = int 100)
25     and
26 (
27 (tx(0).casstrcv = addr NULL ADD )))))))

```

```
28
29
30 #OUTPUT: TEAL SMART CONTRACT
31
32 PAY ASSET          <<<
33
34
35
36 #pragma version 2
37 gtxn 0 Type
38 int axfer
39 ==
40 gtxn 0 Fee
41 int 1000
42 <
43 gtxn 0 AssetReceiver
44 addr ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFNSNJJBYAFZM
45 ==
46 gtxn 0 XferAsset
47 int 185061
48 ==
49 gtxn 0 Sender
50 addr 5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3JFSNJJCYAFZTSERT
51 ==
52 gtxn 0 AssetSender
53 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQ
54 ==
55 gtxn 0 AssetAmount
56 int 100
57 ==
58 gtxn 0 AssetCloseTo
59 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQ
60 ==
61 &&
62 &&
63 &&
64 &&
65 &&
66 &&
67 &&
```

8.0.4. Rekey account transaction smart contract

Rekeying is a powerful protocol feature which enables an Algorand account holder to maintain a static public address while dynamically rotating the authoritative private spending key(s). This is accomplished by issuing a "rekey-To transaction" which sets the authorized address field within the account object.

A rekey-to transaction is a payment type transaction which includes the RekeyTo parameter set to a well-formed Algorand address. Authorization for this transaction must be provided by the existing authorized address.

In our model we define Rekey transaction as a specific type of transaction different from close and pay. This is because some of the necessary conditions to be specified for a safe smart contract in this case are `crcv` and `rekey`, respectively equal and different from the *Null Address*. From the script below is possible to notice the presence of all the necessary conditions in the input SecTeal contract (lines 14-20). These conditions are correctly translated in Teal (lines and 39-47). The required parameter are reported in the SecTeal contract in lines 10 with the respective Teal translation in lines 36-38. In particular the necessary condition in line 7 is different from the others because the value assigned to the `rcv` (Receiver) parameter is not a constant value but another parameter. The Teal translation of this condition is reported in lines 30-35 Also in this case all the operators are listed at the end of the Teal contract.

```

1 #INPUT: SECTEAL SMART CONTRACT
2
3 (
4   (tx(0).type = int rekey)
5     and
6   (
7     (tx(0).rcv = tx(0).snd)
8       and
9     (
10      (tx(0).snd = addr
11        GTDE5ARA4MEC5PVDOP64JM505MQST63Q2K0Y2FLYFLXXD3PFSNJBYKITL)
12        and
13      (
14        (tx(0).val = int 0)
15          and
16        (
17          (tx(0).crcv = addr NULL ADD)
18            and
19          (tx(0).rekey = addr

```

```

20      LPHG7ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBK0Y2))))))
21    )
22 #OUTPUT: TEAL SMART CONTRACT
23
24
25 REKEY          <<<<
26
27
28
29 #pragma version 2
30 gtxn 0 Type
31 int pay
32 ==
33 gtxn 0 Receiver
34 gtxn 0 Sender
35 ==
36 gtxn 0 Sender
37 addr GTDE5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBK0Y2
38 ==
39 gtxn 0 Amount
40 int 0
41 ==
42 gtxn 0 CloseRemainderTo
43 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQ
44 ==
45 gtxn 0 RekeyTo
46 addr LPHG7ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBK0Y2
47 ==
48 &&
49 &&
50 &&
51 &&
52 &&

```

Now we show a series of tests of the Compiler in case the contract is not safe or it presents some errors and see how our tool works in different cases.

Let consider again CLOSE transaction smart contract.

8.0.5. Close without CloseRemainderTo

In the following contract it is not included any information regarding the CloseRemainderTo parameter.

NOTE: From the *GS-TABLE* we know that *crcv* (CloseRemainderTo) is a *Necessary* parameter so it is crucial for the safeness of the contract to indicate it in the right way.

From the Script it is evident that the compiler notice that the *crcv* parameter is missing (lines 24,36 and 49) and before doing something, it tries to interact with the user (lines 24-25, 37-38 and 50-51).

From this, we have two different outcomes depending on the response of the user.

For negative answers the program is interrupted displaying the error message (line 37-45):

[*THE CONTRACT IS NOT SAFE*]

For positive answers instead the tool shows the adjusted Teal contract with the proper placement of the CloseRemainderTo parameter (lines 49-85). The condition is added in lines 77-79 with the proper operator && (line 80).

```

1 #INPUT: SECTEAL SMART CONTRACT
2
3 (
4   (tx(0).type = int close)
5     and
6   (
7     (tx(0).fee < int 1000)
8       and
9     (
10      (tx(0).rcv = addr
11        ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBYPFZM)
12        and
13      (
14        (tx(0).snd = addr
15          LPHG7ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBYPKOY2)
16          and
17        (
18          (tx(0).val = int 0)
19            and
20          (tx(0).rekey = addr NULL ADD))))))
21
22 #OUTPUT: ERROR - IN CASE OF NEGATIVE CHOICE
23
24 ERROR :   crcv non present
25 Do you want to continue ? (Y/N) : N
26
27

```

```
28 CLOSE <<<
29
30
31
32 #pragma version 2
33 ['ERROR : crcv non present']
34
35
36 ERROR : crcv non present
37 Do you want to continue ? (Y/N) : Y
38 Insert necessary condition? (Y/N) : N
39
40 CLOSE <<<
41
42
43
44 #pragma version 2
45 ['Contract IS NOT SAFE : Condition on crcv non present']
46
47 #OUTPUT: TEAL SMART CONTRACT - IN CASE OF POSITIVE CHOICE
48
49 ERROR : crcv non present
50 Do you want to continue ? (Y/N) : Y
51 Insert necessary condition? (Y/N) : Y
52
53
54 CLOSE <<<
55
56
57
58 #pragma version 2
59 gtxn 0 Type
60 int pay
61 ==
62 gtxn 0 Fee
63 int 1000
64 <
65 gtxn 0 Receiver
66 addr ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBYPFZM
67 ==
68 gtxn 0 Sender
69 addr LPHG7ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBYPFZM
70 ==
71 gtxn 0 Amount
```

```

72 int 0
73 ==
74 gtxn 0 RekeyTo
75 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQ
76 ==
77 gtxn 0 CloseRemainderTo
78 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQ
79 !=
80 &&
81 &&
82 &&
83 &&
84 &&
85 &&

```

8.0.6. Close with parameter type error

As before, take a smart contract for close transaction with the wrong type associated to the parameter. More precisely on line 7 we indicate `addr` (address) as type for the parameter `fee`. However we know from the Table B.1 that the correct type is the integer one.

First, our tool notifies to the user that the type of `fee` is wrong and it asks if the user wants to modify the error (lines 28-29 and 67-68). In case the answer is negative the algorithm ends with the Teal contract without any correction (32-63); instead in the opposite case the error is automatically adjusted (line 80).

```

1 #INPUT: SECTEAL SMART CONTRACT
2
3 (
4 (tx(0).type = int close)
5   and
6   (
7     ( tx(0).fee < addr 1000)
8       and
9       (
10        (tx(0).rcv = addr
11          ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFNSNJJBYAFZM)
12        and
13        (
14          (tx(0).crcv = addr
15            ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFNSNJJBYAFZM)

```

```

16         and
17     (
18         (tx(0).snd = addr
19             LPHG7ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBKYOY2)
20         and
21     (
22         (tx(0).val = int 0)
23         and
24         (tx(0).rekey = addr NULL ADD))))))
25
26 #OUTPUT: TEAL SMART CONTRACT - NEGATIVE CHOICE
27
28 ERROR: type fee is non correct
29 Do you want modify it? N
30
31
32 CLOSE                <<<
33
34
35
36 #pragma version 2
37 gtxn 0 Type
38 int pay
39 ==
40 gtxn 0 Fee
41 addr 1000
42 <
43 gtxn 0 Receiver
44 addr ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBKYOY2
45 ==
46 gtxn 0 CloseRemainderTo
47 addr ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBKYOY2
48 ==
49 gtxn 0 Sender
50 addr LPHG7ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBKYOY2
51 ==
52 gtxn 0 Amount
53 int 0
54 ==
55 gtxn 0 RekeyTo
56 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQ
57 ==
58 &&
59 &&

```

```
60 &&
61 &&
62 &&
63 &&
64
65 #OUTPUT: TEAL SMART CONTRACT - POSITIVE CHOICE
66
67 ERROR: type fee is non correct
68 Do you want modify it? Y
69
70
71 CLOSE <<<
72
73
74
75 #pragma version 2
76 gtxn 0 Type
77 int pay
78 ==
79 gtxn 0 Fee
80 int 1000
81 <
82 gtxn 0 Receiver
83 addr ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2K0Y2FLYFLXXD3PFSNJJBYPFZM
84 ==
85 gtxn 0 CloseRemainderTo
86 addr ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2K0Y2FLYFLXXD3PFSNJJBYPFZM
87 ==
88 gtxn 0 Sender
89 addr LPHG7ARA4MEC5PVDOP64JM505MQST63Q2K0Y2FLYFLXXD3PFSNJJBYPK0Y2
90 ==
91 gtxn 0 Amount
92 int 0
93 ==
94 gtxn 0 RekeyTo
95 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQ
96 ==
97 &&
98 &&
99 &&
100 &&
101 &&
102 &&
```

8.0.7. Value type error

Line 11 presents the condition imposed to the Receiver of the close transaction. In this case the associated type is correct but there is an error in the address. Indeed it contains a lowercase letter so it does not respect the features of Address .

In this case, as usual, Secteal asks the user if he wants to continue. If the answer is positive then our tool compiles the TEAL smart contract without any correction (lines 86-121). In negative case, again the compiler asks if the user wants to adjust the error. Then, in case of positive or negative answer we have two different outcomes. Secteal displays an Error message (lines 30-40) or Secteal requires to insert the right value and compiles the Teal contract with the new one (45-82). In particular we can see the correct address in line 63.

```

1 #INPUT: SECTEAL SMART CONTRACT
2
3 (
4   (tx(0).type = int close)
5     and
6     (
7       (tx(0).fee < int 1000)
8         and
9         (
10          (tx(0).rcv = addr
11            ZZpF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJB YAFZM)
12              and
13              (
14                (tx(0).crcv = addr
15                  ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJB YAFZM)
16                    and
17                    (
18                      (tx(0).snd = addr
19                        LPHG7ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJB YKOY2)
20                          and
21                          (
22                            (tx(0).val = int 0)
23                              and
24                              (tx(0).rekey = addr NULL ADD)))))))))
25
26
27 #OUTPUT: TYPE ERROR
28
29

```

```
30 ERROR : rcv type error
31 Do you want to continue? (Y/N) : N
32 Do you want to change? (Y/N) : N
33
34
35 CLOSE <<<
36
37
38
39 #pragma version 2
40 ['ERROR : rcv type error']
41
42
43 #OUTPUT: TEAL SMART CONTRACT - ADJUSTED
44
45 ERROR : rcv type error
46 Do you want to continue? (Y/N) : N
47 Do you want to change? (Y/N) : Y
48 Insert : ZZPF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBYAFZM
49
50
51 CLOSE <<<
52
53
54
55 #pragma version 2
56 gtxn 0 Type
57 int pay
58 ==
59 gtxn 0 Fee
60 int 1000
61 <
62 gtxn 0 Receiver
63 addr ZZPF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBYAFZM
64 ==
65 gtxn 0 CloseReminderTo
66 addr ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBYAFZM
67 ==
68 gtxn 0 Sender
69 addr LPHG7ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBYKOY2
70 ==
71 gtxn 0 Amount
72 int 0
73 ==
```

```
74 gtxn 0 rekey-to
75 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQ
76 ==
77 &&
78 &&
79 &&
80 &&
81 &&
82 &&
83
84 #OUTPUT: TEAL SMART CONTRACT - NO ADJUSTMENT
85
86 ERROR : rcv type error
87 Do you want to continue? (Y/N) : Y
88
89
90 CLOSE <<<
91
92
93
94 #pragma version 2
95 gtxn 0 Type
96 int pay
97 ==
98 gtxn 0 Fee
99 int 1000
100 <
101 gtxn 0 Receiver
102 addr ZZpF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJB YAFZM
103 ==
104 gtxn 0 CloseRemainderTo
105 addr ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJB YAFZM
106 ==
107 gtxn 0 Sender
108 addr LPHG7ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJB YK0Y2
109 ==
110 gtxn 0 Amount
111 int 0
112 ==
113 gtxn 0 RekeyTo
114 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQ
115 ==
116 &&
117 &&
```

```

118 &&
119 &&
120 &&
121 &&

```

8.0.8. Necessary parameter value error

Let consider the case in which a smart contract for close transaction presents a *Value error*. The difference with the previous example is that, accordingly to the definition of *Value error* for *Necessary* parameter, the address associated to a specific parameter is correct but it does not fulfil the value condition associated to it in the corresponding *GS-Table*. In this case the *rcv* (CloseRemainderTo) parameter value is equal to the Null Address (lines 14). The output in line 34 is an Error message.

```

1 #INPUT: SECTEAL SMART CONTRACT
2
3 (
4 (tx(0).type = int close)
5     and
6 (
7 (tx(0).fee < int 1000)
8     and
9 (
10 (tx(0).rcv = addr
11     ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFPSNJJBYAFZM)
12     and
13 (
14 (tx(0).rcv = addr NULL ADD)
15     and
16 (
17 (tx(0).snd = addr
18     LPHG7ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFPSNJJBYKOY2)
19     and
20 (
21 (tx(0).val = int 0)
22     and
23 (tx(0).rekey = addr NULL ADD))))))
24
25
26
27
28

```

```

29 #OUTPUT: ERROR - CONTRACT NOT SAFE
30
31 CLOSE                <<<
32
33 #pragma version 2
34 ['ERROR : Contract is not safe: crcv is wrong']

```

8.0.9. Parameter not in *GS-Table*

Consider a contract with a parameter that is not included in *GS-Table* for that specific type. We consider a smart contract with type equal to close. Also the parameter `asstsnd` (asset sender) is reported in line 27. This parameter is not included in the corresponding *GS-Table*. In that case Secteal manages the problem with a Warning message asking to the user whether it wants to stop or continue (lines 36 and 48). If the answer of the user is positive SecTeal compiles the Teal contract including the `asstsnd` parameter (lines 51-85). Otherwise, we can see in line 42 that the tool returns an Error message .

```

1 #INPUT: SECTEAL SMART CONTRACT
2
3 (
4 (tx(0).type = int close)
5   and
6 (
7 (tx(0).fee < int 1000)
8   and
9 (
10 (tx(0).rcv = addr
11     ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJB YAFZM)
12   and
13 (
14 (tx(0).crcv = addr
15     ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJB YAFZM)
16   and
17 (
18 (tx(0).snd = addr
19     LPHG7ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJB YKOY2)
20   and
21 (
22 (tx(0).val = int 0)
23   and
24 (
25 (tx(0).rekey = addr NULL ADD)
26   and

```

```
27         (tx(0).asstsnd = addr NULL ADD)))))))))
28
29
30
31
32
33 #OUTPUT: WARNING - USER STOP THE PROGRAM
34
35 WARNING : asstsnd is not a field of this transaction type
36 Do you want to continue? (Y/N) : N
37
38
39 CLOSE          <<<
40
41 #pragma version 2
42 ['ERROR : asstsnd is not a field of this transaction type']
43
44
45 #OUTPUT: TEAL SMART CONTRACT - WITH THIS PARAMETER
46
47 WARNING : asstsnd is not a field of this transaction type
48 Do you want to continue? (Y/N) : Y
49
50
51 CLOSE          <<<
52
53
54 #pragma version 2
55 gtxn 0 Type
56 int pay
57 ==
58 gtxn 0 Fee
59 int 1000
60 <
61 gtxn 0 Receiver
62 addr ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2K0Y2FLYFLXXD3PFSNJJBAYAFZM
63 ==
64 gtxn 0 CloseRemainderTo
65 addr ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2K0Y2FLYFLXXD3PFSNJJBAYAFZM
66 ==
67 gtxn 0 Sender
68 addr LPHG7ARA4MEC5PVDOP64JM505MQST63Q2K0Y2FLYFLXXD3PFSNJJBAYK0Y2
69 ==
70 gtxn 0 Amount
```



```
10 (tx(0).rcv = addr
11     ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJB YAFZM)
12     and
13 (
14 (tx(0).val = int 100)
15     and
16 (
17 (tx(0).snd = addr
18     5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3JFSNJJC YAFZTOIJK)
19     and
20 (
21 (tx(0).crcv = addr NULL_ADD)
22     and
23 (tx(0).rekey = addr NULL_ADD))))))
24
25 #OUTPUT: LIST OF OPTIONS
26
27 WARNING : Possible transaction's type:
28 0 pay
29 1 close
30 2 rekey
31 3 None
32 Insert one option : 0
33
34 #OUTPUT: TEAL SMART CONTRACT - ADJUSTED FOR A PAY
35
36
37 PAY <<<
38
39
40
41 #pragma version 2
42 gtxn 0 FirstValid
43 int 100
44 ==
45 gtxn 0 Fee
46 int 1000
47 <
48 gtxn 0 Receiver
49 addr ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJB YAFZM
50 ==
51 gtxn 0 Amount
52 int 100
53 ==
```

```

54 gtxn 0 Sender
55 addr 5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3JFSNJJCYAFZT
56 ==
57 gtxn 0 CloseRemainderTo
58 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQ
59 ==
60 gtxn 0 RekeyTo
61 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQOIJK
62 ==
63 &&
64 &&
65 &&
66 &&
67 &&
68 &&

```

8.0.11. Contract with OR operator

As explained in chapter 7 the algorithm becomes more complex when the contracts include the OR operator. The tool parses the input structure and checks if each branch of the smart contract is Safe. In this particular example we have two OR operators that means that we should consider three branches (e.g. each branch is a Table in \mathcal{T}). In other words, there are three independent sets of conditions that should be satisfied at the same time in order the transaction to be accepted by the contract. For each branch our tool has the same performance of the examples above. First we notice in lines 43-48, how the tool reports to the user all the problems in each branch. In this example all the errors are sequentially fixed and the Teal smart contract is displayed in lines 51-114. Notice the correct position of the OR operator in lines 112-113. It is important to point out that in that case not all the parameters are listed at the end of the Teal smart contract. In lines 74-77, 93-96 and 109-111 there are three groups of operators that are associated to the three different branches of the contract.

```

1 #INPUT: SECTEAL SMART CONTRACT
2
3 (
4   (tx(0).type = int pay)
5     and
6   (
7     (
8       (tx(0).rcv = addr
9         ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBAYAFZM)

```

```

10         and
11     ((tx(0).val = int 100)
12         and
13     ((tx(0).snd = addr
14         5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3JFSNJCYAFZTSERT)
15         and
16     (tx(0).crcv = addr NULL ADD))))
17
18         or
19     (
20     (
21     (tx(0).rcv = addr
22         ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFNSNJBAYAFZM)
23         and
24     ((tx(0).val = int 100)
25         and
26     ((tx(0).snd = addr
27         5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3JFSNJCYAFZTSERT)
28         and
29     (tx(0).rekey = addr NULL ADD))))
30
31         or
32     (
33     (tx(0).rekey = addr NULL ADD)
34         and
35     ((tx(0).val = int 100)
36         and
37     ((tx(0).snd = addr
38         5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3JFSNJCYAFZTSERT)
39         and
40     (tx(0).crcv = addr NULL ADD))))))
41
42
43 ERROR : rekey non present
44 Insert condition? (Y/N) : Y
45 ERROR : crcv non present
46 Insert condition? (Y/N) : Y
47 WARNING : rcv non present
48 Do you want to continue? (Y/N) : Y
49
50
51 PAY <<<
52
53

```

```
54
55 #pragma version 2
56 gtxn 0 Type
57 int pay
58 ==
59 gtxn 0 Receiver
60 addr ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBYPFZM
61 ==
62 gtxn 0 Amount
63 int 100
64 ==
65 gtxn 0 Sender
66 addr LPHG7ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBYPKOY2
67 ==
68 gtxn 0 CloseRemainderTo
69 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQ
70 ==
71 gtxn 0 RekeyTo
72 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQ
73 ==
74 &&
75 &&
76 &&
77 &&
78 gtxn 0 Receiver
79 addr ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBYPFZM
80 ==
81 gtxn 0 Amount
82 int 100
83 ==
84 gtxn 0 Sender
85 addr LPHG7ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJBYPKOY2
86 ==
87 gtxn 0 rekey-to
88 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQ
89 ==
90 gtxn 0 CloseRemainderTo
91 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQ
92 ==
93 &&
94 &&
95 &&
96 &&
97 gtxn 0 RekeyTo
```

```

98 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQ
99 ==
100 gtxn 0 Amount
101 int 100
102 ==
103 gtxn 0 Sender
104 addr LPHG7ARA4MEC5PVDOP64JM5O5MQST63Q2KOY2FLYFLXXD3PFSNJJBK0Y2
105 ==
106 gtxn 0 CloseRemainderTo
107 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQ
108 ==
109 &&
110 &&
111 &&
112 ||
113 ||
114 &&

```

In the "A formal model of Algorand smart contracts" [16] more structured and complex smart contracts - listed in the Table 6.1 - are presented. All these contracts are completely detailed in [16] and we have reported the structure of the Oracle smart contract in chapter 6.

We show three examples of these family of smart contracts written in Secteal: Oracle, HTLC and Periodic payment [17]. Then, as we have done before with the other examples, we test our tool using the first one.

8.0.12. Hash Time Lock Contract (HTLC)

A user **a** promises that it will either reveal a secret s_a by round $r_{max} = 3000$, or pay a penalty to **b**. More sophisticated contracts, e.g. gambling games, use this mechanism to let players generate random numbers in a fair way. We define the HTLC as the following contract, parameterised on the two users **a**, **b** and the hash $h_a = H(s_a)$ of the secret. Where :

a = ZZAF5ARA4MEC5PVDOP64JM5O5MQST63Q2KOY2FLYFLXXD3PFSNJJBK0Y2

b = SOEI4UA72A7ZL5P25GNISSVWW724YABSGZ7GHW5ERV4QKK2XSXLXGXP5Y

H_a = bXkgc3RyaW5n

The contract accepts only close transactions with receiver **a** or **b**. User **a** can collect the funds in the contract only by providing the secret s_a in $\text{arg}(0)$, effectively making s_a public. Instead, if **a** does not reveal s_a , then **b** can collect the funds after round r_{max} .

The script below reports the SecTeal representation of the HTLC. The specified Type parameter is equal to close (line 2) and it includes an OR operator (line 10) but, all the necessary conditions for a contract of this type are not indicated. For this reason the contract is correct but it is not safe.

```

1 (
2   (tx(0).type = int close)
3     and
4   (
5     (
6       (tx(0).rcv = addr
7         ZZAF5ARA4MEC5PVDOP64JM5O5MQST63Q2KOY2FLYFLXXD3PFSNJJBAYAFZM)
8         and
9       (H(arg(0)) = byte base64 bXkgc3RyaW5n))
10      or
11     (
12       (tx(0).rcv = addr
13         SOEI4UA72A7ZL5P25GNISSVWW724YABSGZ7GHW5ERV4QKK2XSXLXGXP5Y)
14       and
15       (tx(0).fv > int 3000))))

```

8.0.13. Periodical payment

We want to ensure that:

a = ZZAF5ARA4MEC5PVDOP64JM5O5MQST63Q2KOY2FLYFLXXD3PFSNJJBAYAFZM can withdraw a fixed amount of $v = 10000$ Algos at fixed time windows of $p = 2000$ rounds. We can implement this behaviour through the following contract, which can be refilled when needed.

The contract accepts only pay transactions of v Algos to receiver **a**. The conditions $tx.fv \% p = 0$ and $tx.lv = tx.fv + d$ ensure that the contract only accepts transactions with validity interval $[k * p, k * p + d]$, for $k \in \mathbb{N}$. The condition $tx.lx = n$ ensures that at most one such transactions is accepted for each time window.

The contract accepts only pay transactions of v Algos to receiver **a**. The conditions $tx.fv \% p = 0$ and $tx.lv = tx.fv + d$ ensure that the contract only accepts transactions with validity interval $[k * p, k * p + d]$, for $k \in \mathbb{N}$. The condition $tx.lx = n$ ensures that at most one such transactions is accepted for each time window.

The script below reports the SecTeal representation of the Periodical payment. The specified Type parameter is equal to pay (line 2) but all the necessary conditions for a

contract of this type are not indicated. For this reason the contract is correct but it is not safe.

```

1 (
2   (tx(0).type = int pay)
3     and
4   (
5     (
6       (tx(0).val = int 10000)
7         and
8       (
9         (tx(0).rcv = addr
10          ZZAF5ARA4MEC5PVDOP64JM5O5MQST63Q2KOY2FLYFLXXD3PFSNJJB YAFZM)
11          and
12        (
13          (
14            (tx(0).fv \% int 2000) = int 0)
15              and
16            (
17              (tx(0).lv < (tx(0).fv + int 1000))
18                and
19              (tx(0).lx = int 4000))))))

```

8.0.14. Oracle

We want to ensure that either

a = ZZAF5ARA4MEC5PVDOP64JM5O5MQST63Q2KOY2FLYFLXXD3PFSNJJB YAFZM
or **b** = SOEI4UA72A7ZL5P25GNISSVWW724YABSGZ7GHW5ERV4QKK2XSXLXGXP G5Y
can withdraw all the Algos in the contract, depending on the outcome of a certain boolean event. Such an event is certified by an oracle

o = NKOS3PFXD73DDFSEE6SSQQ6SEZGOQBTE6KQAGEDQZRGOCXSNH3XGCXWGHU
by signing either the value 1 or 0. The contract only accepts a close tx. $\arg(0)$ is the outcome of the event, $\arg(1)$ is **o**'s signature on $(\text{Oracle}, \arg(0))$. After round $r_{max} = 3000$, **a** can collect the money. The following expression encodes all constraints.

From the script below we show how SecTeal works for the Oracle smart contract. It is a more complex example of what we have detailed in subsection 8.0.11. The type is equal to close (line 4) and from line 34 to 57 are reported sequentially all the errors and the interactions between the tool and the user for each branch as in subsection 8.0.11. After that the Teal smart contract is compiled (lines 60-124).

```

1 #INPUT: SECTEAL ORACLE SMART CONTRACT
2
3 (
4 (tx(0).type = int close)
5     and
6 (
7     (
8     (tx(0).fv > int 3000)
9         and
10    (tx(0).crcv = addr
11        ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJB YAFZM))
12        or
13    (
14    (
15    (arg(0) = byte base64 bXkgc3RyaW5n)
16        and
17    (versig(arg(0),arg(1),addr
18        NKOS3PFXD73DDFSEE6SSQQ6SEZGOQBTE6KQAGEDQZRGOCXSNH3XGCXWGHU)
19        and
20    (tx(0).crcv = addr
21        ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFSNJJB YAFZM))
22    )
23        or
24    (
25    (arg(0) = byte base64 bXkgc3RyaW5n)
26        and
27    (versig(arg(0),arg(1),addr
28        NKOS3PFXD73DDFSEE6SSQQ6SEZGOQBTE6KQAGEDQZRGOCXSNH3XGCXWGHU)
29        and
30    (tx(0).crcv = addr
31        SOEI4UA72A7ZL5P25GNISSVWW724YABSGZ7GHW5ERV4QKK2XSXLXGXP5Y))
32    ))))
33
34 #OUTPUT: TEAL ORACLE SMART CONTRACT - ADJUSTED
35
36 WARNING : snd non present
37 Do you want to continue? (Y/N) : Y
38 WARNING : rcv non present
39 Do you want to continue? (Y/N) : Y
40 ERROR : val non present
41 Insert condition? (Y/N) : Y
42 ERROR : rekey non present
43 Insert condition? (Y/N) : Y
44 WARNING : snd non present

```

```
43 Do you want to continue? (Y/N) : Y
44 WARNING : rcv non present
45 Do you want to continue? (Y/N) : Y
46 ERROR : val non present
47 Insert condition? (Y/N) : Y
48 ERROR : rekey non present
49 Insert condition? (Y/N) : Y
50 WARNING : snd non present
51 Do you want to continue? (Y/N) : Y
52 WARNING : rcv non present
53 Do you want to continue? (Y/N) : Y
54 ERROR : val non present
55 Insert condition? (Y/N) : Y
56 ERROR : rekey non present
57 Insert condition? (Y/N) : Y
58
59
60 CLOSE <<<
61
62
63 #pragma version 2
64 gtxn 0 Type
65 int pay
66 ==
67 gtxn 0 FirstValid
68 int 3000
69 >
70 gtxn 0 CloseRemainderTo
71 addr ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2K0Y2FLYFLXXD3PFSNJJBAYAFZM
72 ==
73 gtxn 0 Amount
74 int 0
75 ==
76 &&
77 gtxn 0 RekeyTo
78 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQ
79 ==
80 &&
81 &&
82 arg 0
83 byte base64 0
84 ==
85 arg 0
86 arg 1
```

```
87 addr NKOS3PFXD73DDFSEE6SSQQ6SEZGOQBTE6KQAGEDQZRGOCXSNH3XGCXWGHU
88 ed25519verify
89 gtxn 0 CloseReminderTo
90 addr ZZAF5ARA4MEC5PVDOP64JM505MQST63Q2KOY2FLYFLXXD3PFNSJJBYAFZM
91 ==
92 gtxn 0 Amount
93 int 0
94 ==
95 &&
96 gtxn 0 RekeyTo
97 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQ
98 ==
99 &&
100 &&
101 &&
102 arg 0
103 byte base64 1
104 ==
105 arg 0
106 arg 1
107 addr NKOS3PFXD73DDFSEE6SSQQ6SEZGOQBTE6KQAGEDQZRGOCXSNH3XGCXWGHU
108 ed25519verify
109 gtxn 0 CloseReminderTo
110 addr SOEI4UA72A7ZL5P25GNISSVW724YABSGZ7GHW5ERV4QKK2XSXLXGXP5Y
111 ==
112 gtxn 0 Amount
113 int 0
114 ==
115 &&
116 gtxn 0 RekeyTo
117 addr AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAY5HFKQ
118 ==
119 &&
120 &&
121 &&
122 ||
123 ||
124 &&
```

9 | Conclusions and future developments

In this work we have presented the idea and the realisation of an extended compiler suitable for smart contract on Algorand. Our thesis bases its implementation on the formal model for Algorand smart contracts [16]. At the beginning some notation is introduced and the model of [16] is explained in details. As already pointed out, our contribution is the improvement of the tool Secteal [17]. In particular we build various extensions to the compiler. First of all we have updated the syntax and the architecture of the declarative language with respect to the new version of ASC1 always considering stateless smart contract. We also slightly modify the architecture of it creating two new transactions try to get Secteal more suitable for the user. For each type, we have associated a Gold Standard-Table describing precisely the required parameters which belong to that transaction type, specifying which are necessary to be express into the contract to satisfy the condition of safeness. The core of our work is the implementation of the Secure function algorithm. As final result, Secteal is able to compile stateless smart contract for all the types listed in [16] compiling it in a Safe Teal smart contract. It checks the contract written by the user, adjust it in case it is not Safe or if there are some errors. In this way the output of our teal is a correct and safe stateless smart contract ready to be deployed on Algorand testnet. In the last part of our work we detail 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 model of Algorand smart contracts presented in this work can be expanded depending on the evolution of Algorand itself and its original model. It could be also very interesting investigate declarative languages for stateful Algorand smart contracts. Before doing that also the model should be modified in order 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.

Bibliography

- [1] Python, 2021. URL <https://www.python.it/>.
- [2] Vsc, 2021. URL <https://code.visualstudio.com/>.
- [3] Algorand. Algorand testnet, 1999. URL <http:https://developer.algorand.org/docs/get-details/algorand-networks/testnet/>.
- [4] Algorand. Introduction, 2021. URL https://developer.algorand.org/docs/get-details/dapps/smart-contracts/?from_query=ASC1#create-publication-overlay.
- [5] Algorand. Introduction smart contract, 2021. URL <https://developer.algorand.org/docs/get-details/dapps/smart-contracts/>.
- [6] Algorand. Understanding-algorand-the-blockchain-which-claims-to-solve-the-trilemma, 2021. URL <https://community.algorand.org/blog/understanding-algorand-the-blockchain-which-claims-to-solve-the-trilemma/>.
- [7] Algorand. Avm, 2021. URL <https://developer.algorand.org/docs/get-details/dapps/avm/>.
- [8] Algorand. Opcodes, 2021. URL <https://developer.algorand.org/docs/get-details/dapps/avm/teal/opcodes/>.
- [9] Algorand. How algorand is building a scalable blockchain ecosystem, 2021. URL <https://www.algorand.com/resources/blog/algorand-building-scalable-sustainable-blockchain-ecosystem>.
- [10] Algorand. Smart contract, 2021. URL <https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/>.
- [11] Algorand. The algorand virtual machine (avm) and teal, 2021. URL <https://developer.algorand.org/docs/get-details/dapps/avm/teal/specification>.
- [12] Algorand. The smart contract language, 2021. URL <https://developer.algorand.org/docs/get-details/dapps/avm/teal/>.

- [13] M. Andrychowicz, S. Dziembowski, D. Malinowski, and Ł. Mazurek. Fair two-party computations via bitcoin deposits. In *International Conference on Financial Cryptography and Data Security*, pages 105–121. Springer, 2014.
- [14] M. Andrychowicz, S. Dziembowski, D. Malinowski, and Ł. Mazurek. Secure multi-party computations on bitcoin. *Communications of the ACM*, 59(4):76–84, 2016.
- [15] W. Banasik, S. Dziembowski, and D. Malinowski. Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In *European symposium on research in computer security*, pages 261–280. Springer, 2016.
- [16] M. Bartoletti, A. Bracciali, C. Lepore, A. Scalas, and R. Zunino. A formal model of algorand smart contracts. In *International Conference on Financial Cryptography and Data Security*, pages 93–114. Springer, 2021.
- [17] M. Bartoletti, A. Bracciali, C. Lepore, A. Scalas, and R. Zunino. Secteal, 2021. URL <http://secteal.cs.stir.ac.uk>.
- [18] I. Bentov and R. Kumaresan. How to use bitcoin to design fair protocols. In *Annual Cryptology Conference*, pages 421–439. Springer, 2014.
- [19] 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.
- [20] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*, pages 91–96, 2016.
- [21] M. Brenner, T. Moore, and M. Smith. *Financial cryptography and data security*. Springer, 2014.
- [22] M. M. Chakravarty, J. Chapman, K. MacKenzie, O. Melkonian, M. Peyton Jones, and P. Wadler. The extended utxo model. In *International Conference on Financial Cryptography and Data Security*, pages 525–539. Springer, 2020.
- [23] S. Chandra, S. Paira, S. S. Alam, and G. Sanyal. A comparative survey of symmetric and asymmetric key cryptography. In *2014 international conference on electronics, communication and computational engineering (ICECCE)*, pages 83–93. IEEE, 2014.
- [24] J. Chen and S. Micali. Algorand: A secure and efficient distributed ledger. *Theoretical Computer Science*, 777:155–183, 2019.

- [25] L. Chen, L. Xu, N. Shah, Z. Gao, Y. Lu, and W. Shi. On security analysis of proof-of-elapsed-time (poet). In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 282–297. Springer, 2017.
- [26] S. Delgado-Segura, C. Pérez-Solà, G. Navarro-Arribas, and J. Herrera-Joancomartí. A fair protocol for data trading based on bitcoin transactions. *Future Generation Computer Systems*, 107:832–840, 2020.
- [27] R. Fagin. Horn clauses and database dependencies. *Journal of the ACM (JACM)*, 29(4):952–985, 1982.
- [28] P. Gaži, A. Kiayias, and D. Zindros. Proof-of-stake sidechains. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 139–156. IEEE, 2019.
- [29] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*, pages 51–68, 2017.
- [30] S. Goldberg, J. Vcelak, D. Papadopoulos, and L. Reyzin. Verifiable random functions (vrf). 2018.
- [31] I. Grishchenko, M. Maffei, and C. Schneidewind. Ethertrust: Sound static analysis of ethereum bytecode. *Technische Universität Wien, Tech. Rep*, pages 1–41, 2018.
- [32] I. Grishchenko, M. Maffei, and C. Schneidewind. Foundations and tools for the static analysis of ethereum smart contracts. In *International Conference on Computer Aided Verification*, pages 51–78. Springer, 2018.
- [33] I. Grishchenko, M. Maffei, and C. 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.
- [34] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, et al. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217. IEEE, 2018.
- [35] Y. Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*, pages 520–535. Springer, 2017.
- [36] K. Karantias, A. Kiayias, and D. Zindros. Proof-of-burn. In *International conference on financial cryptography and data security*, pages 523–540. Springer, 2020.

- [37] A. Kiayias and D. Zindros. Proof-of-work sidechains. In *International Conference on Financial Cryptography and Data Security*, pages 21–34. Springer, 2019.
- [38] R. Klomp and A. Bracciali. On symbolic verification of bitcoin’s script language. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 38–56. Springer, 2018.
- [39] R. Kumaresan and I. Bentov. How to use bitcoin to incentivize correct computations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 30–41, 2014.
- [40] R. Kumaresan, T. Moran, and I. Bentov. How to use bitcoin to play decentralized poker. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 195–206, 2015.
- [41] C. Lepore, M. Ceria, A. Visconti, U. P. Rao, K. A. Shah, and L. Zanolini. A survey on blockchain consensus with a performance comparison of pow, pos and pure pos. *Mathematics*, 8(10):1782, 2020.
- [42] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269, 2016.
- [43] T. J. Parr and R. W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.
- [44] N. Popper. Hal finney, cryptographer and bitcoin pioneer, dies at 58. *NYTimes*. *Archived from the original on*, 3, 2014.
- [45] N. Popper. Decoding the enigma of satoshi nakamoto and the birth of bitcoin. *New York Times*, 15, 2015.
- [46] K. Rupić, L. Rožić, and A. Derek. Mechanized formal model of bitcoin’s blockchain validation procedures. In *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- [47] M. Salimitari and M. Chatterjee. An overview of blockchain and consensus protocols for iot networks. *arXiv preprint arXiv:1809.05613*, pages 1–12, 2018.
- [48] C. Schwarz-Schilling, J. Neu, B. Monnot, A. Asgaonkar, E. N. Tas, and D. Tse. Three attacks on proof-of-stake ethereum. *arXiv preprint arXiv:2110.10086*, 2021.
- [49] A. Yakovenko. Solana: A new architecture for a high performance blockchain v0.8.13. *Whitepaper*, 2018.

- [50] K. Zhidanov, S. Bezzateev, A. Afanasyeva, M. Sayfullin, S. Vanurin, Y. Bardinova, and A. Ometov. Blockchain technology for smartphones and constrained iot devices: A future perspective and implementation. In *2019 IEEE 21st Conference on Business Informatics (CBI)*, volume 2, pages 20–27. IEEE, 2019.

A | Appendix A

a, b, \dots	Users (key pairs)	$T_w \subseteq T$	Transactions in last Δ_{max} rounds
$x, y, \dots \in X$	Addresses	$f_{asst} \in A \rightarrow X$	Asset manager
$\tau, \tau', \dots \in A$	Assets	$f_{lx} \in (X \times N) \rightarrow N$	Lease map
$v, w, \dots \in 0..2^{64} - 1$	Values	$f_{frz} \in X \rightarrow 2^A$	Freeze map
$\sigma, \sigma' \in A \rightarrow N$	Balances	Γ, Γ', \dots	Blockchain states
$x[\sigma]$	Accounts	$\models \sigma$	Valid balance
$t, t', \dots \in T$	Transactions	$f_{lx}, r \models t$	Valid time constraint
e, e', \dots	Scripts	$\mathcal{W} \models \mathcal{T}, i$	Authorized transaction in group
$r, r', \dots \in N$	Rounds	$\llbracket e \rrbracket_{\mathcal{T}, i}^{\mathcal{W}}$	Script evaluation

Table A.1: Summary of notation [16]

pay	snd,rcv, val, asst	snd transfers val units of asst to rcv (possibly creating rcv)
close	snd,rcv, asst	snd gives asst to rcv and removes it (if Algo, closes snd)
gen	snd,rcv, val	snd mints val units of a new asset, managed by rcv
optin	snd, asst	snd opts in to receive units of asset asst
burn	asst	asst is removed from the creator (if sole owner)
rvk	snd,rcv, val, asst	asst's manager transfers val units of asst from snd to rcv
frz	snd, asst	asst's manager freezes snd's use of asset asst
unfrz	snd, asst	asst's manager unfreezes snd's use of asset asst
delegate	snd, asst, rcv	asst's manager delegates asst to new manager rcv

Table A.2: Transaction types. Fields type, fv, lv, lx are common to all types. [16]

$e ::= v$	constant
$e \circ e$	arithmetic ($\circ \in \{+, -, <, \leq, =, \geq, >, *, /, \%, \text{and}, \text{or}\}$)
note	negation
txlen	number of transactions in the atomic group
txpos	index of current transaction in the atomic group
$\text{txid}(n)$	identifier of n -th transaction in the atomic group
$\text{tx}(n).f$	value of field f of n^{th} transaction in the atomic group
$\text{arg}(n)n^{\text{th}}$	argument of the current transaction
$H(e)$	hash
$\text{versig}(e, e, e)$	signature verification

Syntactic sugar: $\text{false} ::= 1 = 0$ $\text{true} ::= 1 = 1$ $\text{tx}.f ::= \text{tx}(\text{txpos}).f$
 $\text{txid} ::= \text{txid}(\text{txpos})$ *if* e_0 *then* e_1 *else* $e_2 ::= (e_0 \text{ and } e_1) \text{ or } ((\text{not } e_0) \text{ and } e_2)$

Table A.3: Smart contract scripts (inspired by PyTeal) [16]

$$\begin{aligned}
\llbracket e \rrbracket_{\mathcal{T},i}^{\mathcal{W}} = v \quad & \llbracket e \circ e' \rrbracket_{\mathcal{T},i}^{\mathcal{W}} = \llbracket e \rrbracket_{\mathcal{T},i}^{\mathcal{W}} \circ_{\perp} \llbracket e' \rrbracket_{\mathcal{T},i}^{\mathcal{W}} \quad & \llbracket \text{not } e \rrbracket_{\mathcal{T},i}^{\mathcal{W}} = \neg_{\perp} \llbracket e \rrbracket_{\mathcal{T},i}^{\mathcal{W}} \\
\llbracket \text{tx}(n).f \rrbracket_{\mathcal{T},i}^{\mathcal{W}} = (\mathcal{T}.n).f \quad & (0 \leq n < |\mathcal{T}|) \quad & \llbracket \text{txid}(n) \rrbracket_{\mathcal{T},i}^{\mathcal{W}} = \mathcal{T}.n \quad (0 \leq n < |\mathcal{T}|) \\
\llbracket H(e) \rrbracket_{\mathcal{T},i}^{\mathcal{W}} = H(\llbracket e \rrbracket_{\mathcal{T},i}^{\mathcal{W}}) \quad & \llbracket \text{versig}(e_1, e_2, e_3) \rrbracket_{\mathcal{T},i}^{\mathcal{W}} = \text{ver}_k(m, s) \quad & \left(\begin{array}{l} m = (\mathcal{T}.i.\text{snd}, \llbracket e_1 \rrbracket_{\mathcal{T},i}^{\mathcal{W}}) \\ s = \llbracket e_2 \rrbracket_{\mathcal{T},i}^{\mathcal{W}} \quad k = \llbracket e_3 \rrbracket_{\mathcal{T},i}^{\mathcal{W}} \end{array} \right)
\end{aligned}$$

Table A.4: Evaluation of scripts in Table A.3. [16]

Secteal	Teal
and	&&
or	
=	==
+	+
-	-
*	*
/	/
%	%
<	<
>	>
≥	≥
≤	≤

Table A.5: Operator defined in the SecTeal language grammar

exp:= val	a value
(exp ◦ exp)	◦ ∈ <i>operation</i>
not(exp)	negation
txlen	number of tx in the atomic group
txid(INT)	identifier of n-th tx in the atomic group
tx(INT).FIELD	value of the current tx
arglen	number of arguments in the current tx
H(exp)	Hash
versig(exp1,exp2,exp3)	signature verification

Table A.6: SecTeal grammar defines the expression accepted by the compiler

val:= byte base64 BYTE
int BYTE
addr BYTE
BYTE:= [a-z0-9] any sequence of letters and digits
INT:= [0-9] any sequence of digits

Table A.7: SecTeal grammar for val

snd
rcv
crecv
val
fv
lv
lx
type
asst
asstval
asstsnd
asstrev
casstrev
fee
note
rekey
confasst
confasstman
confasstfrz
confasstres
confasstclbk
frzasst
frzacnt
asstfrz

Table A.8: SecTeal: FIELD, they are all the possible fields for a transaction

Secteal	Teal
H	sha256
arg	arg
int	int
byte	byte
snd	Sender
rcv	Receiver
crcv	CloseReminderTo
val	Amount
fv	FirstValid
lv	LastValid
lx	Lease
type	Type
asst	Asset
asstval	AssetAmount
asstsnd	AssetSender
asstrcv	AssetReceiver
casstrcv	AssetCloseTo
fee	Fee
note	Note
rekey	RekeyTo
len	len
confasst	ConfigAsset
confasstman	ConfigAssetManager
confasstfrz	ConfigAssetFreeze
confasstres	ConfigAssetReserve
confasstclbk	ConfigAssetClawback
frzasst	FreezeAsset
frzacnt	FreezeAccount
asstfrz	AssetFrozen
tx	gtxn
txlen	Groupsize
txid	TxID
verisig	ed25519verify
byte base64	byte base64
Round	Round

Table A.9: All the parameter: SecTeal vs Teal grammar

B | Appendix B

B.1. Type Parameter

Parameter	Type
type	int
snd	addr
rcv	addr
val	int
crcv	addr
fv	int
lv	int
lx	int
note	byte
fee	int
asst	int
asstrev	addr
asstval	int
casstrev	addr
asstsnd	addr
confasst	int
confasstman	addr
confasstfrz	addr
confasstres	addr
confasstclbk	addr
frzasst	int
frzacnt	addr
asstfrz	int
rekey	addr

Table B.1: Type parameters, in this table is associated each parameter with the corresponding Teal translation

B.2. *GS*-Table

PAY:

Param	Category	Type	Value
TYPE	T	String	
SND	T	Address	
RCV	T	Address	
VAL	T	Integer	
CRCV	**	Address	Null Address
REKEY	**	Address	Null Address
FV	⊥	Integer	
LV	⊥	Integer	
LX	⊥	Integer	
NOTE	⊥	String	
FEE	⊥	Integer	

Table B.2: PAY - GS-Table

PAY ASSET:

Param	Category	Type	Value
TYPE	T	String	
ASSTRCV	T	Address	
ASST	T	Integer	
SND	**	Address	!= Null Address
ASSTSND	**	Address	Null Address
ASSTVAL	**	Integer	!= Null
CASSTRCV	**	Address	Null Address
FV	⊥	Integer	
LV	⊥	Integer	
LX	⊥	Integer	
NOTE	⊥	String	
FEE	⊥	Integer	

Table B.3: PAY A - GS-Table

CLOSE:

Param	Category	Type	Value
TYPE	⊥	String	
SND	⊥	Address	
RCV	⊥	Address	
VAL	**	Integer	Null
REKEY	**	Address	Null Address
CRCV	**	Address	!= Null Address
FV	⊥	Integer	
LV	⊥	Integer	
LX	⊥	Integer	
NOTE	⊥	String	
FEE	⊥	Integer	

Table B.4: CLOSE - *GS-Table*

CLOSE ASSET:

Param	Category	Type	Value
TYPE	⊥	String	
ASS	⊥	Integer	
CASSTRCV	**	Address	!= Null Address
ASSTSND	**	Address	!= Null Address
ASSTVAL	**	Integer	Null
FV	⊥	Integer	
LV	⊥	Integer	
LX	⊥	Integer	
NOTE	⊥	String	
FEE	⊥	Integer	

Table B.5: CLOSE A - *GS-Table*

GEN:

Param	Category	Type	Value
TYPE	T	String	
SND	T	Address	
ASST	T	Integer	
ASSTVAL	T	Integer	
CONFASSTMAN	**	Address	!= Null Address
CONFASST	**	Integer	Null
CONFASSTFRZ	**	Address	CONFASSTMAN
CONFASSTRES	**	Address	CONFASSTMAN
CONFASSTCLBK	**	Address	CONFASSTMAN
FV	⊥	Integer	
LV	⊥	Integer	
LX	⊥	Integer	
NOTE	⊥	String	
FEE	⊥	Integer	

Table B.6: GEN - *GS-Table*

OPTIN:

Param	Category	Type	Value
TYPE	T	String	
ASST	T	Integer	
SND	**	Address	!= Null Address
ASSTSND	**	Address	Null Address
ASSTRCV	**	Address	SND
ASSTVAL	**	Address	Null
FV	⊥	Integer	
LV	⊥	Integer	
LX	⊥	Integer	
NOTE	⊥	String	
FEE	⊥	Integer	

Table B.7: OPTIN - *GS-Table*

 REVOKE:

Param	Category	Type	Value
TYPE	\top	String	
ASST	\top	Integer	
SND	**	Address	!= Null Address
ASSTSND	**	Address	!= Null Address
ASSTRCV	\top	Address	SND
ASSTVAL	\top	Integer	
FV	\perp	Integer	
LV	\perp	Integer	
LX	\perp	Integer	
NOTE	\perp	String	
FEE	\perp	Integer	

Table B.8: RVK - *GS-Table*

 FREEZE:

Param	Category	Type	Value
TYPE	\top	String	
FRZASST	\top	Integer	
FRZACNT	\top	Address	
ASSTFRZ	**	bool	True
FV	\perp	Integer	
LV	\perp	Integer	
LX	\perp	Integer	
NOTE	\perp	String	
FEE	\perp	Integer	

Table B.9: FRZ - *GS-Table*

UNFREEZE:

Param	Category	Type	Value
TYPE	\top	String	
FRZASST	\top	Integer	
FRZACNT	\top	Address	
ASSTFRZ	**	bool	False
FV	\perp	Integer	
LV	\perp	Integer	
LX	\perp	Integer	
NOTE	\perp	String	
FEE	\perp	Integer	

Table B.10: UNFRZ - *GS-Table*

BURN:

Param	Category	Type	Value
TYPE	\top	String	
CONFASST	\top	Integer	
CONFASSTMAN	**	Integer	Length == 0
CONFASSTFRZ	**	Integer	Length == 0
CONFASSTRES	**	Integer	Length == 0
CONFASSTCLBK	**	Integer	Length == 0
FV	\perp	Integer	
LV	\perp	Integer	
LX	\perp	Integer	
NOTE	\perp	String	
FEE	\perp	Integer	

Table B.11: BURN - *GS-Table*

DELEGATE:

Param	Category	Type	Value
TYPE	$\bar{\top}$	String	
SND	\top	Address	
CONFASSTMAN	,	Address	!= Null Address
CONFASSTMAN	**	Address	Length != 0
CONFASST	**	Integer	!= Null
CONFASSTFRZ	**	Address	CONFASSTMAN
CONFASSTRES	**	Address	CONFASSTMAN
CONFASSTCLBK	**	Address	CONFASSTMAN
FV	\perp	Integer	
LV	\perp	Integer	
LX	\perp	Integer	
NOTE	\perp	String	
FEE	\perp	Integer	

Table B.12: DELEGATE - *GS-Table*

REKEY:

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

Table B.13: REKEY - *GS-Table*

List of Figures

5.1	TEAL Architecture Overview, from Algorand Developer Portal [12]	28
5.2	Getting Transaction Properties, from Algorand Developer Portal	29
5.3	Pseudo Opcodes, from Algorand Developer Portal	29
5.4	Operators, from Algorand Developer Portal	30
7.1	SecTeal Architecture	50
7.2	Extended SecTeal Architecture	51
7.3	Table representation of OR and AND Tree of example	60
7.4	Table representation of P parameters	62
7.5	Table representation of P parameters	63

List of Tables

3.1	PoS	11
3.2	PoW	11
6.1	selection of smart contracts and the design patterns implemented in [16] . .	45
7.1	Left table τ : Algo, Right table τ : Asset	50
7.2	PAY <i>GS-TABLE</i>	55
7.3	PAY A <i>GS-TABLE</i>	56
A.1	Summary of notation [16]	107
A.2	Transaction types. Fields type, fv, lv, lx are common to all types. [16] . . .	107
A.3	Smart contract scripts (inspired by PyTeal) [16]	108
A.4	Evaluation of scripts in Table A.3. [16]	108
A.5	Operator defined in the SecTeal language grammar	108
A.6	SecTeal grammar defines the expression accepted by the compiler	109
A.7	SecTeal grammar for val	109
A.8	SecTeal: FIELD, they are all the possible fields for a transaction	109
A.9	All the parameter: SecTeal vs Teal grammar	110
B.1	Type parameters, in this table is associated each parameter with the corresponding Teal translation	111
B.2	PAY - <i>GS-Table</i>	112
B.3	PAY A - <i>GS-Table</i>	112
B.4	CLOSE - <i>GS-Table</i>	113
B.5	CLOSE A - <i>GS-Table</i>	113
B.6	GEN - <i>GS-Table</i>	114
B.7	OPTIN - <i>GS-Table</i>	114
B.8	RVK - <i>GS-Table</i>	115
B.9	FRZ - <i>GS-Table</i>	115
B.10	UNFRZ - <i>GS-Table</i>	116
B.11	BURN - <i>GS-Table</i>	116

B.12 DELEGATE - <i>GS-Table</i>	117
B.13 REKEY - <i>GS-Table</i>	117