DiaLekTos: Privacy-preserving Smart Contracts

Tadas Vaitiekūnas
vtadas25@gmail.com

Abstract

Digital ledger technologies supporting smart contracts usually does not ensure any privacy for user transactions or state. Most solutions to this problem either use private network setups, centralized parties, hardware enclaves, or cryptographic primitives, which are novel, complex, and computationally expensive. This paper looks into an alternative way of implementing smart contracts. Our construction of a protocol for smart contracts employs an overlay protocol design pattern for decentralized applications, which separates transaction ordering from transaction validation. This enables consensus on application state while revealing only encrypted versions of transactions to public consensus protocol network. UTXO-based smart contract model allows partitioning state of distributed ledger in a way that participants would need to decrypt and reach consensus only on those transactions, which are relevant to them. We present security analysis, which shows that, assuming presence of a secure consensus protocol, our construction achieves consensus on UTXO-based transactions, while hiding most of transaction details from all protocol parties, except a limited subset of parties, which need particular transactions for construction of their state.

1 Introduction

Distributed ledger technologies (DLTs) enable parties to reach consensus on some state without depending on central authority for validation and ordering of events. This enables applications, in which only a minimal amount of trust is needed between parties of a system. This is especially desirable feature for governance and financial systems. Naturally, there has been a proliferation of these kinds of systems being created using distributed ledger technologies as a foundation [21].

The main feature of DLTs (ensuring that all parties of the protocol have the same view of the ledger) is also its one of the main weakness. The ledger and all the transactions on it are typically visible to every participant in the network [8]. This means a lack of privacy. Alternative to DLTs are of course more traditional centralized systems. They offer much better privacy guarantees, simply because all the data is managed by one entity and while that implies that user data is available to that entity, the data is at least not available to every other user. This kind of privacy is a standard for most financial applications in today's world. Financial applications also happen to be one the main areas of use for DLTs. So, it is no wonder, that lack of privacy is often named as the biggest challenge for the adoption of these technologies [16].

Thus, privacy in DLT-based applications is an active area of interest and there are already projects which focus on it in various ways. Projects like Monero [3] and ZCash [20] achieve privacy of user account balances and transfers but are limited in their solution to the use case of cryptocurrency. With the popularity of smart contract platforms like Ethereum and DeFi (decentralized finance) projects on them, it becomes clear that general programability (i.e. smart contracts) is a very desirable feature of DLT networks. Naturally, privacy is a desirable feature of smart contracts as well.

If we look for privacy solutions on smart contract platforms we can find various projects and techniques which allow hiding part of the information in transactions and contract state from the public eye [6, 24, 33]. We can also find projects, which sacrifice decentralized, permissionless nature of typical smart contract platforms to achieve better privacy [6, 10, 22, 26] or projects which use hardware enclaves in order to allow validating transactions without seeing their full decrypted contents [2, 14, 23]. As will be explained later (section 1.1), while all of these approaches represent valuable advances in terms of privacy, they all introduce their own limitations and costs. What is still desirable is a decentralized permissionless smart contract platform, which would by default ensure privacy of transactions and the whole state of every smart contract, without depending on any specialised hardware. The purpose of this paper is to introduce a cryptographic construction that would move us closer towards the aformentioned goal.

1.1 Related work

A large portion of work on privacy in DLT space focuses on specific use cases, the most common of them being payments. *Monero* [3] is a blockchain based decentralized cryptorcurrency, which hides origin, destination and amount of currency in transactions using ring signatures [29]. *Zerocash* [32] is another blockchain based alternative with similar guarantees achieved using *zero-knowledge Succinct Non-interactive ARguments of Knowledge* (zk-SNARKs). A modified version of Zerocash protocol is now deployded as *ZCash* [20].

A number of projects focus on improving privacy of existing smart contract platforms. *Zether* [11] implements confidential payments as a smart contract on *Ethereum* [36], using zero-knowledge proofs. It is interoperable with other smart contracts on Ethereum, which allows it to improve privacy of a number of other decentralized applications, but its privacy guarantees are limited to payments. *zkay* [33] is a smart contract programming language with special private types, allowing it to define owners of certain values. It generates Solidity smart contracts for Ethereum which implement the specified logic and privacy guarantees. It uses proofs generated using *Zokrates* [15] - a frameowork which compiles Ethereum smart contracts into zero-knowledge circuits. All of these projects on Ethereum use zero-knowledge proofs, and introduce additional cost for transactions in terms of resources needed to generate proofs and to verify them on-chain. Furthermore, they do not hide the smart contracts that tranactions interact with. For example, a token implemented using Zether would hide the details of payments which use this token, but would not hide which transactions use this token [8].

Kachina [24] proposes a model and a method for developing privacy-preserving smart contracts. Similarly to zkay, this model allows specifying which data should be private and which can be leaked. It further provides proof guaranteeing that all smart contracts satisfying the requirements of the model will not leak anything else besides the data specified to be leaked explicitly. Kachina solution is also based on zero-knowledge proofs, meaning all smart contracts based on it will have to use zero-knowledge proofs. In Kachina as well as in zkay, the privacy guarantees depend on the actual smart contract and how it is implemented. Meaning privacy guarantees are different for different smart contracts.

Some projects depend on centralized parties to ensure privacy. *Hawk* [26] uses MPC (multi party computation) to perform decentralized, private and fair computation but its privacy guarantees depend on a centralized manager being honest and not revealing private data. *Arbitrum* [22] describes how to implement off-chain execution of smart contracts, by a committe of managers. In the optimistic case an agreement of all members of committe on the output of the computation is enough. In case of a dispute an on-chain protocol is run to resolve the dispute and penalize dishonest parties. The drawback is that all the members of the committe have to be honest in order to ensure privacy.

Enigma [2], Ekiden [14] and Kaptchuk et all [23], propose using hardware enclaves (such as *Intel Software Guard extensions*) in order to ensure privacy. While this approach offers good performance [24], the dependence on specialised hardware means that security depends on this hardware not having any security flaws, and a number of them have already been found over the years [34, 35].

Smart contract platforms like *Quorum* [6] and *Corda* [10] focus on privacy as well, but, unlike the rest of projects mentioned here, they are aimed at permssioned networks. Privacy in Corda depends on trade-off relating to security [25]. We cover this issue in section 2.4.

1.1.1 UTXO-based smart contracts

Corda uses an *UTXO (unspent transaction output) model* for smart contracts, which we use as a basis for the protocol presented here. This model is an alternative to an *account-based model* that Ethereum uses [12]. In the *account-based model* application data of every smart contract is stored in the same global ledger state, shared by every node in the network, whereas in UTXO-based model the state can be partitioned, which enables a lot more flexibility in terms of privacy. This model is described in detail in section 2. UTXO-based model does not fascilitate expressiviness as easily as the account-based model, which is why Ethereum and many other smart contract platforms choose the account-based model [12]. However, as existing work shows, UTXO-based smart contracts are possible and offer privacy benefits [17], as well as better security against programming errors, which are very easy to make in the account-based model because of the mutable, shared state [12].

Zexe [8] proposes a decentralized private computation platform. It uses UTXO-based model for smart contracts as well but unlike Corda it is not aimed solely for permissioned setting. It also differs from most of the projects mentioned earlier in that it hides smart contracts associated with each transaction as well as the data in it. In fact, transactions in Zexe only leak an upper bound on the number of consumed inputs and created outputs. It achieves this, by having all transactions carry zkSNARK proof with them to prove their validity, without leaking much else. These proofs are then verified by consensus protocol. Usage of zkSNARKs of course, has a cost. It was measured to take around 52 seconds to create an average transaction in Zexe and around 46ms to verify it [8].

1.1.2 Overlay protocols

Most privacy solutions and most DLT platforms in general use consensus protocol for ordering as well as validation of transactions, so that a sequence of only valid transactions is produced. However, there are some projects which separate transaction ordering from validation and use consensus protocol only for transaction ordering. *Hyperledger Fabric* [5] and *Corda* [17] are examples of decentralized application platforms which take this modular approach. To understand why this separation is possible, consider a set of deterministic state machines. If all of them are given the same sequence of inputs they will innevitably produce the same outputs. It follows that in order for the nodes of decentralized application to reach consensus on the whole state of an application, it is enough for them to have consensus on the sequence of inputs, which can be determined by a separate process. More formally, it has been shown that consensus and atomic broadcasts are reducible to each other [13]. *Atomic broadcast* or *total order broadcast* is a *broadcast where all correct processes receive the same set of messages in the same order* [13] and the fact that consensus and atomic broadcast problems are reducible to each other means that a solution to one problem yields a solution to the other. So if we have atomic broadcast protocol, we can derive a solution to achieve consensus as well.

Another set examples of decentralized applications which leverage atomic broadcast - consensus equivalence come in a form of *overlay protocols*. They use blockchains as an atomic broadcast protocol and execute all the application logic off-chain [37]. Blockchains often allow additional metadata to be submitted with transactions. This data can be totally arbitrary, as blockchain nodes ignore it while validating transactions. Overlay protocol transactions are embedded into blockchain transactions by using this metadata field. Overlay application nodes can use it to build their application state. Of course, they need to apply additional rules to filter only valid overlay protocol transactions.

One of the first examples of overlay protocols came in the form of *colored coins* [31]. They use Bitcoin blockchain in order to implement "coins" representing real world assets. *Virtualchains* [28] and *Counterparty* [1] generalizes this idea by allowing one to implement any kind of smart contract this way. Virtualchains has been used to create a global naming and storage system [4], while Counterparty implements a platform for Ethereum smart contracts by using Bitcoin blockchain for ordering transactions and Ethereum virtual machine for executing smart contracts. However, none of these overlay protocols offer any privacy.

Dialektos, which is presented in this paper, can be constructed as an overlay protocol for some blockchain (or other kind of consensus protocol), extending it with privacy-preserving smart contracts.

1.2 Contributions

This paper proposes *Dialektos - a privacy-preserving distributed ledger protocol for UTXO-based smart contracts*. More concretely, we make the following contributions:

- **UTXO-based smart contract model** - we formalize the notion of *UTXO-based smart contracts*, which is inspired by smart contract platforms like Corda, and which we will use in later parts of the work.
- **UPC scheme** - we introduce the notion of *UTXO-based private computation (UPC) scheme*, which captures functionality and security guarantees of a protocol for execution of UTXO-based smart contracts. Realizations of this scheme can execute any smart contract, which satisfies the *UTXO-based smart contract model*, while hiding all information relating to any transaction from the public, except the public keys of the parties interacting, size of a transaction and references to spent states (outputs of previous transactions), that are intelligible only to parties which have access to full transactions that created these states. For any transaction t, only parties, which are participants of t (parties to whom t is relevant) or some other transaction, which uses t as a dependency, gain access to t. These guarantees will be explained in more depth in section 3.
- **Dialektos** - we provide a construction of a UPC scheme, called Dialektos and provide proofs that this construction satisfies the security definition of a UPC scheme.

Additionally, the Dialektos protocol exhibits the following features, which distinguish it from related work:

Overlay protocol. As was already briefly mentioned in the previous section, Dialektos protocol can be viewed as an overlay protocol over consensus protocol (blockchain or other kind of DLT network). It separates transaction ordering which is done by the consensus protocol from transaction validation which is done by Dialektos. This separation is partly what enables privacy features of Dialektos, as the fact that public and permissionless consensus protocol is only used for ordering of transactions means, that it can do so without having plaintext access to transactions. Also, the fact that consensus protocol is not responsible for transaction validation, minimizes the amount of processing that consensus protocol has to do, which has positive implications for scalability.

No dependence on complex cryptographic primitives. A lot of privacy solutions, mentioned earlier, use some kind of zero knowledge proofs (like zkSNARKs). These novel cryptographic schemes have performance cost, especially when it comes to proof generation, and are often based on hardness assumptions, which are still considered to be strong [30]. The protocol proposed here only uses encryption, hash functions and digital signature schemes.

No dependence on specialised hardware. As was mentioned earlier, some of the privacy solutions for DLT networks use hardware enclaves like Intel SGX, to achieve their privacy guarantees. Dialektos does not depend on any specialised hardware for security.

No dependence on a centralized party / parties. Dialektos does not depend on any centralized party.

Programming language agnostic. A protocol realizing a construction presented here could be used for smart contracts implemented in any programming language, as long as it produces deterministic code to run when validating transactions.

Permissionless. Dialektos can be fully public and permisionless, but there is also nothing preventing it from being permissioned.

Smart contract interoperability. In Dialektos the same transaction can interact with multiple smart contracts and these smart contracts can take into account the inputs given to other smart contracts, besides itself.

Same privacy guarantees for all smart contracts. Dialektos is a general purpose protocol for smart contracts. All smart contracts executing on it inherit the same privacy and consensus guarantees.

Solution for security privacy trade-off of Corda. UTXO-model used by Dialektos is almost the same as the one used by Corda. Indeed, Dialektos could be seen as a modification of Corda, which replaces notaries (special parties in Corda, which have a responsibility of preventing double spends) with some protocol which produces a total order of transactions. The important effect of this is that resulting protocol does not require *security-privacy trade-off* that exists in Corda [25]. Section 2.4 explains the problem presented by this trade-off.

2 UTXO-based smart contract model

Dialektos is a protocol for UTXO-based smart contracts. Concepts of UTXO smart contract model are deeply built into the rules of Dialektos protocol, therefore it is necessary to understand how this smart contract model works first. Corda is one of the smart contract platforms, which use UTXO-based model [17]. The model presented in this section is inspired by it. First we will present a short overview of how UTXO-based smart contracts work, then a more formalized data model will be presented, which will be used later when defining Dialektos protocol. Finally, some concepts relating to the data model will be defined.

2.1 Overview

The concept of a UTXO (unspent transaction output) transaction model was first introduced by Bitcoin [17]. In this kind of model each transaction creates a set of outputs that future transactions spend. In Bitcoin, outputs are amounts of Bitcoin cryptocurrency (or coins). So every transaction spends a set of coins (*inputs*) and creates a new set of coins (*outputs*). Of course, the protocol has a built-in rule that the total amount of cryptocurrency created is not more than the amount of cryptocurrency spent and each output can only be spent once. Additionally, each output in bitcoin can be associated with a script that defines rules when this particular output can be spent. This scripting language however is very limited.

Some ledger platforms extend this concept of UTXO in order to enable general purpose smart contracts [12, 17]. Besides providing a more powerful programming language to determine when outputs can be spent, they allow defining arbitrary types for outputs. This means that these outputs can be used to represent arbitrary assets or other parts of application state and not just amounts of some cryptocurrency.

In Corda, and in the model we use here, transaction outputs are called *states*. Smart contract developers can define arbitrary types to use as states. For example, a state type representing an ownership in some piece of digital art could be created, where this type would consists of at least three fields: an owner, a hash of a file of an art piece, and a link to this file. In order to transform a state (e.g.: to transfer ownership of an asset), a transaction would be created which has old state as input and has a new state, with modified fields, as output. Besides defining state types, smart contracts also define a verify function. This function is called when verifying a transaction that spends or creates states of the type defined by this smart contract. It determines if a transaction is valid according to smart contract. Continuing with our example, a verify function for a smart contract of digital art piece would check if an output state (representing a new ownership) would have the same values for hash and link fields as the input state had (owner field could be different), and that transaction is signed by the public key specified as the owner in the input state (old owner of an art piece). This is how verify function would work in case of a transaction which intends to transfer ownership of an art piece. Smart contract might also want to allow other types of functionality, like destroying the digital asset. For this purpose, smart contracts also define command types. Each transaction which spends states of this smart contract also defines a list of commands, which specify an intention of this transaction. The verify function of a smart contract will usually apply different rules for transactions with different commands.

2.2 Data model

We define UTXO-based smart contract to be a tuple:

$$Contract = (stateTypes, cmdTypes, \mathsf{verify})$$

Where *stateTypes* is a collection defining data types for states, *cmdTypes* - a collection defining data types for commands and verify is a function for verifying transactions, which has input or output states of types defined in *stateTypes*. It takes a transaction and returns an output specifying whether transaction represents a valid state transition, according to this contract:

$$\mathsf{verify}(tx) \to \top/\bot$$

The argument tx here is of type Tx, which is a tuple:

$$Tx = (inputRefs, outputs, cmds)$$

The $inputRefs$ field specifies a set of inputs that transaction spends. More specifically it is a collection of state references to the output states created by previous transactions. The $cmds$ field specifies commands - values of type $Command$ (defined below), which represent intent of a transaction. Output states ($outputs$) are of type $State$, defined below. This is where the actual application specific data resides.

Each element in $inputRefs$ field of Tx is of $StateRef$ type, which is a pair:

$$StateRef = (stxId, index)$$

Here $stxId$ is a hash of some signed transaction and $index$ is an index refering to a specific state in the $outputs$ collection of the signed transaction that $stxId$ refers to.

Each element of $outputs$ collection in a Tx is a $State$, which is a tuple:

$$State = (partPKeys, contractRef, stateData)$$

Here $stateData$ is application specific data. Smart contracts define the type for this field in the $stateTypes$ field of $Contract$. Next, $contractRef$ is a reference to a smart contract (type $Contract$ above) which defines valid state transitions for this type of state. The $partPKeys$ contains a collection of public keys identifying parties for whom this state is relevant. In the UTXO model states often belong to a single party (in which case he is the sole owner of a state) or a limited set of parties. All other parties, typically do not need access to the particular state. The $participant$ field is meant to make determining who should gain access to the state easier.

Tx also contains $cmds$ field which is a collection of $Command$s:

$$Command = (cmdData, signers)$$

The $cmdData$ field specifies what state transition is intended. Possible values are determined by smart contracts in their $cmdTypes$ field. Additionally, each $Command$ contains $signers$ field, which specifies public keys authorizing this transaction.

When transactions are signed they are stored in $SignedTx$ tuple:

$$SignedTx = (tx, sigs, salt)$$

The tx field is the actual transaction (Tx) defined above, $sigs$ contains the mapping between the public keys and their signatures on tx. The salt field is for making the hashed value of signed transaction unpredictable even when knowing transaction and its signatures.

2.3 Model concepts

Here we define some concepts, which will be used later.

Input and output states. Transactions contain references to states created by previous transactions. This paper will sometimes refer to these states as *input states* of a transaction. Accordingly, the states in $outputs$ field of a transaction can be called *outputs of a transaction*.

Spent state. A state which is an input state to some valid transaction (more on transaction validity later in this section), is termed *consumed* or *spent*.

Participants of a transaction. As was already mentioned, $partPKeys$ field of $State$ contain public keys of parties to whom the particular state is relevant (who should gain access to the state). We call these public keys *participant public keys of state s* and we call parties to whom these public keys belong - *participants of state s*. We also, introduce related concepts - *participant public keys of transaction* and *participants of transaction*. The former refers to the set of public keys which are participant public keys of any of the states in transaction (either input states or output states) and the latter refers to the set of owners of these public keys. More formally, for any transaction t, with input states $i_0, ..., i_n$ and output states $o_0, ..., o_m$, participant public keys of t are given by function txPart:

$$\mathsf{txPart}(i, o) = \bigcup_{k=0}^{n} \mathsf{stPart}(i_k) \cup \bigcup_{k=0}^{m} \mathsf{stPart}(o_k)$$

Where, $\mathsf{stPart}(s)$ returns $partPKeys$ field of state s.

Transaction dependencies. For any transaction t, other transactions which are referenced in $inputRefs$ field of t (meaning that t spends outputs created by those transactions) are called *direct dependencies of* t. Direct dependencies of t have their own direct dependencies and so on, producing *a tree of dependencies* of t. All transactions in that tree are called *dependencies* of t. All dependencies of t which are not among its direct dependencies are called *indirect dependencies* of t.

Conflicting transactions. Two transactions t_1, t_2 are said to be *conflicting* if their input state references overlap. In other words, transactions are conflicting if they try to spend the same state.

Target smart contracts. As should be clear from the previous section, each input or output state of transaction is associated with some *stateType* of some $Contract$. This means that each transaction is associated with a set of smart contracts. This set of smart contracts is called *target smart contracts* of transaction.

Transaction validity rules. For a signed transaction t, in a UTXO-based smart contract model to be valid, this minimum set of rules has to be satisfied:

1. For every target smart contract $(*, *, \mathsf{verify}')$ of transaction t, verify' function must pass: $\mathsf{verify}'(t) = \top$.
2. *sigs* field of t contains a signature for all public keys declared as signers of any of commands in the *cmds* field, and all of these signatures have to be valid.
3. All of dependencies of t are also valid.

A protocol that implements this smart contract model can extend this set of validity rules. One of the reasons to do so is to determine consensus on conflicting transactions. Any sensible protocol that implements this model would need to prevent double spends, which would happen if multiple conflicting transactions would be allowed to be valid. Therefore, some additional rules would normally be needed that select one transaction out of multiple conflicting. In Corda this takes the form of checking if transaction has a signature of a notary. *Notaries* are special nodes in Corda, which have responsibility to record every state that has been spent and only sign on transactions which spend states which have not been spent yet.

2.4 Privacy and security in Corda

Corda is a smart contract platform, which implements UTXO-based smart contracts, which match the model just presented (maybe not exactly, but well enough for our purposes). Dialektos, which will be presented in the following sections can be seen as a modification of Corda, which solves *security and privacy trade-off* [25] present in Corda. Therefore, we introduce this and related problems here, for completeness. This section is not essential for understanding the rest of the paper, but might help because Corda is a simpler version of a protocol for UTXO-based smart contracts.

One consequence of UTXO model is that the state of the whole application can be partitioned. For example, if we take cryptocurrency as an application, then its whole state can be defined as a table containing balances of each account. In Corda, however, the way tokens are implemented, there is no such table stored anywhere, there are only *states* defining token amounts and who they belong to. It is responsibility of an owner himself to track all of the token states he owns. This means, that transactions do not need to be broadcasted to every node in the network. *State* has a *participants* field to help with that. It defines all the parties for whom this state is relevant. Normally, every transaction in Corda is sent to the set of parties specified by the union of all participants of input and output states (typically transaction creators to distribute transactions to participants). This way transaction is recorded only by participants of transaction instead of the whole network, which preserves the privacy of transaction.

However, this mechanism does not provide any guarantees about transaction (and all of the states and identities recorded in it) staying private in the future. More specifically, future transactions which spend the states created by the previous transaction has to reveal previous transaction and the whole chain that lead to it, in order to prove their validity. Corda offers a technique for helping with this issue, called *confidential identities* [17]. The idea is to generate a new key pair for every new transaction (or just the ones for which confidentiality is important) a node signs. It can prove that this new key pair belongs to him, by using a simple interactive protocol. So if Alice wants to send tokens to Bob, but Bob does not want for future validators of this transaction to see that the tokens went to him, he asks Alice to send tokens to a key pair which is freshly generated by him. Alice runs interactive protocol to verify that Bob owns these keys and then does as Bob asked. Alice knows tokens went to Bob, but for any other party which gets this transaction, the receiver of tokens will not be known. This way at least the anonimity of the transaction is preserved when it used as a dependency.

Notaries have important implications for transaction privacy. Corda network can be configured to have validating or non-validating notaries. *Validating notary* validates all the transactions before signing them (in addition to checking if it does not try to spend states which are already spent), while *non-validating notary* sees only input state references, time window and notary identity parts of transactions. Non-validating notaries are possible because it is enough to see state references in order to prevent double-spends. Non-validating notaries are obviously better in terms of privacy, because most of the transaction data is not revealed to them. However this comes at a cost of susceptibility to denial-of-state (DoSt) attacks [25]. They work by an attacker sending to the notary an invalid transaction which claims to consume some state. Since notary is non-validating, he will sign it. When a genuine owner of the state tries to create a valid transaction, which spends the state, the notary will reject it, because in his eyes it is already consumed. Besides the fact that this attack prevents a valid state spend, it is also invisible to honest nodes, until they attempt to spend the state.

To the best of authors knowledge, there is currently no satisfiable solution to DoSt attack, when using non-validating notaries. So it remains a trade-off between security of validating notaries and privacy of non-validating notaries [25]. Although, it has to be noted, that permissioned nature of Corda networks make it somewhat less likely, because attackers identity will be seen once attack is discovered. Then network operators can manually fix the ledger and take steps against the attacker.

In the sections that follow, we provide a security definition of a protocol for UTXO-based smart contracts, which does not have this security-privacy trade-off and later provide a construction for such algorithm.

3 UTXO private computation scheme

This section defines *UTXO-based private computation (UPC)* schemes. The purpose of these schemes is to execute transactions for UTXO-based smart contracts that were defined in the previous section, while preserving privacy and other security guarantees. First, an overview of data structures will be presented, interface (algorithms) for these schemes will be defined, then informal definition of security properties will be provided and finally, a formal security definition will be introduced.

3.1 Data structures

UPC schemes use the structures from the UTXO-based smart contract model defined in section 2. Additionally, it introduces the concept of a *ledger*. A ledger in UPC is represented by *StxMap* type, which is a table, mapping signed transaction identifiers (their hashes) to their signed transactions:

$$StxMap = Map[StxId, SignedTx]$$

Informally, the ledger should hold all signed transactions, which are recorded by the party as having been executed by the network. Since UPC is privacy-preserving, ledgers of different parties will usually differ: each party is only aware of transactions that he is a participant of.

3.2 Algorithms

A UPC scheme is a tuple of algorithms:

$$UPC = (\mathsf{NewKeyPair}, \mathsf{UploadContract}, \mathsf{GetStxs}, \mathsf{SubmitTx})$$

Here are the interfaces of the algorithms and their intent described informally. We assume security argument to be an implicit for all the algorithms.

Key generation. $\mathsf{NewKeyPair}() \rightarrow pk$. Each user who uses the protocol will have an associated set of keypairs generated by some public key encryption scheme. Public keys will be used for identifying transactions, which are relevant to the user. This algorithm generates a keypair and returns its public key.

Uploading contract. $\mathsf{UploadContract}(c) \rightarrow \top/\bot$. This algorithm is used for registering smart contracts, which transactions can interact with (as will be seen in security definition section, transactions referencing contracts which were not registered are invalid). It takes a value of *Contract* type as argument and returns whether the registering of smart contract was successful.

Submitting a transaction. $\mathsf{SubmitTx}(stx) \rightarrow \top/\bot$. This algorithm takes a signed transaction (type *SignedTx*) as input, verifies it against the current state of the ledger and if it is valid, stores it as another valid transaction in the ledger of the party that submitted it. Returns $\top$ if it was successful and *stx* was stored in the ledger.

Returning recorded transactions. $\mathsf{GetStxs}() \rightarrow stxs$. User may invoke GetStxs to retrieve the current state of *his* ledger. This algorithm takes no arguments and returns *StxMap* value. When *GetStxs* is invoked through party p, then the value returned is said to be a *ledger of p*.

3.3 Security definition

For a cryptographic scheme to be considered *a secure UPC scheme*, it has to satisfy certain security requirements. Informally, these are the properties we seek:

- **Consensus** - for every transaction *stx*, all honest participants of *stx* reach consensus regarding whether to include it in the ledger or not (they all include *stx* in their ledger or they none of them do). In other words, if *GetStxs* returns *stx* as part of a ledger for some honest party, then all participants of *stx*, get *stx* in their ledger as well (when they invoke *GetStxs* they get *stx* in the value returned).

- **Integrity** - for any transaction *stx* and any party p if $SubmitTx(stx)$ returns $\top$ (as opposed to $\bot$), then any subsequent call to *GetStxs* by party p returns *stx*.

- **Correctness** - for every honest party p and for every transaction *stx* in its ledger, the following must hold:

  - For every target smart contract $(*, *, \mathsf{verify}')$ of transaction *stx*, verify' function must pass: $\mathsf{verify}'(stx) = \top$.
  - *sigs* field of *stx* contains a signature for all public keys declared as signers of any of commands in the *cmds* field
  - *stx* spends every state at most once
  - All the dependencies of *stx* must also be in the ledger of p

- **Uniqueness** - for every honest party p and for every transaction *stx* in its ledger, there exists no other transaction in the ledger of p, which spends any of the same states.

- **Privacy** - for every transaction *stx*, that is successfuly submitted by some honest party, only the following information is revealed and only to the specified parties:

  - *Participants* of *stx* gain access to *stx*.

- *Participants* of *stx* gain access to every dependency of *stx*.
- *Participant public keys* of *stx* are revealed publicly.
- For any *direct dependency* stx_0 of *stx*, *participants of* stx_0 are revealed the output states of stx_0 that *stx* spends.
- Size of *stx* is revealed publicly.

A formal security definition for UPC consists of two parts:

- A game-based definition for consensus, which models dynamic adversarial setting where adversary can get access to any of the protocol machines and send any (potentially dishonest) messages through them.
- A simulation-based definition in a standalone model with static, passive-but-curious adversaries.

The game-based definition provides stronger guarantees, but only for consensus. The rest of the informal properties defined above are proven in a weaker standalone model for now.

3.3.1 Consensus

We define consensus property that a secure UPC construction has to satisfy, as a game G_c between adversary $\mathcal{A}$ and a challenger $\mathcal{C}$. A challenger in this game interacts with a set of parties $\mathcal{P}$ running some protocol π. The challenger provides the following interface for the adversary:

- $\mathcal{C}.\mathsf{GetPartyState}(p) \rightarrow \{0,1\}^*$ - challenger returns the whole state of party $p \in \mathcal{P}$.
- $\mathcal{C}.\mathsf{NewKeyPair}(p) \rightarrow PKey$ - challengers sends $\mathsf{NewKeyPair}()$ to $p \in \mathcal{P}$ and forwards the returned public key to $\mathcal{A}$.
- $\mathcal{C}.\mathsf{UploadContract}(p,c) \rightarrow \top/\bot$ - challenger sends $\mathsf{UploadContract}(c)$ to $p \in \mathcal{P}$ and forwards the response to $\mathcal{A}$
- $\mathcal{C}.\mathsf{SubmitTx}(p,stx) \rightarrow \top/\bot$ - challenger sends $\mathsf{SubmitTx}(stx)$ to $p \in \mathcal{P}$ and forwards the response to $\mathcal{A}$.
- $\mathcal{C}.\mathsf{GetStxs}(p) \rightarrow StxMap$ - challenger sends $\mathsf{GetStxs}()$ to $p \in \mathcal{P}$ and forwards returned value to $\mathcal{A}$.
- $\mathcal{C}.\mathsf{SendMsg}(p,r,m)$ - challengers sends m to party r, through party $p \in \mathcal{P}$

The last function supplied by the interface of the challenger allows $\mathcal{A}$ to send any message to any party in a way that it would seem to the recipient that message came from a protocol party p. Note that recipient could be anyone, not just the party of protocol π.

Adversary can call any of the functions challenger provides. He wins consensus game if there exists a transaction s and two parties p_1, p_2, which would be participants of s, but s would be accepted by one party and not accepted by the other. More formally:

Definition 1 (**Adversary winning conditions for consensus**). *Adversary wins Gc defined above if, there exists two parties p_1, p_2 with corresponding public keys pk_1, pk_2, a transaction $s \in SignedTx$, a sequence S of n other transactions $\{s_0,...,s_n\}$ and a sequence K of State values $\{t_0,...,t_n\}$, such that the following four statements would hold:*

1. $\forall i < n$: $s_i.tx.outputs[s.tx.inputRefs[i].index] = t_i$
2. $\forall i < n$: $\mathsf{hash}(s_i) = s.tx.inputRefs[i].stxId$
3. $pk_1, pk_2 \in \mathsf{txPart}(K, s.tx.output)$
4. $s \cup S \subseteq \mathcal{C}.\mathsf{GetStxs}(p_1) \wedge s \cup S \nsubseteq \mathcal{C}.\mathsf{GetStxs}(p_2)$

Definition 2 (**UPC consensus**). *A protocol π implementing a UPC scheme is said to achieve consensus if the probability that $\mathcal{A}$ wins game Gc is negligible.*

3.3.2 Ideal functionality emulation

For privacy and other desirable properties listed at the beginning of section 3.3 we create a simulation-based definition of security, which we model as follows.

Each game is defined for a tuples $(\mathcal{E},\pi,\mathcal{A})$, where $\mathcal{E}$ is some PPT machine representing environment, π is some PPT protocol, $\mathcal{A}$ is a PPT machine representing an adversary. We denote a game n for some $(\mathcal{E},\pi,\mathcal{A})$ as $G_n^{\mathcal{E},\pi,\mathcal{A}}$. $\mathcal{F}_{UPC}$ is ideal functionality for UPC protocol, as described in appendix A.

In game $G_r^{\mathcal{E},\pi,\mathcal{A}}$, representing the real world, $\mathcal{C}$ runs parties P of protocol π and simply forwards each message from $\mathcal{E}$ to some $p \in P$, where p is specified by the environment. Responses from protocol parties are returned to $\mathcal{E}$. Some number $c <= |P|$ of protocol parties are corrupted. After each activation of one of corrupted parties, challenger sends tuple (pid, msg, out, st) to $\mathcal{A}$, where pid is identifier of a corrupt party p, which was activated, msg is a message from the environment, which activated p, out is a response (to the environment) from p, st is the current state of p. At the end of the game (after the last message was sent by the environment), $\mathcal{C}$ returns state of $\mathcal{A}$ to $\mathcal{E}$. $\mathcal{E}$ then outputs 1 bit value.

In game $G_i^{\mathcal{E},\pi,\mathcal{S}}$, which represents the ideal world, $\mathcal{C}$ runs ideal functionality $\mathcal{F}_{UPC}$ and forwards each message from $\mathcal{E}$ to it and returns all outputs of $\mathcal{F}_{UPC}$ back to $\mathcal{E}$. $\mathcal{F}_{UPC}$ can leak certain data to adversary. We call these messages - backdoor messages. All of these messages are collected by $\mathcal{C}$. At the end of execution, $\mathcal{C}$ calls some PPT simulator $\mathcal{S}$ passing it all the collected backdoor messages. Output of $\mathcal{S}$ is then passed to $\mathcal{E}$. $\mathcal{E}$ then outputs 1 bit.

Informally, the task of $\mathcal{S}$ in this game is to generate state of $\mathcal{A}$ and pass it as output, such that it would not help $\mathcal{E}$ to distinguish if $G_i^{\mathcal{E},\pi,\mathcal{A}}$ was run or $G_r^{\mathcal{E},\pi,\mathcal{A}}$, for any $\mathcal{E}$.

We denote a bit produced as output by $\mathcal{E}$, after execution of some game g as $EXEC(g,k)$, where k is a security parameter used in execution.

Definition 3 (**Emulation of ideal UPC functionality**). *Protocol π is said to emulate UPC ideal functionality $\mathcal{F}_{UPC}$ if for any PPT adversary $\mathcal{A}$, there exists such PPT simulator $\mathcal{S}$, that for any PPT environment $\mathcal{E}$, the following would hold:*

$$\{EXEC(G_r^{\mathcal{E},\pi,\mathcal{A}},k)\}_{k\in\mathbb{N}} \overset{c}{\equiv} \{EXEC(G_i^{\mathcal{E},\pi,\mathcal{S}},k)\}_{k\in\mathbb{N}}$$

where $\overset{c}{\equiv}$ denotes computational indistinguishability of probability ensembles in [27].

Definition 4 (**Secure UPC protocol**). *A protocol π implementing a UPC scheme is said to be secure if it achieves consensus and emulates ideal functionality $\mathcal{F}_{UPC}$.*

4 UPC construction

This section introduces a specific construction of a UPC scheme, called *Dialektos*.

The design for this protocol is based on the *consensus - atomic broadcast equivalence* [13]. Instead of using consensus protocol for validating all transactions, we use it only for ordering of transactions. That is, we use it as an atomic broadcast protocol.

Atomic broadcast protocols (or *ABP*s in the remaining text) are simply a set of protocols that ensure that all the parties agree on the set of messages that were received and their order. As already mentioned in section 1.1.2, atomic broadcast and consensus problems are reducible to each other, which means that Dialektos can use almost any consensus protocol (including existing blockchains) as an atomic broadcast protocol to achieve the needed transaction ordering. The only requirements are:

1. The same ABP must be used to spend the state as was used to create it.

2. The chosen ABP must allow embedding arbitrary data of arbitrary size in its messages (or transactions for, say, blockchain protocols).

There might be ways to overcome the first limitation - to migrate some states to other atomic broadcast protocol, but this is a topic for further research. The second requirement enables Dialektos to embed its transactions in ABP messages.

If consensus protocol (or ABP) is not used for validating transactions, but only for ordering that means that transactions contents are not that important for this protocol. It might as well be ordering encrypted transactions, which is the approach taken by Dialektos in order to achieve privacy.

Parties running Dialektos protocol apply a set of rules to transactions ordered by ABP protocol to determine which of them are valid. These rules are called *validity rules* of transactions in Dialektos. They are the essence of Dialektos protocol. Once parties have consensus on a sequence of valid transactions they can reach consensus on the application state, by executing transactions using deterministic state machines.

Validity rules of transactions have to satisfy some requirements, most of which can be inferred from the ideal functionality presented in appendix A. First and foremost, they have to be deterministically computable, so that two Dialektos parties validating the same transaction reach the same conclusion regarding its validity. Second, these validity rules have to resolve conflicts (cases where to transactions spend the same input), in order to prevent double-spends. Third, we want to preserve privacy. For this purpose, all transactions submitted to ABP network are encrypted. This makes the consensus requirement harder to achieve, but we use the fact that in the privacy-preserving smart contract platform parties do not need to achieve consensus on the whole state, they only need consensus on the part of the state that is relevant to them (if everyone would have consensus on the whole state, there could be no privacy). This is where the UTXO model fits perfectly: transaction outputs (and transactions themselves) are revealed only to the relevant parties. Therefore, parties in Dialektos do not achieve consensus on validity of all transactions, but only on transactions which are relevant to them. This brings us to another requirement: Dialektos has to ensure that parties are able to get access (decrypt) all transactions which are relevant to them, in order to determine their validity. The protocol described in the following sections is constructed according to these requirements.

In the following sections we first introduce cryptographic building blocks, data structures of Dialektos and then transaction validity rules. These validity rules determine both how transactions should be validated as well as how to create a valid transaction, so in a sense, they are the essence of the protocol. The more detailed pseudocode specification for validity rules and the rest of the protocol is in appendix B.

4.1 Building blocks

Here we introduce syntax for some cryptographic building blocks that Dialektos uses. Below, k represents a security parameter.

Collision-resistant hash function. $\mathsf{hash}(v) \to h$. Takes a value v of arbitrary size and returns its hash. It is compuationaly infeasible to find two distinct values v, v', such that $\mathsf{hash}(v) = \mathsf{hash}(v')$.

Deterministic asymetric encryption. $\mathsf{AEnc} = (\mathsf{Kg}, \mathsf{E}, \mathsf{D})$. Kg generates a keypair containing public key and secret key: $\mathsf{AEnc.Kg}(1^k) \to (pk, sk)$. $\mathsf{AEnc.E}(pk, m) \to c$

encrypts a message $m \in M_k$. $\mathsf{AEnc.D}(sk, c) \rightarrow m/\perp$ decrypts ciphertext c using secret key sk. We assume AEnc is deterministic (meaning same plaintext always encrypts to the same ciphertext) and is PRIV-IND-MU secure as defined in [9]. PRIV-IND-MU, like its predecessor, PRIV [7] defines security for deterministic asymetric encryption, such that if the plaintext is sampled from high enough min-entropy it is secure against *chosen-plaintext attacks*. PRIV-IND-MU extends PRIV security definition to multi-user, multi-message setting, where adversary can potentially have auxilary information relating to the plaintext.

Authenticated symetric encryption. $\mathsf{SSenc} = (\mathsf{Kg}, \mathsf{E}, \mathsf{D})$. SSenc is an authenticated symetric encryption scheme, which is one-time secure (AE-OT) as defined in [19]. AE-OT security definition captures privacy (indistinguishability against one-time attacks) and authenticity (ciphertext authenticity against one-time attacks). $\mathsf{SSenc.Kg}(1^k) \rightarrow key$ outputs keys $key \in K_k$, which are uniformly distributed. $\mathsf{SSenc.E}(key, m) \rightarrow c$ encrypts message $m \in M'_k$ using $key \in K_k$. $SEnc.\mathsf{D}(key, c) \rightarrow m/\perp$ decrypts ciphertext c using $key \in K_k$.

Hybrid encryption $\mathsf{HEnc} = (\mathsf{E}, \mathsf{D})$. We use AEnc and SSenc to construct a hybrid encryption scheme HEnc, which works as shown in figure 1. Its E algorithm takes a number of public keys and encrypts a single plaintext with them, using the same symetric key. D decrypts ciphertext using one of c_1, c_2 pairs produced by AEnc.E.

We assume that HEnc is secure against *chosen-plaintext attacks* (i.e.: is IND-CPA secure). We do not show here that such constructions of SSenc, AEnc and HEnc exist, but based on results from Hofheinz et al. [19] and Heranz et al. [18] we conjecture that they do. Note that in our hybrid encryption as defined in 1 the plaintext that is used for symetric encryption is always random (as already mentioned SSenc.Kg returns keys from a uniform distribution). Therefore, the main point of attack for breaking deterministic encryption scheme in IND-CPA/IND-CCA setting (running the same deterministic encryption algorithm with the same inputs as challenger does) is unavailable. In that case AEnc achieves security against chosen plaintext attacks (definition of PRIV-IND-MU [9]). According to [18] security of key encapsulation mechanism against chosen-plaintext attacks is enough to achieve IND-CPA security of a hybrid encryption scheme.

Unforgeable digital signatures. $\mathsf{Sig} = (\mathsf{Kg}, \mathsf{Sign}, \mathsf{Valid})$. Kg generates a keypair: $\mathsf{Sig.Kg}(1^k) \rightarrow (pk_{sig}, sk_{sig})$. Sign creates a signature on some message m: $\mathsf{Sig.Sign}(sk_{sig}, m) \rightarrow s$. V validates some signature s: $\mathsf{Sig.Valid}(s, pk_{sig}, m) \rightarrow \top/\perp$.

```
1: procedure $\mathsf{E}(pks, m)$ for HEnc
2:     $k \leftarrow \mathsf{SSenc.Kg}()$
3:     $c_2 \leftarrow \mathsf{SSenc.E}(k, m)$
4:     $cs \leftarrow []$
5:     for $pk$ in $pks$
6:         $c_1 \leftarrow \mathsf{AEnc.E}(pk, k)$
7:         $cs \leftarrow cs \,\|\, (pk, c1)$
8:     return $(cs, c_2)$
9: procedure $\mathsf{D}(sk, (c1, c2))$ for HEnc
10:    $k \leftarrow \mathsf{AEnc.D}(sk, c1)$
11:    $m \leftarrow \mathsf{SSenc.D}(k, c2)$
12:    if $k \neq \perp \wedge m \neq \perp$ then
13:        return $(k, m)$
14:    else
15:        return $\perp$
```

Figure 1: *HEnc* scheme

Atomic broadcast protocol. $\mathcal{F}_{ABP} = (\mathsf{Submit}, \mathsf{Read})$. In our construction we assume the presence of an ABP protocol ideal functionality we denote by $\mathcal{F}_{ABP}$. $\mathsf{Submit} : ABPMsg \rightarrow \top$ function takes a message of type $ABPMsg$ assigns it a number, records it in the state of $\mathcal{F}_{ABP}$ and leaks this message to adversary. $\mathsf{Read} : \rightarrow ABPMsg[]$ returns all the messages that has been submitted, ordered by their number (the order of submition).

4.2 Data structures

Before introducing validity rules of transactions it is necessary to introduce the data structures that these rule operate on. Dialektos uses data structures of *UTXO-based smart contract model* and UPC schemes defined in the previous sections. Plus it introduces some additional data structures.

Parties running Dialektos protocol interact (exchange messages) only with ABP protocol, by submitting transactions and retrieving transactions from it. In order to create a transaction party creates a $SignedTx$ value, encrypts it, wraps it in a ABP protocol message and sends this message to the ABP protocol. This ABP protocol message is represented by the $ABPMsg$ type:

$$ABPMsg = (num, payload)$$

Here, num is the sequence number given to the message by the ABP protocol. Remember that it is the purpose of a ABP protocol to order messages. So num value represents this order. The $payload$ field can be anything as far as ABP protocol is concerned: it does not validate it. We use it to store our Dialektos transactions.

ABP messages that represent Dialektos transactions have a $payload$ value of type $ABPTxMsg$:

$$ABPTxMsg = (encKeys, inputRefs, etx)$$

Field *etx* stores an encrypted transaction. It is a symetric encryption of $TxMsg$, which is defined below. The purpose of $inputRefs$ is to store state references (values of type $StateRef$ defined in section 2.2), that transaction encrypted in *etx* field spends. The *encKeys* field is a mapping of public keys to ciphertexts:

$$EncKeys = Map[PKey, c]$$

The purpose of *encKeys* field is to provide decryption keys for *etx* to the participants of transaction. Each (pk, c) pair in this mapping should be constructed by taking a symetric key k, that was used when encrypting some $TxMsg$ (defined below) value to *etx*, and encrypting it with *pk* using a *deterministic public key encryption scheme* (why it has to be deterministic will be explained in the section 4.3). This way, only the owner of the private key corresponding to *pk* can decrypt a symetric key, which decrypts *etx*.

The value, which is encrypted as *etx* in *ABPTxMsg*, has a $TxMsg$ type:

$$TxMsg = (stx, depKeys)$$

The *stx* field holds the actual transaction. It is a value of type *SignedTx* defined in 2.2 section. The second field is a mapping from *message identifiers* to symetric keys:

$$Map[MsgId, Key]$$

The identifier (*MsgId*) of an *ABPMsg* is a hash of that *ABPMsg*. In *depKeys* field, each message identifier m should map to a symetric key used to encrypt a transaction that m identifies.

4.3 Transaction validity rules

Every time an honest Dialektos party is activated it checks the ABP protocol for new messages which have one of their public keys in *encKeys* field and downloads them. Then they decrypt them, by first decrypting the symetric key that is encrypted with their public key in *encKeys* field and then using that symetric key to decrypt *etx*. Then they validate the decrypted transaction and add it to their ledger if it is valid. Here we describe the validation part of this process as a set of rules, with rationale for each of the rules. These rules apply to a single value of type *ABPMsg* as well as everything wrapped within it (*ABPTxMsg* and $TxMsg$ encrypted as *etx* within it). If any of the rules fail, the transaction represented by a particular ABP message is considered invalid. A more formal definition of validity rules in a form of pseudocode is in B.

First and foremost, before applying any of the other validity rules Dialektos party has to check that, decrypted *etx* produces value that is indeed $TxMsg$. If it is not, validation fails and ledger is not updated.

Rule 1 (**Transaction format**). *A value computed by decrypting etx field of ABPTxMsg, has to be of type* $TxMsg$.

The decryption might not produce a value of type $TxMsg$ for a couple of reasons. Party submitting a transaction could have provided an arbitrary value for *etx* or it could have encrypted a wrong symetric key for one specific party. The latter case is especially interesting: what happens if a message encrypts different symetric keys for different participants, such that some of them decrypt to valid $TxMsg$ values and some do not? Or what if some participants are not provided by the *encKeys* field with an encrypted value at all? Obviously, we need to make sure that *all* participants decrypt the same value, because we need them to reach consensus on transaction validity. The "Encrypted keys" rule defined below, solves this.

Next we check that transaction *stx* in the $TxMsg$ decrypted, has the authorizations it claims to have.

Rule 2 (**Signatures**). *sigs field of stx in* $TxMsg$ *must contain a signature for all public keys declared as signers of any of commands in the cmds field of stx.tx and these signatures have to be valid signatures of tx by the private keys corresponding to these public keys.*

Later we will check that transaction does not spend states that are already spent by other transactions, but first we check that the same transaction does not spend the same state twice.

Rule 3 (**No duplicates**). *No* $StateRef$ *value can appear in* $inputRefs$ *field of ABPTxMsg more than once.*

Both *ABPTxMsg* and $SignedTx$ within it have $inputRefs$ field, which specifies which states transaction intends to spend. For *ABPTxMsg* this field is needed to detect conflicting transactions (which is needed for "Conflicts" rule below). For $SignedTx$ it is needed to specify whole state transition, which can be signed by the authorizing parties. These collections of input references have to be equal.

Rule 4 (**Input references**). *The* $inputRefs$ *value in stx.tx field of* $TxMsg$*, must be equal* $inputRefs$ *value declared in the ABP message (ABPTxMsg) that carries this* $TxMsg$ *value.*

Input state references and rest of the fields of Tx are not enough to verify the state transition that a particular transaction represents. Actual states that input state references refer to are needed. But these states are part of transactions which are stored encrypted in an ABP network. Therefore, it needs to be ensured that every participant of transaction, has a key to decrypt all of the needed dependencies. This is the purpose of $depKeys$ field of $TxMsg$.

Rule 5 (**Dependencies**). *The* $depKeys$ *field of* $TxMsg$ *has to contain a a symetric encryption key for every direct dependency of stx within the* $TxMsg$ *and all of direct dependencies decrypted this way must be valid*

This means that all the validity rules defined here (including this one) have to be applied to every direct dependency of the transaction being validated. This validates the whole transaction tree. In case any dependencies of transaction is invalid, it is invalid too. This also means, that transaction can be invalid if it provides the wrong dependency keys in the *depKeys* field.

Next, we define a rule to ensure that all participants of transaction get access to it:

Rule 6 (**Encrypted keys**). *encKeys field of ABPTxMsg must contain encrypted value for each participant public key of transaction, and these encrypted values must decrypt to the same symetric key.*

If the same symetric key is encrypted for *all* participants of a transaction and stored in *encKeys* field, then all participants will get the same value when decrypting *etx*, which ensures that they will come to the same conclusion regarding transaction validity when applying the rest of the deterministic rules defined here. In order to enable checking this rule we use *deterministic public key* encryption. It ensures that the same value encrypted with the same public key will always produce the same ciphertext. So to fully check this rule a Dialektos party does the following:

1. Determines participant public keys of transaction *stx* in question, using txPart function (defined in section 2.3 and input, output states of *stx* (input states have to already be retrieved from dependencies).

2. Checks if *encKeys* contains a public key - ciphertext pair for each participant public key from the previous step.

3. Checks that for every (pk, c) pair within *encKeys* the following equation holds: $\mathsf{aenc}(pk, k) = c$, where aenc is *deterministic public key encryption* of the second argument using the public key in the first argument, and k is the symetric key that the validating party used to decrypt the transaction being validated.[1]

Do these checks really translate to the Rule 6? The first two steps ensure that *encKeys* includes all participant public keys which is the purpose of the first part of rule 6. The third step is intended to ensure that *encKeys* maps these public keys to ciphertexts, which are encryptions of the same symetric keys. Let's say we have a symetric key k with which we decrypted *etx* of some *ABPTxMsg*. If *encKeys* field of that *ABPTxMsg* provides encryptions of the same k for all public keys then we should get the same ciphertexts if we encrypt k with these same public keys (otherwise encryption scheme would not be deterministic). So if the check fails either we have the wrong k or *encKeys* provides encryptions of different symetric keys for different public keys. In the second case, the transaction is clearly invalid because it does not follow rule 6. In the first case, we should think about where we get k. If we get it by decrypting a ciphertext provided by *encKeys* for our public key, then we know that rule 6 is not satisfied, because a different key is encrypted for us than from some of the other public keys, and so transaction is invalid. We could have also received k from *depKeys* in *TxMsg* of transaction which uses the transaction in question as a dependency. In that case it is both possible that this key provided by *depKeys* is wrong as well as that *encKeys* is invalid. Fortunatelly, for validation of transaction which provides the *depKeys*, it does not matter which of these two cases is true since the conclusion is the same: the validation of dependencies failed, which means that validation of transaction failed and the ledger should not be updated.

[1] Note that, if the validating party has already come to this validation step it means that it decrypted *etx* field using some symetric key. This is the symetric key that is represented by k in the third step.

Next, we check if *stx* within the *TxMsg* value decrypted represents a valid state transition according to all the target smart contracts:

Rule 7 (**State transition**). *For every target smart contract* $(stateTypes, cmdTypes, \mathsf{verify}')$ *of stx,* verify' *function must pass:* $\mathsf{verify}'(stx) = \top$.

Finally, we need to check if the states being spent by a transaction are not already spent. For that purpose we need to get all conflicts of the transaction we are validating. These are all ABP messages which attempt to spend any of the same states. The *inputRefs* field of *ABPTxMsg* declares what states references a transaction within the message spends, so the protocol uses that to detect all messages representing conflicting transactions. We call the set of states, which are referenced by *inputRefs* of both m and t, *common input states of m and t*. Then the following rule is applied to *TxMsg* we are validating, which we call t:

Rule 8 (**Conflicts**). *For every conflicting ABP message m of t, if the following holds:*

1. *m is recorded in the ABP protocol used to record t*

2. *For every participant public key, of every common input state of m and t, encKeys within m contains a mapping for that public key.*

Then the following must hold too:

1. *depKeys of t must contain a decryption key k for m.*

2. *The validation of m using decryption key k, must fail.*

4.4 Security analysis

Theorem 1. *Dialektos achieves consensus property (def. 2) in a world with ideal functionality* $\mathcal{F}_{ABP}$.

Proof (sketch). A proof by contradiction. Let's say the adversary is successful and there exists two parties p_1, p_2 with corresponding public keys pk_1, pk_2, a transaction $s \in SignedTx$, a sequence S of n other transactions $\{s_0,...,s_n\}$ and a sequence K of $State$ values $\{t_0,...,t_n\}$, such that the four statements in the adversary winning conditions would hold.

If $s \in \mathcal{C}.\mathsf{GetStxs}(p_1)$, that means that there exists some $amsg \in ABPMsg$, such that it is recorded by the ABP network that Dialektos uses and triggered an update to the ledger of p_1 such that s was added to it. This means that $amsg$ and s it contains satisfies all the validity rules according to p_1. All parties use atomic broadcast protocol ideal functionality $\mathcal{F}_{ABP}$, which ensures that if adversary submits $ABPMsg$ to $\mathcal{F}_{ABP}$, then all parties get it the next time they update their state and before handling any other messages. That means that p_2 received $amsg$ too, but it or the $SignedTx$ within it did not pass its validity rule checks or p_2 ignored it, because $encKeys$ in $amsg$ did not contain a public key - ciphertext pair for p_2. In case of validation error p_2 throws an error and stops validating on the first rule that transaction fails. Let's consider each potential cause for consensus failure in p_2 case by case.

Case 1: p_2 ignores $amsg$. This is the case where p_2 does not decrypt $amsg.payload.etx$ because for all public keys pk of p_2, $pk \notin \mathsf{getMapKeys}(amsg.payload.encKeys)$. But, from the hypothesis at the beginning of the proof, we know that p_2 is a participant of s (second statement in the adversary winning conditions), which is within $amsg$. We also know, that honest party p_1 was able to decrypt $amsg.payload.etx$ to s and that p_1 included s in its ledger (4th statemend in the adversary winning conditions). In turn, this means, that s and $amsg$ passed all the validity rules, including rule 6, which states that $amsg.payload.encKeys$ contains a mapping for each participant public key of s. A contradiction: rule 6 cannot be satisfied if participant key of p_2 is not within the $encKeys$.

Case 2: Rule 1 is not satisfied. In this case the value computed by p_2 decrypting $amsg.payload.etx$ field is not a valid value of type $TxMsg$. But we know that it did decrypt to a correctly formatted $TxMsg$ for p_1. The $ams.payload.etx$ is the same for both parties since $amsg$ is the same. Therefore, p_2 must have used a different symetric key for decryption of etx if it decrypts to a different value than what p_1 decrypts. Both parties compute this symetric key by decrypting c from some pair $(pk,c) \in encKeys$, using a private key corresponding to pk. So if parties received different symetric keys, it means that different (pk,c) pairs in $encKeys$ encrypt different symetric key. But this is a contradiction to the rule 6, which means that validation performed by p_1 should have failed and it must not have included s in its ledger. A contradiction.

Case 3: One of the rules 2, 3, 4 or 6 are not satisfied. This is the case where one of these rules do not pass when applied to $TxMsg$ that p_2 decrypts from $amsg.payload.etx$. We know that p_2 used the same symetric key to decrypt etx as p_1, because otherwise rule 6 check would have failed according to p_1 and it would not have included s in its ledger. Therefore, we know that p_2 must apply these rules to the same value s as p_1 did and we know that s satisfies them, because honest party p_1 included s in its ledger. A contradiction.

Case 5: One of the rules 5, 7 or 8 are not satisfied The proof for impossibility of this case is equivalent to the proof of the previous case, except for these three rules some information needs to be retrieved from the ABP protocol. Specifically, a set of ABP messages for rules 5 and 8, as well as a smart contract for rule 7. Smart contracts are uploaded using ABP messages and hence downloading a smart contracts is reduced to retrieving ABP message as well. We know that both parties p_1 and p_2 see the same sequence of ABP messages from $\mathcal{F}_{ABP}$. Therefore, if s satisfies rules 5, 7 and 8 according to p_1, then they must pass for p_2 too.

So for each case where validation performed by p_2 could fail, we get a contradiction. Therefore, honest party p_2 must include s in its ledger, and the initial hypothesis stated at the start of the proof is false. □

Theorem 2. *Dialektos emulates UPC ideal functionality $\mathcal{F}_{UPC}$ as per def. 3.*

Theorem 3. *Dialektos is a secure UPC protocol as per def. 4.*

Proof for theorem 2 is in the appendix C. If proofs presented here for theorems 1 and 2 hold, it is trivial to see that theorem 3 holds too.

4.5 Further research

This paper only analyses a theoretical construction of UPC protocol. Further research is needed to measure performance of Dialektos implementation in reality. One important area, not covered by security analysis in this paper is resilience against DOS attacks. Invalid transactions have to be processed by honest Dialektos parties in case they conflict with any transactions that they want to submit. An attacker could spam the network with invalid conflicting transactions, increasing the work required from an honest party. In order to perform this attack, attacker needs to have state references, of states that honest party will want to spend, which often will not be the case. However, the attack might work in some contexts. Various techniques could be employed to combat this or make it more expensive, which are not within the scope of this paper.

Appendix

A UPC ideal functionality

We denote ideal functionality of a UPC scheme $\mathcal{F}_{UPC}$. It is a trusted third-party which securely executes all transactions for all parties. Code is provided is provided in figure 2, 3. Some helper functions that $\mathcal{F}_{UPC}$ uses are defined in figure 4.

B Dialektos protocol

Here we provide pseudocode for our UPC construction, called Dialektos (figures 5, 6, 7). It uses additional functionality provided by helper functions, defined in figures 8, 9, 10 and $\mathcal{F}_{DB}$ defined in figures 11, 12. $\mathcal{F}_{DB}$ is a functionality providing database-like access to set of transactions recorded in the ABP network. It simplifies the Dialektos algorithms by providing the following interface:

- GetSpends($refs$) $\rightarrow amsgs$ - takes a set of state references and returns ABP messages which claim to spend any of them.
- GetTx($msgId$) $\rightarrow amsg/\perp$ - takes ABP message id (which is a hash of some $ABPMsg$) and if message with that id is recorded by the ABP protocol, that message is returned.
- GetTxUpdate($pks, from$) $\rightarrow (f, amsgs)$ - takes a set of public keys and an integer $from$. Returns $(f, amsgs)$, where $amsgs$ is a set of ABP messages $(num, payload)$, which as $payload$ contain transactions which claim to have one of pks public keys as participant public keys (according to $encKeys$ field) and where $num \geq from$. The f returned is $f = from + n$, where n is the number of new ABP messages that have been processed in the process.
- GetContract($cref$) $\rightarrow Contract$ - takes contract reference and returns the corresponding contract if it is recorded by the ABP protocol.
- UploadContract(c) $\rightarrow \top/\perp$ - uploads contract c to the ABP.
- Update() - $\mathcal{F}_{DB}$ reads new messages in the ABP protocol and updates its state accordingly.

C Proof of theorem 2

Proof (sketch). We need to show that for any PPT adversary $\mathcal{A}$, there exists such PPT simulator $\mathcal{S}$, that for any PPT environment $\mathcal{E}$, the following would hold:

$$\{EXEC(G_r^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{A}},k)\}_{k\in\mathbb{N}} \overset{c}{\equiv} \{EXEC(G_i^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}},k)\}_{k\in\mathbb{N}}$$

```
state of $\mathcal{F}_{UPC}$
    ▷ $VAR\_NAME: TYPE = INITIAL\_VALUE$
    $pkeyParty: Map[PKey, PartyId] = \emptyset$
    $partyPKeys: Map[PartyId, Set[PKey]] = \emptyset$
    $pkeySkey: Map[PKey, SKey] = \emptyset$
    $txCount: int = 0$
    $txNums: Map[StxId, int] = \emptyset$
    $confirmedTxs: StxMap = \emptyset$
    $spentStates: Set[StateRef] = \emptyset$
    $contracts: Map[Hash, Contract] = \emptyset$
    $partyTxs: Map[PartyId, Set[StxId]] = \emptyset$
    $adversaryTxs: Set[StxId] = \emptyset$
    $corruptions: Set[PartiId]$  ▷ Set of corrupt party ids

procedure verifyStx($t: SignexTx, i: State[]$) for $\mathcal{F}_{UPC}$
    $((inRefs, outputs, cmds), sigs, *) \leftarrow t$
    verifySigs($t$)
    verifyNoDuplicates($inRefs$)
    verifyContracts($i, outputs, cmds, contracts$)
    for $ref$ in $inRefs$
        if $ref \in spentStates$ then
            throw $DOUBLE\_SPEND\_ATTEMPT$
procedure addAdversaryTx($stx: SignedTx$) for $\mathcal{F}_{UPC}$
    $id \leftarrow$ hash($stx$)
    $adversaryTxs \leftarrow adversaryTxs \cup \{id\}$
    send backdoor (RevealedTx,$txNums[id], stx$) to $\mathcal{A}$
procedure addDepsToParty($p: PartyId, tx: Tx$) for $\mathcal{F}_{UPC}$
    for $ref$ in $tx.inputRefs$
        if $p \in corruptions$ then
            addAdversaryTx($confirmedTxs[ref.stxId]$)
        $partyTxs[p] \leftarrow partyTxs[p] \cup \{ref.stxId\}$
        addDepsToParty($p, confirmedTxs[ref.stxId].tx$)
procedure addTxToParties($stx: SignedTx, ps: Set[PKey]$) for $\mathcal{F}_{UPC}$
    $id \leftarrow$ hash($stx$)
    for $p$ in $ps$
        if $p \in$ getMapKeys($pkeyParty$) then
            $party \leftarrow pkeyParty[p]$
            if $party \in corruptions$ then
                addAdversaryTx($confirmedTxs[id]$)
            $partyTxs[party] \leftarrow partyTxs[party] \cup \{id\}$
            addDepsToParty($party, stx.tx$)
procedure recordStx($stx: SignexTx, parts: Set[PKey]$) for $\mathcal{F}_{UPC}$
    for $ref$ in $stx.tx.inputRefs$
        $spentStates \leftarrow spentStates \cup \{ref\}$
    $confirmedTxs[$hash($stx$)$] \leftarrow stx$
    $txNums[$hash($stx$)$] \leftarrow txCount$
    $txCount \leftarrow txCount + 1$
    addTxToParties($stx, parts$)
```

Figure 2: $\mathcal{F}_{UPC}$: ideal functionality for UPC. Part 1

```
procedure getPartyStxs($p$) $\rightarrow$ $StxMap$ for $\mathcal{F}_{UPC}$
    return map($partyTxs[p], i \mapsto (i, confirmedTxs[i])$)
procedure partyState($p$) $\rightarrow$ $(Map[PKey, SKey], StxMap, Map[Hash, Code]$ for $\mathcal{F}_{UPC}$
    $kps \leftarrow []$
    if $p \in$ getMapKeys($partyPKeys$) then
        $kps \leftarrow$ map($partyPKeys[p], k \mapsto (k, pkeySkey[k])$)
    return ($kps$, getPartyStxs($p$), $contracts$)
procedure respond($name, p, args, m$) for $\mathcal{F}_{UPC}$
    if $p \in corruptions$ then
        send backdoor ($name$,$p$,$args$,$m$, partyState($p$)) to $\mathcal{A}$
    send output ($m$) to $p$

on input (NewKeyPair, ) to $\mathcal{F}_{UPC}$ from $p$
    $(pk, sk) \leftarrow$ AEnc.Kg()
    $pkeyParty[pk] \leftarrow p$
    $partyPKeys[p] \leftarrow partyPKeys[p] \,||\, pk$
    $pkeySKey[pk] \leftarrow sk$
    respond(NewKeyPair, $p$, (), $pk$)
on input (UploadContract, $c$) to $\mathcal{F}_{UPC}$ from $p$
    $ret \leftarrow \top$
    try
        ▷ Compiles contract and adds it to $contracts$
        ▷ Throws error if comppilation errors happen
        $code \leftarrow$ addContract($c, contracts$)
    catch $err$
        $ret \leftarrow \bot$
    send backdoor (NewContract,$c$,$code$) to $\mathcal{A}$
    respond(UploadContract, $p$, ($c$), $ret$)
on input (GetStxs, ) to $\mathcal{F}_{UPC}$ from $p$
    $ret \leftarrow$ getPartyStxs($p$)
    respond(GetStxs, $p$, (), $ret$)
on input (SubmitTx, $stx$: $SignedTx$) to $\mathcal{F}_{UPC}$ from $p$
    try
        $((inRefs, outputs, cmds), sigs, *) \leftarrow stx$
        for $ref$ in $inRefs$
            $(stxid, *) \leftarrow ref$
            if $stxid \notin partyTxs[p]$ then
                throw $ERR\_NO\_ACCESS\_TO\_TX$
        $inputs \leftarrow$ getStates($inRefs, confirmedTxs$)
        $pks \leftarrow partyPKeys[p]$
        if $\neg$statesSpendable($inputs, pks$) then
            throw $ERR\_NOT\_PARTICIPANT$
        verifyStx($stx, inputs$)
        $allPart \leftarrow$ txPart($inputs, outputs$)
        recordStx($stx, allPart$)
        $s \leftarrow$ size($stx$)
        send backdoor (NewTx,$inRefs$,$allPart$,$s$,$p$) to $\mathcal{A}$
    catch $err$
        return $\bot$
    return $\top$
```

Figure 3: $\mathcal{F}_{UPC}$: ideal functionality for UPC. Part 2

```
function getMapKeys($m$: $Map[T, *]$) $\rightarrow$ $Set[T]$
    return map($m, (k, v) \mapsto k$)
function getStates($rs$: $StateRef[]$, $ts$: $StxMap$) $\rightarrow$ $State[]$
    return map($rs, r \mapsto ts[r.stxId].outputs[r.index]$)
procedure verifySigs($stx$: $SignedTx$)
    $(tx, sigs, *) \leftarrow stx$
    for $cmd$ in $tx.cmds$
        for $signer$ in $cmd.signers$
            if $signer \notin$ getMapKeys($sigs$) then
                throw $SIG\_NOT\_PRESENT$
            if $\neg$Sig.Valid($sigs[signer], signer, tx$) then
                throw $SIG\_INVALID$
procedure verifyNoDuplicates($refs$: $StateRef[]$)
    $rset \leftarrow \emptyset$
    for $r$ in $refs$
        if $r \in rset$ then
            throw $DUPL\_INPUT$
        else
            $rset \leftarrow rset \cup \{r\}$
function getContractRefs($s$: $State[]$) $\rightarrow$ $Set[Hash]$
    return map($s, st \mapsto st.contractRef$)
procedure verifyContracts($i$: $State[]$, $o$: $State[]$, $cmds$: $Command[]$, $cs$: $Map[Hash, Contract]$)
    $crefs \leftarrow$ getContractRefs($i$) $\cup$ getContractRefs($o$)
    for $cref$ in $cRefs$
        $c \leftarrow cs[c]$        ▷ Throws if contract does not exist
        if $\neg$c.verify($i, o, cmds$) then
            throw $INVALID\_STATE\_TRANSITION$
procedure addContract($c, cs$ : $Map[Hash, Contract]$) $\rightarrow$ $Contract$
    $code \leftarrow$ compileContract($c$)
    $cid \leftarrow$ hash($c$)
    if $cid \in$ getMapKeys($cs$) then
        throw $CONTRACT\_ALREADY\_EXISTS$
    $cs[cid] \leftarrow code$
    return $code$
function getStParticipants($sts$: $State[]$) $\rightarrow$ $Set[PKey]$
    $pks \leftarrow \emptyset$
    for $st$ in $sts$
        $pks \leftarrow pks \cup st.participants$
    return $pks$
function txPart($i$: $State[]$, $o$: $State[]$) $\rightarrow$ $Set[PKey]$
    return getStParticipants($i$) $\cup$ getStParticipants($o$)
function statesSpendable($st$: $State[]$, $pk$: $PKey[]$) $\rightarrow$ $\top/\bot$
    for $s$ in $st$
        if $st.participants \cap pk = \emptyset$ then
            return $\bot$
    return $\top$
```

Figure 4: Helper functions. Part 1

```
**state of** $\mathcal{P}_{DLT}$
    $\triangleright$ $VAR\_NAME: TYPE = INITIAL\_VALUE$
    $id: PartyId,$                                    $\triangleright$ Id of this node
    $keypairs: Map[PKey, SKey] = \emptyset,$
    $txs: StxMap = \emptyset,$
    $txKeys: Map[MsgId, Key] = \emptyset,$
    $stxIds: Map[MsgId, StxId] = \emptyset$
    $msgIds: Map[StxId, MsgId] = \emptyset$
    $txCount: int = 0$
    $db: \mathcal{F}_{DB}$                            $\triangleright$ Trusted syntax node

**types**
    $Dep = (msgId: MsgId, txMsg: TxMsg, key: Key)$
    $DepMap = Map[StxId, Dep]$
```

Figure 5: $\mathcal{P}_{DLT}$: Dialektos UPC construction. Part 1

where $G_r^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{A}}$ and $G_i^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$ work as described in section 3.3.2.

For this purpose, we will need alternative versions of Dialektos protocol, which we denote by $\mathcal{P}'(O_{kg}, O_e)$. They behave identically to $\mathcal{P}_{DLT}$, except whenever they need to use AEnc.Kg they use O_{kg}.Kg instead and whenever they need to use HEnc.E, they use O_e.E instead.

O_e is a stateful machine, which contains a list of pre-generated outputs to return for each call to O_e.E. This machine is created by a function genEncOracle$(bm) \rightarrow O_e$, which constructs a list of outputs for the generated O_e to return by taking a sequence of backdoor messages from $\mathcal{F}_{UPC}$, parsing RevealedTx messages from them, so that nth call to O_e.E returns nth encrypted transaction as revealed by RevealedTx. For transactions, which do not have a corresponding RevealedTx message among the backdoor messages, a value of $s \in \mathbb{N}$ zeroes is encrypted, where s is the size of that transaction as revealed by corresponding NewTx backdoor message from $\mathcal{F}_{UPC}$. Fig. 14 displays this functionality in pseudocode.

O_{kg} is a similar stateful machine, which contains a list of pre-generated outputs to return for each call to O_{kg}.Kg. This list of outputs is generated by a function that creates O_{kg}: genKgOracle$(bm) \rightarrow O_{kg}$. It does so by parsing sequence of backdoor messages of type NewKeyPair generated by $\mathcal{F}_{UPC}$, and making generated O_{kg} return the public keys specified in those messages.

The algorithm for simulator $\mathcal{S}(bm, O_{kg}, O_e)$ is in figure 13.

After these preparations, we proceed by constructing a sequence of games and applying a hybrid argument to them.

Game 1. We construct $G_1^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$ by making the following changes to $G_r^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{A}}$:

- $\mathcal{E}$ interacts with $\mathcal{F}_{UPC}$ instead of $\mathcal{P}_{DLT}$ protocol parties.

```
**procedure** recordStx($t: TxMsg, k: Key, id: MsgId$) **for** $\mathcal{P}_{DLT}$
    $stxid \leftarrow$ hash$(t.stx)$
    **if** $stxid \notin$ getMapKeys$(txs)$ **then**
        $txs[stxid] \leftarrow t.stx$
        $stxIds[id] \leftarrow stxid$
        $msgIds[stxid] \leftarrow id$
        $txKeys[id] \leftarrow k$
**procedure** declareConflicts($t: TxMsg$) **for** $\mathcal{P}_{DLT}$
    $inRefs \leftarrow t.stx.inputRefs$
    **send input** (GetSpends,$inRefs$) **to** $db$ **and**
        **receive output** $msgs$
    **for** $m$ **in** $msgs$
        $oldId \leftarrow$ hash$(m)$
        $cInRefs \leftarrow inRefs \cap m.payload.inputRefs$
        $cInputs \leftarrow$ getStates$(cInRefs, txs)$
        $cPart \leftarrow$ getStParticipants$(cInputs)$
        $ePKeys \leftarrow$ getMapKeys$(m.payload.encKeys)$
        **if** $cPart \subseteq ePKeys$ **then**
            $myKeys \leftarrow$ getMapKeys$(keypairs) \cap ePKeys$
            $ekey \leftarrow ePKeys[myKeys[0]]$
            $key \leftarrow$ AEnc.D$(keypairs[myKeys[0]], ekey)$
            $t.depKeys[oldId] \leftarrow key$
**procedure** constructTx($s: SignedTx$) $\rightarrow$ $(ABPMsg, TxMsg)$ **for** $\mathcal{P}_{DLT}$
    $txMsg: TxMsg \leftarrow (s, \emptyset)$
    **for** $inRef$ **in** $s.tx.inputRefs$
        **if** $inRef.stxId \notin$ getMapKeys$(txs)$ **then**
            **throw** $ERR\_NO\_ACCESS\_TO\_TX$
        $depId \leftarrow msgIds[inRef.stxId]$
        $txMsg.depKeys[depId] \leftarrow txKeys[depId]$
    $inputs \leftarrow$ getStates$(s.tx.inputRefs, txs)$
    $pk \leftarrow$ getMapKeys$(keypairs)$
    **if** $\neg$statesSpendable$(inputs, pk)$ **then**
        **throw** $ERR\_NOT\_PARTICIPANT$
    declareConflicts$(txMsg)$
    $allp \leftarrow$ txPart$(inputs, s.tx.outputs)$
    $(ek, etx) \leftarrow$ HEnc.E$(allp, txMsg)$
    $tm \leftarrow ABPTxMsg(ek, s.tx.inputRefs, etx)$
    **return** $(ABPMsg(\infty, tm), txMsg)$
```

Figure 6: $\mathcal{P}_{DLT}$: Dialektos UPC construction. Part 2

```
procedure updateState() for $\mathcal{P}_{DLT}$
    send input (Update,) to $db$
    $p \leftarrow$ getMapKeys($keypairs$)
    send input (GetTxUpdate,$p$,$txCount$) to $db$ and
        receive output ($n$,$msgs$)
    $txCount \leftarrow n$
    ▷ Iterates from older to newer
    for $msg$ in $msgs$
        $depMap \leftarrow \emptyset$
        try
            $depMap \leftarrow$ verifyABPMsg($msg$,$keypairs$)
        catch $err$
            ▷ Don't record invalid txs
            continue            ▷ Skip to next loop iteration
        for ($stxId$,($msgId$,$txMsg$,$txKey$)) in $depMap$
            recordStx($txMsg$,$txKey$,$msgId$)
on input (NewKeyPair, ) to $\mathcal{P}_{DLT}$ from $p$
    updateState()
    ($pk$,$sk$) $\leftarrow$ AEnc.Kg()
    $keypairs[pk] \leftarrow sk$
    return $pk$
on input (UploadContract, $contr$) to $\mathcal{P}_{DLT}$ from $p$
    updateState()
    send input (UploadContract,$contr$) to $db$ and
        receive output $resp$
    send output ($resp$) to $p$
on input (GetStxs, ) to $\mathcal{P}_{DLT}$ from $p$
    updateState()
    send output ($txs$) to $p$
on input (SubmitTx, $stx$: $SignedTx$) to $\mathcal{P}_{DLT}$ from $p$
    updateState()
    try
        ($msg$,$txm$) $\leftarrow$ constructTx($stx$)
        verifyABPMsg($msg$,$keypairs$)
    catch $err$
        send output ($err$) to $p$
        stop
    send input (Submit,$msg$) to $\mathcal{F}_{ABP}$
    send output ($\top$) to $p$
```

Figure 7: $\mathcal{P}_{DLT}$: Dialektos UPC construction. Part 3

```
procedure verConfl($m1$: $ABPMsg$,$t2$: $TxMsg$,$d$: $DepMap$)
    $inRefs \leftarrow m1.payload.inputRefs$
    $commonInRefs \leftarrow inRefs \cap t2.stx.inputRefs$
    $commonInputs \leftarrow$ getDepStates($commonInRefs$,$d$)
    $commonPart \leftarrow$ getStParticipants($commonInputs$)
    $epk \leftarrow m1.payload.encKeys$
    if $commonPart \nsubseteq$ getMapKeys($epk$) then
        return                          ▷ $m1$ invalid
    if hash($m1$) $\notin$ getMapKeys($t2.depKeys$) then
        throw $CONFLICT\_KEY\_MISSING$▷ $t2$ invalid
    $depMsgIds \leftarrow$ map($d$,($k$,($i$,$*$,$*$)) $\mapsto i$)
    if hash($m1$) $\in depMsgIds$ then
        throw $DOUBLE\_SPEND\_ATTEMPT$        ▷ $t2$ invalid
    else
        return                          ▷ $m1$ invalid
procedure verifyConflicts($n$: $int$,$t$: $TxMsg$,$vdeps$: $DepMap$) for
    $inRefs \leftarrow t.stx.inputRefs$
    send input (GetSpends,$inRefs$) to $\mathcal{F}_{DB}$ and
        receive output $msgs$
    for $m$ in $msgs$
        ▷ Skip t (m.num = n) and all txs that come after it
        if $m.num < n$ then
            verConfl($m$,$t$,$vdeps$)
procedure getContract($cref$: $Hash$) $\rightarrow$ $Code$
    send input (GetContract,$cref$) to $\mathcal{F}_{DB}$ and
        receive output $c$
    if $c = \perp$ then
        throw $CONTRACT\_DOES\_NOT\_EXIST$
    else
        return $c$
function getDepStates($rs$: $StateRef[]$,$d$: $DepMap$) $\rightarrow$ $State[]$
    return map($rs$,$r \mapsto d[r.stxId].txMsg.stx.tx.outputs[r.index]$)
procedure verifyEncKeys($ek$: $Map[PKey,String]$, $i$: $State[]$,$o$: $State[]$,$k$: $Key$)
    $parts \leftarrow$ txPart($i$,$o$)
    for $pkey$ in $parts$
        if $pkey \notin$ getMapKeys($ek$) then
            throw $KEY\_FOR\_PARTICIPANT\_MISSING$
    for ($pk$,$c$) in $ek$
        $eTxKey \leftarrow$ AEnc.E($pk$,$k$)
        if $eTxKey \neq c$ then
            throw $KEY\_ENCRYPTION\_MISSMATCH$
```

Figure 8: Helper functions. Part 2

```
procedure verifyInputRefs($m$: $ABPTxMsg, t$: $Tx$)
    ▷ Sequences should be equal
    if $m.inputRefs \neq t.inputRefs$ then
        throw $PUBLIC\_INPUT\_REFS\_INVALID$
procedure verifyTxMsg($t$: $TxMsg, vdeps$: $DepMap$)
    for $inRef$ in $t.stx.tx.inputRefs$
        if $inRef.stxId \notin$ getMapKeys($vdeps$) then
            throw $DEP\_NOT\_VERIFIED$
procedure verifyStx($m$: $ABPMsg, t$: $TxMsg, k$: $Key$,
$d$: $DepMap$)
    $(((inRefs, outputs, cmds), *, *), *) \leftarrow t$
    verifyTxMsg($t, d$)
    verifySigs($t.stx$)
    verifyNoDuplicates($inRefs$)
    verifyInputRefs($m.payload, t$)
    $inputs \leftarrow$ getDepStates($inRefs, d$)
    verifyEncKeys($m.payload.encKeys, inputs, outputs, k$)
    verifyContracts($inputs, outputs, cmds$, getContract)
function joinMaps($m1$: $Map[K,V], m2$: $Map[K,V]$) $\rightarrow$
$Map[K,V]$
    for $(k, v)$ in $m2$
        if $k \notin$ getMapKeys($m1$) then
            $m1[k] \leftarrow v$
    return $m1$
procedure verifyTxTree($m$: $ABPMsg, t$: $TxMsg, k$: $Key$)
$\rightarrow DepMap$
    $depMap$: $DepMap \leftarrow \emptyset$
    for $(msgId, key)$ in $t.depKeys$
        send input (GetTx,$msgId$) to $\mathcal{F}_{DB}$ and
            receive output $msg$
        if $msg = \perp$ then
            throw $DEP\_DOES\_NOT\_EXIST$
        $(txNum, depMsg) \leftarrow msg$
        ▷ Deps of $t$ have to come before it
        if $txNum \geq m.num$ then
            throw $NEWER\_TX\_IN\_DEPS$
        try
            ▷ Throws if decrypted value is not TxMsg
            $d$: $TxMsg \leftarrow$ SEnc.D($key, depMsg.etx$)
            $dd \leftarrow$ verifyTxTree($msg, d, key$)
            $depMap \leftarrow$ joinMaps($depMap, dd$)
        catch $err$
            ▷ Dep might not be required, so ignore
            continue            ▷ Skip to next loop iteration
    verifyStx($m, t, k, depMap$)
    verifyConflicts($m.num, t, depMap$)
    $depMap[$hash$(depTx.stx)] \leftarrow ($hash$(m), t, k)$
```

Figure 9: Helper functions. Part 3

```
procedure unpackTx($m$: $ABPMsg, k$: $Map[PKey, SKey]$)
$\rightarrow (Key, TxMsg)$
    $(*, (eks, *, etx)) \leftarrow m$
    $pks \leftarrow$ getMapKeys($eks$) $\cap$ getMapKeys($k$)
    if $pks = \emptyset$ then
        throw $NO\_TX\_KEY$
    $(k, txMsg) \leftarrow$ HEnc.D$(k[pks[0]], (eks[pks[0]], etx))$
    if $txMsg$ is $TxMsg$ then
        return $(k, txMsg)$
    else
        throw $WRONG\_TYPE$
procedure verifyABPMsg($m$: $ABPMsg$,
$keypairs$: $Map[PKey, SKey]$) $\rightarrow DepMap$
    $(txMsg, txKey) \leftarrow$ unpackTx($m, keypairs$)
    return verifyTxTree($msg, txMsg, txKey$)
```

Figure 10: Helper functions. Part 4

```
state of $\mathcal{F}_{DB}$
    ▷ $VAR\_NAME$: $TYPE = INITIAL\_VALUE$
    $txs$: $Map[MsgId, ABPMsg] = \emptyset$
    $txOrdered$: $MsgId[] = []$
    $contracts$: $Map[Hash, Code] = \emptyset$,
    $spends$: $Map[StateRef, MsgId[]] = \emptyset$
    $msgCount$: $int = 0$

procedure recordTx($t$: $ABPMsg$) for $\mathcal{F}_{DB}$
    $txid \leftarrow$ hash($t$)
    $txs[txid] \leftarrow t$
    $txOrdered \leftarrow txOrdered \,||\, txid$
    $(*, refs, *) \leftarrow t.payload$
    for $r$ in $refs$
        $spends[r] \leftarrow spends[r] \,||\, \{txid\}$
```

Figure 11: $\mathcal{F}_{DB}$: protocol for syntax node. Part 1.

```
16: procedure update($msgs$: $ABPMsg[]$) for $\mathcal{F}_{DB}$
17:     while $msgCount < \mathsf{len}(msgs)$ do
18:         $(num, m) \leftarrow msgs[msgCount]$
19:         if $m$ is $ABPTxMsg$ then
20:             $\mathsf{recordTx}((num, m))$
21:         else $m$ is $ABPContractMsg$
22:             try
23:                 $code \leftarrow \mathsf{addContract}(m, contracts)$
24:             catch $err$
25:                 ▷ Ignore invalid contracts
26:         $msgCount \leftarrow msgCount + 1$
27: on input (UploadContract, $c$: $Contract$) to $\mathcal{F}_{DB}$ from $p$
28:     try
29:         $cs \leftarrow contracts$                ▷ Copy contract to cs
30:         $code \leftarrow \mathsf{addContract}(c, cs)$
31:     catch $err$
32:         send output $((\bot))$ to $p$
33:         stop
34:     send input (Submit,$ABPMsg(\infty, c)$) to $\mathcal{F}_{ABP}$
35:     send output $(\top)$ to $p$
36: on input (GetSpends, $refs$: $StateRef[]$) to $\mathcal{F}_{DB}$ from $p$
37:     $msgs \leftarrow []$
38:     for $r$ in $refs$
39:         if $r \in \mathsf{getMapKeys}(spends)$ then
40:             for $msgid$ in $spends[r]$
41:                 $msgs \leftarrow msgs \,\|\, txs[msgid]$
42:     send output $(msgs)$ to $p$
43: on input (GetTx, $msgId$: $MsgId$) to $\mathcal{F}_{DB}$ from $p$
44:     if $msgId \in \mathsf{getMapKeys}(txs)$ then
45:         send output $(txs[msgId])$ to $p$
46:     else
47:         send output $(\bot)$ to $p$
48: on input (GetContract, $cref$: $Hash$) to $\mathcal{F}_{DB}$ from $p$
49:     if $cref \in \mathsf{getMapKeys}(contracts)$ then
50:         send output $(contracts[cref])$ to $p$
51:     else
52:         send output $(\bot)$ to $p$
53: on input (GetTxUpdate, $pks$: $Set[PKey], fromTx$: $int$)
    to $\mathcal{F}_{DB}$ from $p$
54:     $msgs \leftarrow []$
55:     while $fromTx < \mathsf{len}(txOrdered)$ do
56:         $m \leftarrow txs[txOrdered[fromTx]]$
57:         $fromTx \leftarrow fromTx + 1$
58:         $(*, (eks, *, *)) \leftarrow m$
59:         if $pks \cap \mathsf{getMapKeys}(eks) \neq \emptyset$ then
60:             $msgs \leftarrow msgs \,\|\, m$
61:     send output $(fromTx, msgs)$ to $p$
62: on input (Update, ) to $\mathcal{F}_{DB}$ from $p$
63:     send input (Read,) to $\mathcal{F}_{ABP}$ and
64:         receive output $msgs$
65:     $\mathsf{update}(msgs)$
```

Figure 12: $\mathcal{F}_{DB}$: protocol for syntax node. Part 2.

- At the end of the execution, instead of returning state of $\mathcal{A}$, value given by $\mathcal{S}(bm, \mathsf{genKgOracle}(bm), o)$ is returned to the environment, where bm are backdoor messages generated by $\mathcal{F}_{UPC}$. Value o here is generated by $o \leftarrow \mathsf{genEncOracle}(bm')$, where bm' is bm plus additional RevealedTx messages added by challenger for each transaction that was submitted by $\mathcal{E}$, but which were not leaked by $\mathcal{F}_{UPC}$.

Observe that in $\mathcal{F}_{UPC}$.UploadContract() the same checks are performed on the input as in $\mathcal{P}_{DLT}$.UploadContract(). Meaning these functions always return the same value $\top/\bot$ for the same input. Since same inputs from the environment are passed to $\mathcal{F}_{UPC}$ as to $\mathcal{P}_{DLT}$, we can conclude that these functions behave identically from the perspective of environemnt.

The same holds for $\mathcal{F}_{UPC}$.SubmitTx() and $\mathcal{P}_{DLT}$.SubmitTx(). Although, output of $\mathcal{F}_{UPC}$.SubmitTx() additionally depends on the transactions already recorded, transactions for parties are recorded based on the same conditions as in $\mathcal{P}_{DLT}$. This means that calls to SubmitTx and GetStxs work identically in both worlds from the perspective of environent.

As for NewKeyPair, in $\mathcal{F}_{UPC}$ these keypairs are generated using the same key generation method, so the output to the environment is indistinguishable as well.

So $\mathcal{E}$ will not be able to distinguish $G_r^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{A}}$ from $G_1^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$, based on outputs alone.

Next we need to consider the view that $\mathcal{S}$ generates to the adversary. This view consists of ABP messages submitted to $\mathcal{F}_{ABP}$ and backdoor messages generated by corrupt parties of $\mathcal{P}_{DLT}$. The way simulator works is that it runs corrupt parties of protocol $\mathcal{P}'(O_{kg}, O_e)$ which differ from $\mathcal{P}_{DLT}$ parties only in that it uses O_{kg}.Kg and O_e.E instead of AEnc.Kg and HEnc.E. But from the way O_{kg} and O_e are generated, we see that values returned by these oracle are actually generated using the same methods and inputs as in $\mathcal{P}_{DLT}$. For O_{kg} they are generated using the same method in $\mathcal{F}_{UPC}$, and for O_e they are generated by using the same HEnc to encrypt the same transactions. Note that O_e in this game was generated using full information about all transactions that were submitted by the environment, so every transaction submitted by the environment through a corrupt party is encrypted the same way as in $\mathcal{P}_{DLT}$.

This is the case for corrupt parties. For parties which are not corrupt simulator only needs to generate ABP messages (no need for party state). It does so through encryption oracle, which as already mentioned, encrypts all transactions using the same methods as in the real protocol run.

Therefore, $\mathcal{P}'(O_{kg}, O_e)$ protocol run simulated by $\mathcal{S}$ proceeds identically to $\mathcal{P}_{DLT}$ protocol run with the same inputs (just places where some values are computed is changed) and the view generated by $\mathcal{S}$ for $\mathcal{A}$ is indistinguishable from the real one.

$EXEC(G_1^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}, k)$ is indistinguishable from $EXEC(G_r^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}, k)$.

Game 2. $G_2^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$ works the same way as $G_1^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$ except the encryption oracle is different. In this game it is generated by $o' \leftarrow \mathsf{genEncOracle}(bm)$, where bm is a sequence of backdoor messages from $\mathcal{F}_{UPC}$, unchanged. This change to encryption oracle means that some transactions submitted by simulated protocol parties are fake (meaning they are zeroes encrypted instead of real encryptions). Fortunatelly, this is only the case for transactions which are not leaked by $\mathcal{F}_{UPC}$, and these are transactions, which corrupted parties should not be able to decrypt anyway.

In order to show that $G_2^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$ and $G_1^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$ are indistinguishable, we use the fact that $o.\mathsf{E}()$ produces probability distribution indistinguishable from $o'.\mathsf{E}()$. To see this, note that HEnc, that both of these encryption oracles use, is secure against chosen-plaintext attacks. So even if distinguisher knows that one of the oracles encrypts zeroes instead of some transactions, that does not allow it to distinguish output of one oracle from another.

Then to see that $G_2^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$ is indistinguishable from $G_1^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$ note that we can construct an algorithm which takes one of $\{o, o'\}$ and without any input regarding which one it took, runs $G_1^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$ if it was o and $G_2^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$ if it was o'. All this algorithm needs to do is work like challenger in $G_2^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$ or $G_1^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$, except use the encryption oracle it receives as input, instead of generating it itself.

Next, observe that $G_2^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$ is indistinguishable from $G_i^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$. The difference between $G_2^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$ and $G_i^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$ is that in $G_i^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$ no encryption or key generation oracle is passed to the simulator. But note that in $G_2^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$ these oracles are generated by passing only the backdoor messages generated by $\mathcal{F}_{UPC}$. This is the same information that is passed to the simulator as well in $G_i^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$. So we can simply modify the simulator used in $G_i^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$, so that it runs $\mathsf{genEncOracle}(bm)$ and $\mathsf{genKgOracle}(bm)$ itself.

Thus by hybrid argument, we know that $G_r^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{A}}$ is indistinguishable from $G_i^{\mathcal{E},\mathcal{P}_{DLT},\mathcal{S}}$, from the point of view of $\mathcal{E}$, which is what we needed to prove.

□

```
types
    $Msg = (name: String, args)$
function $\mathcal{S}(bm: Msg, kg: O_{kg}, enc: O_e) \rightarrow \mathcal{A}$
    $cp: Map[PartyId, \mathcal{P}'(O_{kg}, O_e)] \leftarrow []$
    for $pid$ in $corruptions$
        $cp[pid] \leftarrow \mathcal{P}'(kg, enc)$
    $abp \leftarrow \mathcal{F}_{ABP}()$
    $adv \leftarrow \mathcal{A}()$
    for $(m, index)$ in $bm$
        if $m.name = \mathsf{NewTx}$ then
            if $bm[index+1].name \neq \mathsf{SubmitTx}$ then
                ▷ These oracles ignore inputs
                $(ek, etx) \leftarrow enc.\mathsf{E}(*, *)$
                $(*, (inRefs, *, *, *)) \leftarrow m$
                $m \leftarrow ABPMsg(\infty, ABPTxMsg(ek, inRefs, etx))$
                $advNotif \leftarrow abp.\mathsf{Submit}(m)$
                send input (Submit,m) to abp and
                    receive output $b$
                send $b$ to adv
        else if $m.name = \mathsf{SubmitTx}$ then
            $(*, (pid, args, *, *)) \leftarrow m$
            ▷ This call sends a message to ABP and adversary. $b$ holds these messages
            send input (SubmitTx,args) to cp[pid] and
                receive output $b$
            send input (Submit,b[0]) to abp and
                receive output $b'$
            send $b'$ to $\mathcal{A}$
            send $b[1]$ to $\mathcal{A}$
        else if $m.name = \mathsf{GetStxs}$ then
            send input (GetStxs,) to cp[pid] and
                receive output $b$
            send $b$ to $\mathcal{A}$
        else if $m.name = \mathsf{NewKeyPair}$ then
            $(*, (pid, *, *, *)) \leftarrow m$
            send input (NewKeyPair,) to cp[pid] and
                receive output $b$
            send $b$ to $\mathcal{A}$
        else if $m.name = \mathsf{UploadContract}$ then
            $(*, (pid, a, *, *)) \leftarrow m$
            $r \leftarrow cp[pid]$
            send input (UploadContract,a) to r and
                receive output $b$
            send $b$ to $\mathcal{A}$
        else if $m.name = \mathsf{NewContract}$ then
            $next \leftarrow bm[index+1].name$
            if $next \neq \mathsf{UploadContract}$ then
                $(*, (pid, args, *, *)) \leftarrow m$
                $mc \leftarrow ABPMsg(\infty, args)$
                $advNotif \leftarrow abp.\mathsf{Submit}(mc)$
                send $advNotif$ to $\mathcal{A}$

```

Figure 13: $\mathcal{S}$: simulator.

```
state of $O_e$
    $\triangleright$ $VAR\_NAME: TYPE = INITIAL\_VALUE$
    $db: ((PKey, String)[], String)[] = []$
    $callNum: int \leftarrow 0$

procedure $\mathsf{E}(pks, m)$ for $O_e$
    $v \leftarrow db[callNum]$
    $callNum \leftarrow callNum + 1$
    return $v$
function $\mathsf{genEncOracle}(bm: Msg) \rightarrow O_e$
    $o \leftarrow O_e()$
    $(opk, osk) \leftarrow \mathsf{AEnc.Kg}()$
    $rtxs: (SignedTx, PKey[])[] \leftarrow []$
    $txs: StxMap = \emptyset,$
    $txKeys: Map[MsgId, Key] = \emptyset,$
    $msgIds: Map[StxId, MsgId] = \emptyset$
    $count \leftarrow 0$
    for $m$ in $bm$
        if $m.name = \mathsf{NewTx}$ then
            $(*(*, parts, size, *)) \leftarrow m$
            $rtxs[count] = (0 * size, parts)$
            $count \leftarrow count + 1$
    for $(m, index)$ in $bm$
        if $m.name = \mathsf{RevealedTx}$ then
            $(*, (num, stx)) \leftarrow m$
            $rtxs[num] \leftarrow (stx, *)$
    for $((stx, parts), index)$ in $rtxs$
        if $stx \neq 0$ then
            $stxid \leftarrow \mathsf{hash}(stx)$
            $\triangleright$ Construct txMsg
            $t \leftarrow TxMsg$
            for $inRef$ in $stx.tx.inputRefs$
                $depId \leftarrow msgIds[inRef.stxId]$
                $t.depKeys[depId] \leftarrow txKeys[depId]$
            $i \leftarrow \mathsf{getStates}(stx.tx.inputRefs, txs)$
            $allp \leftarrow \mathsf{txPart}(i, stx.tx.outputs) \cup \{opk\}$
            $(ek, etx) \leftarrow \mathsf{HEnc.E}(allp, t)$
            $(k, *) \leftarrow \mathsf{HEnc.D}(osk, (ek[opk], etx))$
            $ek \leftarrow ek \setminus (opk, *)$
            $a \leftarrow ABPTxMsg(ek, s.tx.inputRefs, etx)$
            $amsg \leftarrow ABPMsg(index, a)$
            $mid \leftarrow \mathsf{hash}(amsg)$
            $\triangleright$ Record
            $txs[stxid] \leftarrow stx$
            $msgIds[stxid] \leftarrow mid$
            $txKeys[mid] \leftarrow k$
            $o.db[index] \leftarrow (ek, etx)$
        else
            $o.db[index] \leftarrow \mathsf{HEnc.E}(parts, stx)$
    return $o$
```

Figure 14: Encryption oracle

References

[1] Counterparty homepage. https://counterparty.io/.

[2] Enigma homepage. https://www.enigma.co/about/.

[3] Monero homepage. https://www.getmonero.org/.

[4] Muneeb Ali, Jude Nelson, Ryan Shea, and Michael J Freedman. Blockstack: A global naming and storage system secured by blockchains. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, pages 181–194, 2016.

[5] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[6] Arati Baliga, I Subhod, Pandurang Kamat, and Siddhartha Chatterjee. Performance evaluation of the quorum blockchain platform. *arXiv preprint arXiv:1809.03421*, 2018.

[7] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. Deterministic and efficiently searchable encryption. In *Annual International Cryptology Conference*, pages 535–552. Springer, 2007.

[8] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 947–964. IEEE, 2020.

[9] Zvika Brakerski and Gil Segev. Better security for deterministic public-key encryption: The auxiliary-input setting. In *Annual Cryptology Conference*, pages 543–560. Springer, 2011.

[10] Richard Gendal Brown. The corda platform: An introduction. *Retrieved*, 27:2018, 2018.

[11] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In *International Conference on Financial Cryptography and Data Security*, pages 423–443. Springer, 2020.

[12] Manuel MT Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The extended utxo model. In *International Conference on Financial Cryptography and Data Security*, pages 525–539. Springer, 2020.

[13] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[14] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200. IEEE, 2019.

[15] Jacob Eberhardt and Stefan Tai. Zokrates-scalable privacy-preserving off-chain computations. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1084–1091. IEEE, 2018.

[16] Paolo Guida. Privacy is the biggest challenge preventing defi lift-off, August 2021. Accessed: 2021-11-22.

[17] Mike Hearn. Corda: A distributed ledger. *Corda Technical White Paper*, 2019, 2019.

[18] Javier Herranz, Dennis Hofheinz, and Eike Kiltz. Kem/dem: Necessary and sufficient conditions for secure hybrid encryption. *Manuscript in preparation*, 2006.

[19] Dennis Hofheinz and Eike Kiltz. Secure hybrid encryption from weakened key encapsulation. In *Annual International Cryptology Conference*, pages 553–571. Springer, 2007.

[20] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification. *GitHub: San Francisco, CA, USA*, 2016.

[21] Josh Howarth. Top 5 defi trends for 2021-2023, November 2021. Accessed: 2021-11-22.

[22] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. Arbitrum: Scalable, private smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1353–1370, 2018.

[23] Gabriel Kaptchuk, Ian Miers, and Matthew Green. Giving state to the stateless: Augmenting trustworthy computation with ledgers. *Cryptology ePrint Archive*, 2017.

[24] Thomas Kerber, Aggelos Kiayias, and Markulf Kohlweiss. Kachina–foundations of private smart contracts. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–16. IEEE, 2021.

[25] Tommy Koens, Scott King, Matthijs van den Bos, Cees van Wijk, and Aleksei Koren. Solutions for the corda security and privacy trade-off: Having your cake and eating it.

[26] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy (SP)*, pages 839–858. IEEE, 2016.

[27] Yehuda Lindell. How to simulate it–a tutorial on the simulation proof technique. *Tutorials on the Foundations of Cryptography*, pages 277–346, 2017.

[28] Jude Nelson, Muneeb Ali, Ryan Shea, and Michael J Freedman. Extending existing blockchains with virtualchain. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, 2016.

[29] Shen Noether, Adam Mackenzie, et al. Ring confidential transactions. *Ledger*, 1:1–18, 2016.

[30] Alexandre Miranda Pinto. An introduction to the use of zk-snarks in blockchains. In *Mathematical Research for Blockchain Economy*, pages 233–249. Springer, 2020.

[31] Meni Rosenfeld. Overview of colored coins. *White paper, bitcoil. co. il*, 41:94, 2012.

[32] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.

[33] Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin Vechev. zkay: Specifying and enforcing data privacy in smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1759–1776, 2019.

[34] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 991–1008, 2018.

[35] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Breaking virtual memory protection and the sgx ecosystem with foreshadow. *Ieee Micro*, 39(3):66–74, 2019.

[36] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[37] Alexei Zamyatin, Nicholas Stifter, Aljosha Judmayer, Philipp Schindler, Edgar Weippl, and William J Knottenbelt. A wild velvet fork appears! inclusive blockchain protocol changes in practice. In *International Conference on Financial Cryptography and Data Security*, pages 31–42. Springer, 2018.