KACHINA – Foundations of Private Smart Contracts

Thomas Kerber, Aggelos Kiayias, and Markulf Kohlweiss

The University of Edinburgh and IOHK
papers@tkerber.org akiayias@ed.ac.uk mkohlwei@ed.ac.uk

"v0.9" prerelease

Abstract. Smart contracts present a uniform approach for deploying distributed computation and have become a popular means for developing security critical applications. A major barrier to adoption for many applications is the inherently public nature of existing systems, such as Ethereum. Several systems satisfying various definitions of privacy and requiring various trust assumptions have been proposed; however, none achieved the universality and uniformity that Ethereum achieved for non-private contracts: One framework sufficing to construct most contracts. We provide a unified security model for private smart contracts which is based on the Universal Composition (UC) model and propose a novel protocol, KACHINA, for deploying privacy-preserving smart contracts, which encompasses previous systems. We demonstrate the practicality of KACHINA by using it to construct a contract that implements privacy-preserving payments, along the lines of Zerocash, which is provably secure in the UC setting and facilitates concurrency.

1 Introduction

Distributed ledgers put forth a new paradigm for deploying Internet services that transcends the classical client-server model. In this new model, it is no longer the responsibility of a single organization or a small consortium of organizations to provide the platform for deploying the relevant business logic. Instead, service deployments can take advantage of decentralised "trustless" computation to improve their transparency and security as well as reduce the need for trusted third parties and intermediaries.

Bitcoin [25], the first successfully deployed distributed ledger protocol, does not lend itself easily to the implementation of arbitrary protocol logic that can support the above paradigm. This led to many adaptations of the basic protocol for specific applications, such as NameCoin [18], a distributed domain registration protocol, or Bitmessage [29], a ledger-based communications protocol, and many others. An obvious problem with this approach is that, even though the Bitcoin source code can be copied arbitrary times, the Bitcoin community of software developers and miners cannot, and hence such systems are typically not sustainable. Smart contracts, originally posited as a form of reactive

computation [27], were popularized by Ethereum [31], solving these problems by providing a uniform and standardized approach for deploying decentralized computation over the same back-end infrastructure.

Smart contract systems rely on a form of *state-machine replication* [26] – all nodes involved in maintaining the smart contract keep a local copy of its state, and advance this copy with a sequence of requests. This sequence of requests needs to match for each node in the system – thus the need for consensus over which requests are made, and their order. In practice, this is achieved through distributed ledgers.

A seemingly inherent limitation of the decentralised computation paradigm is the fact that protocol logic deployed as a smart contract has to be completely non-private. This, naturally, is a major drawback for many of the applications that can potentially take advantage of smart contracts. Promising cryptographic techniques for lifting this limitation are zero-knowledge proofs [17], and secure-computation [16,10]. Motivated by such cryptographic techniques, systems satisfying various definitions of privacy – and requiring various trust assumptions – have been proposed [4,22,32,19], as we detail in Subsection 1.2. Their reliance on trust assumptions nevertheless, in one way or another, fundamentally limits the decentralization features that are afforded by their non-private counterparts. For instance, a common restriction of such systems is to assume a small, fixed set of participants at the core of the system. This fundamentally clashes with the basic principles of a decentralised platform like Bitcoin or Ethereum (collectively classified as *Nakamoto consensus*). In these systems, the set of parties maintaining the system can be arbitrarily large and independent of all platform performance parameters. This puts forth the following fundamental question that is the main motivation for our work.

> *Is it feasible to achieve a* privacy-preserving *and* general-purpose *smart contract functionality under the same availability and decentralization characteristics exhibited by Nakamoto consensus?*

In this work we carve out a large class of distributed computations that we express as smart contracts, including contracts with privacy guarantees, that can be implemented without additional trust assumptions beyond what is assumed for Nakamoto consensus and the existence of a securely generated common reference string[1]. The latter is not an assumption to be taken lightly – however, it is a common requirement for privacy-preserving blockchain protocols with strong cryptographic privacy guarantees. This class allows us to express the protocol logic of dedicated privacy-preserving, ledger-based protocols such as Zerocash [3] as smart contracts and subsumes existing smart contract systems such as Zexe [4], Hawk [22], Zether [6], Enigma [32], and Arbitrum [19]. These protocols mainly rely on either *zero-knowledge* or *signature authentication* for their security. The structure of Kachina is flexible enough to allow contract authors

[1] Recent advances in zero-knowledge, especially for *updateable reference strings*, such as [24] should allow such a reference string to be bootstrapped from consensus alone.

to express each of these systems together with a concise description of the privacy they afford. Thus, KACHINA does not supersede these protocols, but rather gives a common foundation on which one can build further privacy-preserving systems.

1.1 Our Contributions

We make four contributions to the area of privacy-preserving smart contracts: (1) We **model** privacy-preserving smart contracts, (2) we **realise a large class** of such contracts, (3) we **enable concurrent interactions** with smart contracts, without compromising on privacy, and (4) we demonstrate how to **composably build** smart contract systems. Combined, they provide a *practical foundation* for both reasoning about privacy in smart contracts, and constructing a practical, real-world smart contract system with a good foundation for privacy.

Our model. We provide a universally composable model for smart contracts in the form of an ideal functionality that is parameterised to model contracts both with and without privacy. We designed out model to capture a broad range of implementations of smart contracts while still remaining directly applicable in practice. The expressiveness and relative simplicity of our model lends itself to further analyses of smart contracts and their privacy. Moreover, existing privacy-preserving systems benefit from the model as a reference point for cross comparison.

Concretely, we consider a smart contract to be specified by a transition function Δ and a leakage function Λ, which parameterise the smart contract functionality $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$. Δ models the behaviour of the contract were it to run on a local system or by a trusted party with perfect unobservable channels to the parties that interact with it. It is a program that updates a shared state, and has its inputs provided by and outputs returned to the calling party. $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$ models network, ledger, and contract specific "imperfections" that also exist in the ideal world by interacting with a $\mathcal{G}_{\mathsf{ledger}}$-GUC functionality [9], and captures the fundamental, ideal-world leakage through the parameterising function Λ.

Dealing with concurrency in a privacy-preserving manner. There exists a fundamental conflict between concurrency and privacy that needs to be accounted for if we are to remain true to our objective of providing a smart contract functionality with the same decentralisation characteristics as Nakamoto consensus. To illustrate, suppose an ideal smart contract is at a private internal state σ and two parties wish to each apply a function f and g respectively to this state, such that the result is independent of the order of application – i.e. $f(g(\sigma)) = g(f(\sigma)) = \sigma'$. In any implementation of the above (which does not utilise coordination between the parties), the first party (resp. the second) should take into account the publicly known encoding $[\sigma]$ of σ and facilitate its replacement with an encoded state $[f(\sigma)]$ (resp. $[g(\sigma)]$) as it results from the application of the desired transition in each case. It follows now that the encoded states $[f(\sigma)], [g(\sigma)]$ must be publicly reconciled to a single encoded state $[\sigma']$ which necessarily must leak

some information about the transitions f and g. Being able to achieve this type of public reconciliation while retaining some privacy requires a mechanism that enables parties to predict transition conflicts and specify the expected leakage.

We achieve this through the novel concept of *state oracle transcripts*, which are records of what operations and assumptions are performed on, and assumed about, the contract's state. These allow contract authors to optimise when transactions are in conflict and when not, while also ensuring that the leakage is fixed once the transaction is created. We further provide a mechanism for analysing when reordering transactions is safe with respect to a user's private state, by specifying a sufficient condition for which transactions must be declared as dependencies when creating a new transaction.

Our protocol. We construct a practical protocol for realising many privacy-preserving smart contracts utilising only non-interactive zero-knowledge proof systems. The primary goal of this protocol is to provide something sufficiently low-level and general purpose to serve as a basis for building further privacy-preserving systems on top of, without requiring the underlying system to be upgraded with each new extension or upgrade. We focus on a setting where parties may go offline at any time, and do not trust each other. The core idea behind the protocol is to separate a smart contract's state into *shared public* state, and *local private* state, and prove the correctness of updates to the public state in zero-knowledge.

Efficient modular construction. Kachina is designed to be deployed at scale: Previous works using zero-knowledge, do not explicitly maintain a contract state. If such a state σ was modelled anyway, (e.g. as inputs to these systems), the zero-knowledge proofs involved would scale poorly, with a proving complexity of $\Theta(|\sigma|)$ before any computation is performed. For this reason, a naïve approach to state cannot scale to handle systems with a large state – such as a privacy-preserving currency contract, without these being handled as special cases. Our abstracting of state accesses solves this problem:

State oracles also help here, regardless of the size of our state. As the state is never accessed directly, but only through oracles specified by the contract, the complexity of what must be proven is under the full control of the contract author and can be greatly decreased. As a result, a proving complexity of $\Theta(|\mathcal{T}|)$ prior to performing any computation can be expected in Kachina, where $\mathcal{T}$ is either of the oracle transcripts involved. This constitutes a clear improvement, as the state of smart contracts deployed in practice may be very large, however transcripts similarly to the inputs and outputs of traditional, public contract are generally short. This increase in efficiency allows us to present in Appendix J a practical construction of an entire smart contract system, akin to Ethereum [31], *as a* Kachina *contract*.

Not all contracts a user wishes to write will directly match the requirements for realising a Kachina smart-contract. Our model is sufficiently flexible to allow direct application of the transitivity of UC-emulation to solve this: Instead of just

specifying the contract (Δ_r, Λ_r) which should be run, a security-conscious contract author may also write a corresponding *ideal contract* (Δ_i, Λ_i). Our model gives great flexibility for the author to choose what leakage this contract incurs, and what influence the adversary has over its behaviour. If the author then ensures that the implementation of this ideal contract fits KACHINA's realisation requirements, then it is sufficient to prove that the real contract UC-emulates the ideal one – that is, $\mathcal{F}_{\mathsf{sc}}^{\Delta_r,\Lambda_r}$ UC-emulates $\mathcal{F}_{\mathsf{sc}}^{\Delta_i,\Lambda_i}$. By the fact that the real contract meets KACHINA's preconditions, it in turn is UC-emulated by KACHINA – and consequently, by the transitivity of UC emulation, the contract author can securely implement the original, ideal, contract. Te demonstrate this technique we provide a UC realization of the salient features of Zerocash [3] – implemented as a KACHINA contract – in Section 5, and prove that it UC-emulates a much simpler ideal payments contract.

1.2 Related Work

There has been an increasing amount of research into smart contracts and their privacy over the past few years. The results of these often focus on specific use-cases or trust assumptions. We briefly discuss how the most notable of these relate to our work in KACHINA.

Ethereum. As the first practically deployed smart contract system, Ethereum [31] is the basis of a lot of our expectations and assumptions about smart contracts. Ethereum provides no privacy guarantees of any kind, however we refer back to it during this paper when arguing about modelling choices. We assume that the reader has some familiarity with the basic flow of Ethereum, such as contract creation, contract calls, and gas costs.

Zexe. Zerocash [3] is a well-known privacy-preserving payment system, allowing direct private payments on a public ledger. Zexe [4] extends its expressiveness by allowing arbitrary scripts, reminiscent of Bitcoin-scripts, to be evaluated in zero-knowledge in order to spend coin outputs. Zexe itself does not discuss universal zero knowledge, and has no specified mechanism for using the structure of unspent coins to construct larger systems, and therefore by itself suffers from many of the same limitations of expressiveness that Bitcoin-script has. It is a major improvement in expressiveness over Zerocash, which only permits a few types of transactions.

Plutus. Plutus [11] provides a functional interpretation of UTXO ledgers and generalises the UTXO model by introducing an extension that allows for persistent state. While it does not itself consider privacy, it lends itself well to it as a consideration – Plutus makes a similar distinction between off-chain and on-chain code as KACHINA does, and although it does not account for zero-knowledge proofs, adding these is not far-fetched. The extended UTXO model is of particular interest, as in combination with Zexe, and universal zero-knowledge,

it overcomes the former's limits of expressiveness. A combination of these protocols would arrive at a similar result as presented in this paper, although with a more restrictive access to the shared state – due to the reliance on UTXOs, the shared state of Plutus is a set allowing insertions and deletions, while KACHINA permits arbitrary operations.

Hawk. One of the earliest work on privacy in smart contracts, Hawk [22] is also one of the most general. It describes how to compile private variants of smart contracts, given that all participants of the contract trust the same party with its privacy. This party, the "manager", can break the contract's privacy guarantees if corrupt, however it cannot break the correctness of the contract's rules. The construction used in Hawk for the manager party relies of zero-knowledge proofs of correct contract execution, and is closely related to the proofs parties perform in KACHINA. KACHINA's major difference is a greatly relaxed trust model, which allows for contracts in settings where there no trusted third party to facilitate privacy.

Zether. A lot of work on privacy in smart contracts has focused on retro-fitting privacy into existing systems. Zether [6], for instance, constructs a privacy-preserving currency within Ethereum, which can be utilised for a number of more private applications, such as hidden auctions. As with most retro-fitted systems, Zether is constrained by the system it is built for, and does not generalise to many applications.

Enigma. There are two forms of Enigma: A paper discussing running secure multi-party computation for smart contracts [32], and a system of the same name designed to use Intel's SGX enclave to guarantee privacy [14]. The former has a lot of potential advantages, but is severely limited by the efficiency of general-purpose MPC protocols. The latter is a practical construction, and can claim much better performance than any cryptography-based protocol. The most obvious drawbacks are the reliance on an external trust assumption, and the poor track record of secure enclaves against side-channel attacks [5].

Arbitrum. Using a committee-based approach, Arbitrum [19] describes how to perform and agree on off-chain executions of smart contracts. A committee of managers is charged with execution, and, in the optimistic case, simply posts commitments to state updates on-chain. In the case of a dispute, an on-chain protocol can resolve the dispute with logarithmic complexity to the number of computation steps taken. Arbitrum provides correctness guarantees even in the case of a $n - 1$ out of n corrupt committee, however relies on a fully honest committee for privacy.

State channels. State channels, such as those discussed in [13], occupy a similar space to Arbitrum, due to their reliance on off-chain computation and on-chain dispute resolution. The dispute resolution process is different, more aggressively

terminating the channel, and typically it considers only participants on the channel that interact with each other. The privacy given is almost co-incidental, due to the interaction being local and off-chain in the optimistic case. State channels are more restrictive in their expressiveness than Arbitrum, in that only few users can interact with it, a drawback which allows for its main feature however: not requiring on-chain transactions in the optimistic case.

2 Technical Overview

We first aim to establish informally the goals and core technical ideas of this paper. These will be fleshed out in the remainder of the paper's body, with some of the technical details – primarily in-depth UC constructions and proofs – in the appendix. We will discuss each of our contributions in turn, and discuss how combined, they present a powerful tool for constructing privacy-preserving smart contract systems.

Our model. When a user interacts with a smart contract system, two points in time are important: The point where the user decides to query the smart contract, and the point where this query has a lasting effect on the ledger. Typically, and in our model, defined in Section 3, this takes the form of a user *creating a transaction* at one point in time, and this transaction at some point *entering the confirmed ledger state*.

At the core, a smart contract is a reactive state-machine, which alters its state for each user query, according to a *transition function* Δ. A user can not know what state the contract will be in, once one of their transactions is confirmed, and it is quite possible for a transaction to no longer make sense – for instance, two users cannot both spend the same coin. The adversary may provide its own input to the transition function, that allows the adversary to influence the contract's execution as much – or as little – as the transition function allows.

When a user creates a transaction, in broadcasting it he likely leaks *something* about the transaction's content. The transaction the user creates may also depend not just in his original input, but the state of the system, as he sees it. For instance, a user may be able to avoid conflicts with previous transactions they made – such as in the case of spending UTXO-style coins, not spending the same coin as a prior transaction. A *leakage function* Λ is responsible for both deciding what information is leaked, and deciding which information from the point of creating a transaction, may later affect its behaviour. This function also generates a description of the leakage itself, which the user must sign off on before the transaction is broadcast – this is not a technical requirement, but an ethical one: As estimating leakage becomes more complex, users can not be expected to predict the implications of their actions in advance.

The core protocol idea. The kernel of the Kachina protocol, as defined in Section 4, is as follows: We restrict ourselves to contracts which divide their state into a public state σ, and, for each party p, a private state ρ_p. Roughly these

correspond to the shared ledger, and a party's local storage respectively. We constrain the transition function to be over pairs (σ, ρ_p) instead of over *all* private states – i.e. a party may only change their own private state. Honest users will maintain their own private state in accordance with the contracts' rules, while the contract must anticipate that dishonest parties may set it arbitrarily (this can be circumvented by committing to private states, as descripted in Appendix G).

In this setting, we can derive a clear image of what leakage is, and a natural construction based on zero-knowledge proof-systems presents itself: When creating a transaction, a user may evaluate the transition function against the current contract state, and prove the corresponding public state update $\sigma \mapsto \sigma'$ in zero-knowledge. Locally, the user updates his own private state, and publishes a transaction consisting of the transition $\sigma \mapsto \sigma'$, and a proof of this transitions correctness.

State oracles. The core ideas as presented above is naïve in its construction, primarily due to it proving transitions from one public state to another. In practice, a contract does not exist in a vacuum, and other users will interact with it. This is especially true for large systems – if I make an Ethereum transaction, it will almost certainly be applied after many other transactions I do not even know about. As these different states are not the same as the original, the transition cannot be applied, as seen in Figure 1. Instead of capturing a transition from $\sigma \mapsto \sigma'$, we would rather want to capture a *partial function* from states to successor states.

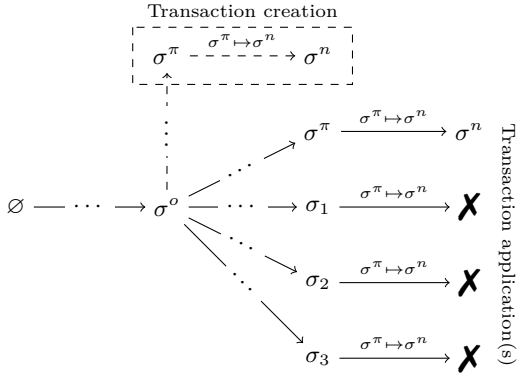

Fig. 1. Direct state-transition based transactions can be applied only in the state σ^π they were proven for.

Our approach for modelling this is to have contracts interact with the state through *oracle queries*, instead of directly – see Subsection 4.1. A contract submits a function q to be run against the current state, this returns a result r, and

an updated state. Instead of proving the state transition correct, we prove that the transition function makes a sequence of queries, given their responses match a corresponding sequence of responses. We refer to this sequence of queries and responses, $((q_1, r_1), \ldots, (q_n, r_n))$, as an *oracle transcript* $\mathcal{T}$. Transactions can be used to determine if a transition makes sense in any given state, and if so, what the new state should be: If each query, applied in turn, returns the expected response, then the transition makes sense, and the same mechanism informs what the resulting state will be. This allows a greater flexibility, sketched in Figure 2, although it is worth noting that some conflicts are inherent, and cannot be resolved. We illustrate our solution, and why it is necessary, in detail in Subsection 4.1.

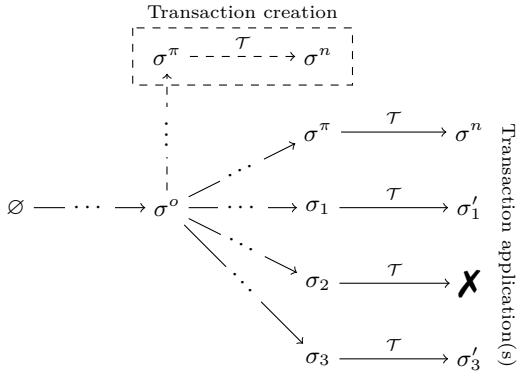

Fig. 2. Oracle-transcript based transactions can be applied in any compatible state.

High-level usage. In practice privacy requirements may not fall within the boundaries of any general system, and will require some adjusting or combining with novel research for their realisation. For instance, building a privacy-preserving currency system is not possible in Kachina directly – after all, it needs the balance to be a shared state, which a user cannot overwrite arbitrarily, but this should also be private – being suitable for neither the shared public state σ, nor the untrusted local private state ρ. In this case, the design of Zerocash [3] can easily be adopted to overcome this issue – an example we work out in detail in Section 5.2. This approach of adding additional layers of privacy is powerful, as can be seen on this example.

This leaves an uncomfortable situation, however: The Zerocash protocol is a poor ideal-world specification, but it is this we can realise directly. We propose using multiple stages of UC-emulation instead: To begin with, we specify as an ideal smart contract, what leakage and transitions a privacy-preserving payments system should have. We then specify the core Zerocash design as a Kachina contract, allowing us to conclude that the ideal Zerocash smart contract is UC-

emulated by KACHINA, parameterised with the Zerocash definition. To bridge the gap, we prove a further UC-emulation: the ideal *private payments* smart contract is UC-emulated by the ideal *Zerocash* smart contract.

This allows us to rely solely on the transitivity of UC-emulation to extend KACHINA with additional privacy features – an approach that is made possible by the design of the ideal smart contract functionality. This functionality provides the abstraction of the *leakage descriptor* to ensure that the two worlds can have the environment sign off on the same leakage, despite underlying differences. Further, while the ideal smart contract's adversarial input a is unused throughout the KACHINA protocol, it is crucial for this stage of the UC-emulation – it affords the simulator a means to influence the ideal private payments contract's execution. In Section 5, where this full analysis is performed, this is used to select party's public keys in the ideal payments contract, for instance. Finally, in Appendix J, we show how a system of multiple interacting contracts can be constructed as an instance of a single, "system" smart-contract.

3 Defining Smart Contracts

In defining smart contracts, their typical implementation as replicated state machines provides a simple solution: If a replicated state machine is the protocol, the single state machine is the model. We assume that, as typically seen, inputs are drawn from a ledger of transactions, and then passed as inputs to the state machine.

Naturally, this definition is unsuitable for privacy-preserving smart contracts: If the state machine's behaviour is known, and its inputs are on a ledger, its behaviour can be simulated by anyone. To allow for privacy then, we ensure that the inputs themselves are not on the ledger – rather a token representing them. The smart contract can ensure the correct input is executed, but an external observer cannot.

3.1 Interactive Automata Interpretation

As mentioned above, smart contracts can be seen as "reactive computation": Parties supply input to the contract, which internally performs computation, potentially involving hidden state. The contract updates its state and sends an output in return back to the original caller. Models of smart contracts may wish for results to be returned asynchronously, and to potentially depend on interactions with other users. This fits less obviously into this model, which assumes each interaction is atomic. However, such contract can be implemented by a party repeatedly sending poll requests for the result. The assumptions made about smart contracts, and those made of trusted third parties are – almost – the same: they will carry out the computation given to them honestly. Typical real-life systems have caveats here of course: 1. They *leak* all computation done, 2. they allow for some *adversarial influence*, and 3. their behaviour may depend on some *context* in addition to the input itself. 1. is a result of the

cryptographic simplicity of the construction, and does not necessarily hold for privacy-preserving systems, although leakage is hard to eliminate completely). 2. is a consequence of the usage of an underlying ledger, which permits adversarial influence. Finally, 3. stems from it not always being possible for a transaction to be applicable in *any* state. As a result, the state against which a user creates a new transaction dictates partially when this transaction makes sense.

To represent the contract, we will consider its state σ, and a transition function Δ. We will adopt the convention that the initial state is denoted by $\varnothing$, even though the contract itself may ascribe a different meaning to this. Δ should be a function which takes as inputs the old state, some input w, and the party identifier of the submitter, p. It should return a new state, σ', and an output y. While this is sufficient for many things, we will also allow as inputs a context z, and an adversarial input a. We assume Δ to be deterministic – nondeterminism can be simulated by including a randomness source in the context, as seen in Subsection 4.3. In sum total, a contract is primarily defined by the following transition function: $(\sigma', y) \leftarrow \Delta(\sigma, p, w, z, a)$.

As mentioned previously, real implementations of smart contracts have some *leakage*. Further, due to the ledger-based nature of them, this leakage occurs separately, before the contract actually evaluates the input. We define a nondeterministic *leakage function* Λ, which takes the input w, the party ID p, and the state of the smart contract in the user's current view, σ_p. This function should return some leakage lkg, which is passed to the adversary.

At the same time, Λ can define what the context z used in the transition function should be. This value is important for allowing the behaviour of a contract depend on when a query was *submitted*, and not just when it is executed – this will be detailed further in Subsection 4.1. As smart contracts are complex objects, the user can not be expected to accurately anticipate what leakage a given query might incur. As a result, we will also return a *leakage descriptor* desc, which the user must sign off on *before* any leakage occurs, or the query is fully submitted.

A few additional variables may affect what gets leaked. In particular, the transactions a user previously made, but which are not yet in the confirmed ledger state may affect the behaviour of new transactions by the same user. For instance, if the user previously spent a coin, they will not attempt to spend it again. For this purpose, each such unconfirmed transaction, identified by some *transaction handle* τ, is passed to the leakage function, as the list of unconfirmed transaction U_p. Further, a mapping T is supplied, mapping each transaction handle to the tuple (p, w, z, a, D), dictating the transaction's owning party, input, context, adversarial input, and lastly dependencies. These are all as discussed before, except dependencies, which will be explained shortly. Finally, the leakage function is given the length of p's view of the ledger, t – named due to its approximation of time.

D represents transaction dependencies, which in this work have proven to be necessary to ensure transactions are executed in the correct order. At a high level, each transaction has associated dependencies, specified as a list of other

transactions, and ideally being ϵ. If a new transaction is processed, each of its dependencies must have previously been successfully processed in the same order, or the new transaction is rejected. These allow enforcing ordering constraints on transactions, and constitute a form of leakage. This is also the final purpose of Λ: to determine which dependencies the new transaction should have. In summary, a contracts leakage and context are computed as follows: $(\mathsf{desc}, \mathsf{lkg}, D, z) \leftarrow \Lambda(t, U_p, T, \sigma_p, p, w)$.

We consider the pair (Δ, Λ) to define a smart contract, and parameterise the ideal smart contract functionality presented in Subsection 3.2 by both.

The general ideal world usage of this model of smart contracts follows this pattern:

1. A party submits a contract input w.
2. The corresponding context and leakage are computed.
3. The party signs off on the leakage description desc, or cancels.
4. The adversary is given (lkg, D), and can supply the adversarial input a.
5. At some later point, the submitting party can retrieve the output of Δ (if any), while other parties can interact with the modified state.

3.2 UC Specification

The *ideal smart contract functionality* $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$ formally captures this notion of a contract as a leaky state transition function that can be queried by parties via a ledger. It is parameterised by the transition function Δ and the leakage function Λ, and it operates in a hybrid world with a ledger functionality $\mathcal{G}_{\mathsf{ledger}}$. A candidate for such a ledger is $\mathcal{G}_{\mathsf{simpleLedger}}$, as introduced in Appendix B, although any compatible functionality is sufficient. To avoid confusion with the various contract state in $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$, we refer to both the authoritative ledger state as well as the party specific prefixes of that state as ledger views.

Functionality $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$ (sketch)

The smart contract functionality $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$ allows parties to query a deterministic state machine determined by Δ and Λ in a ledger-specified order.

Executing a ledger view:

Starting with an initial state $\sigma \leftarrow \varnothing$, and an empty set of confirmed transactions, for each transaction in the ledger's view, if the transaction is unknown, allow the adversary to supply its inputs. Next, verify the transaction's dependencies, and that, for $(\sigma', y) \leftarrow \Delta(\sigma, \ldots)$, $\sigma' \neq \bot$. If both are satisfied, update σ to σ', and record the transaction as confirmed. If an execution output is requested, return y. If, on the other hand, one of the preconditions is not satisfied, skip this transaction and record the transaction as rejected.

Prior to any interaction by p:

Compute which transactions have been rejected in p's view of the ledger state, and remove any unconfirmed transactions for p that – directly or indirectly – depend on them.

When receiving a message (POST-QUERY, w) *from an honest party* p*:*

Retrieve p's current ledger view of length ℓ, and compute the corresponding smart-contract state using Δ. Feed this, together with p's unconfirmed transactions and their inputs, the length ℓ, and the input w to Λ.

Ask p if the leakage description returned is acceptable. If so, query the adversary for a unique transaction ID τ, and some adversarial input corresponding to the leakage, and the transaction's dependencies. Record the original input, the adversarial input, the context returned by Λ, and the transaction's dependencies as being associated with τ and p. Record the transaction as unconfirmed by p, and then send (SUBMIT, τ) to $\mathcal{G}_{\mathsf{ledger}}$, and return τ.

When receiving a message (CHECK-QUERY, τ) *from an honest party* p*:*

If τ is owned by p, and in their current view of the ledger, compute and return the output by executing the ledger view up to τ.

Some combinations of Δ and Λ are not obviously realisable, in particular the more restricted the leakage becomes. They do however give the flexibility necessary to model existing smart contract systems, both privacy-preserving and otherwise. For instance, a leakage function which leaks the input itself, has no context, and no dependencies, corresponds closely to Ethereum [31], while a leakage function returning no leakage makes many transition functions hard or impossible to realise. We will focus on a more interesting middle ground in the rest of this paper. By defining the ideal behaviour to involve the ledger, we avoid having to duplicate the complex adversarial influence of ledger protocols to model the adversarial influence of smart contracts. Further, $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$ makes few assumptions about the ledger, requiring only the common prefix property, and interfaces for submitting and reading transactions to be well defined.

4 The Kachina Protocol

As mentioned in Section 2, a naïve construction divides a contracts state into a shared public state, and a local private states for each party. A user may then prove the validity of any public state transition – that is, that there exists some private state, and some input, such that this transition takes place. This clearly does not scale well, however, as it assumes that the ledger state does not change between the submission and processing of a transaction.

In practice, a user's query may not be evaluated immediately, and the ledger may change drastically in the mean time. Simply proving a direct state transition would lead to a high proportion of queries being rejected. An equally serious flaw with this naïve approach is that NIZK proofs over the entirety of both the public and private contract states may be infeasible in practice. In complex systems such as Ethereum, the state is on the order of hundreds of Gigabytes [15], and only increasing.

We utilise the same trick to circumvent both problems: We make the interaction with the state more abstract, ensuring both that it can be flexible enough to tolerate reordering, and concise enough to allow efficient proofs.

4.1 State Oracles and Transcripts

We introduce the novel tools of *state oracles*, and *state oracle transcripts* to abstract interaction with a contract's state. When abstracting interaction with the public and private states, many options are possible – we could constrain the state to a specific type of data structure, and only support certain operations over it for instance. This is in fact what the extended UTXO model [11], and Zexe [4] do. While this has many practical benefits, and is easy to reason about, it does not, however, enable the action performed on-chain to depend on the chain's most current state, which Ethereum permits.

An example. To better motivate the need for some further abstraction over state interactions (i.e. not proving a state transition directly), we present a representative example of a smart contract, and discuss how different abstractions of its state will affect its ability to function.

In particular, we consider a *sealed bid auction* contract[2]. We will assume that external to this contract exist a notion of time, and two separate privacy-preserving on-chain assets[3], all within some large system the auction contract may interact with (such as those presented in Appendix J). The auction is opened by the *seller* party, and multiple *buyer* parties may bid on it. The structure of this auction allows for the following interactions:

- At initialisation, the seller transfers ownership of asset A to the auction contract, and publicly reveals it (i.e. the buyers can verify the authenticity of the auction).
- Until time t_1, buyers may submit their bids, by transferring some amount of asset B to the auction contract, which remains anonymous.
- Between time t_1 and t_2, buyers may *reveal* their bid. If the buyer's bid exceeds the currently maximum revealed bid, he reveals his committed asset, and both increases the maximum bid, and records himself as the winning bidder. Otherwise, he withdraws his bid from the contract.
- After time t_2, buyers may withdraw any assets they own after the auction, that is either their losing bids or the sold asset for the highest bidder. Sellers on the other hand can withdraw either the highest bid or the original asset if no bids were made.

What state must such an auction contract maintain? We are considering time and asset management to be maintained externally, but nonetheless a little bookkeeping is necessary: The contract will need to publicly maintain a reference to the asset being sold, in order to correctly transfer it at the end. Further, it will need to maintain what the currently winning bid is, along with its value,

[2] This contract is designed to make a good example, not a good auction – we do not recommend using it as presented.

[3] These may be constructed similarly as in Section 5, however should be holdable and spendable by other contracts. We do not go into detail of this construction; this idea is fleshed out in detail in Zether [6].

and some identification of the current winner. It needs to maintain a set of the bids themselves – these may just be some kind of reference for the asset management system to verify, however there is a need to ensure that a buyer cannot create a new bid after time t_1 – something which is not the job of any asset management system. Finally, we keep a further set: bids which have been the maximum revealed at some point, but were then succeeded. Privately, the contract has minimal bookkeeping to do: Parties need to remember which bids are theirs, and what information is needed to open or withdraw them.

Suppose we adopted a naïve approach to state transitions, and proved the transitioning from one state to another directly, with no abstraction of any kind. During the bidding phase it is easily possible for multiple users to attempt to bid simultaneously (especially considering the delay until transactions become confirmed by an underlying ledger). In this case, only one of these transactions will go through – as soon as this transaction changes the state by adding its own bid, the proof of any other simultaneous transaction becomes invalid.

Often simple abstractions are used instead, such as byte-level access. This would allow a buyer and the seller to withdraw concurrently – as their withdraws affect different parts of the state. It does not work as desired in all cases even in this example – assume a singly-linked list is used to store the set of bids. If two users attempt to add their own bid to this set simultaneously, they will both attempt to overwrite the same pointer.

Clearly this is not necessary – a smart abstraction would realise that whichever user bids first, the resulting set of bids is the same, even if its binary representation may not be. It is not always the case that the order does not matter – for instance when claiming the maximum bid later in the auction, if Alice wishes to claim her bid of 5 is the maximum, she cannot do this after Bob has claimed his bid of 7. Even here a smart abstraction buys us something however – while Alice's transaction may get rejected if placed after Bob's, the other way around is fine!

General-purpose state oracles. The solution we propose is to allow the contract itself to define how state is updated – specifically, instead of sending the instruction $(\textsc{append}, x)$, it may send *a program* encoding, for instance, "increment the length l of the array by 1, and insert a pointer to x at position $l-1$". More formally, the generalisation of an abstract data type that the contract may make queries to, is that of a universal machine, operating over an internal state, which the contract may query with contract specific programs.

We define $\mathcal{O} \leftarrow \mathcal{U}(\sigma, z)$ to be a *universal state oracle*. It maintains internally the *current state* σ, the *context* z, and a *transcript* $\mathcal{T}$. Initially, these are set to (σ, z, ϵ). The oracle can be interactively queried with a sequence of inputs $q_1, \ldots, q_n$. For each query q_i that is made, the oracle in state $(\sigma, z, \mathcal{T})$ computes $(\sigma', r_i) \leftarrow q_i(\sigma, z)$, and updates its internal state to $(\sigma', z, \mathcal{T} \parallel (q_i, r_i))$, before finally returning r_i as a response to the query. An exception to this is if $\sigma = \bot$, in which case σ' and r_i will also be $\bot$. This is the *error state*. We define $\mathsf{state}(\mathcal{O})$ to return $(\sigma, \mathcal{T})$, out of its internal state.

The *context* z encodes additional information a party *expects* the oracle to have access to. The need for this context will be detailed later, once the basic functionality is established. If $z = \varnothing$, we may write $q_i(\sigma)$, and omit z. The oracle may itself *abort*, in which case we assume any function using it directly will return $\perp$ (in particular Γ, as used in Subsection 4.3). An overview of the interactions of Γ with public and private state oracles may be found in Figure 3.

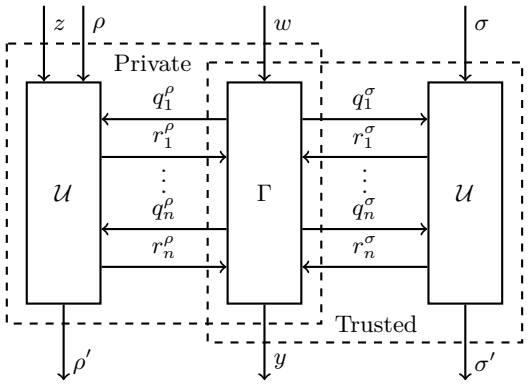

Fig. 3. The interaction of Γ (see Subsection 4.3), with Γ having oracle-access to two separate states: the public state σ, and the private state ρ, together with the context z.

Crucially, we note that the definition above allows retrieving the queries made to the oracle, and the responses given by it. We will refer to this sequence of queries and responses as the *oracle transcript* $\mathcal{T}$. This transcript is crucial in the functioning of Kachina, as it provides a means to decouple the part of a transaction which is proven in zero-knowledge from both the public and private states entirely. Specifically, we prove that given some input, and a sequence of responses recorded in a transcript, the smart contract will make the corresponding queries.

Revisiting our example. To see how this works in practice, we consider what interactions (besides those verifying time, and the authenticity of the assets transferred) are made through the public state oracle in our example of a sealed bid auction. To be more concrete, let us assume that the public state σ consists of the following elements: 1. A public key of the seller, allowing it to withdraw: pk_s, 2. A non-private representation of asset B, the item for auction: b, 3. A non-private representation of the current maximum bid of asset A: a, 4. The value of the bid in 3.: v, 5. The public key of the current maximum bidder, pk_b, 6. A set of private representations of asset A of bids: S, and 7. A set of non-private representations of asset A of previous maximum bids, R. We write $\sigma = (\mathsf{pk}_s, b, a, v, \mathsf{pk}_b, S, R)$, initialised by the seller to $(\mathsf{pk}_s, b, \varnothing, 0, \varnothing, \varnothing, \varnothing)$.

The private state ρ by contrast consists of three variables describing an unopened bid: the opening pubBid that is eventually to be made public, a hiding commitment *which* it is an opening for, bid, and its value v, all of which may be set to $\bot$: $\rho = (\mathsf{pubBid}, \mathsf{bid}, v)$

For each of the (non-initialising) interactions previously listed, we demonstrate the corresponding oracle queries made:

- Bidding: Given a non-private asset pubBid, with value v, corresponding to a private asset bid, which has been bound to the auction contract, Γ first makes the following public oracle query:

  function $\mathsf{makeBid}_{\mathsf{bid}}((\mathsf{pk}_s, b, a, v, \mathsf{pk}_b, S, R))$
  return $((\mathsf{pk}_s, b, a, v, \mathsf{pk}_b, S \cup \{\mathsf{bid}\}, R), \top)$

  Further, it makes the following private oracle query:

  function $\mathsf{recordBid}_{\mathsf{pubBid},\mathsf{bid},v}(\cdot)$
  return $((\mathsf{pubBid}, \mathsf{bid}, v), \top)$

- Revealing: Given a public key to redeem the funds to in case of losing the auction, Γ first makes a private oracle query to retrieve which bid is owned:

  function $\mathsf{retrieveBid}((\mathsf{pubBid}, \mathsf{bid}, v))$
  return $((\mathsf{pubBid}, \mathsf{bid}, v), (\mathsf{pubBid}, \mathsf{bid}, v))$

  Next, the contract makes a public oracle query to determine if the buyer holds the current maximum bid:

  function $\mathsf{isMax}_v(\sigma = (\ldots, v_o, \ldots))$
  return $(\sigma, v > v_o)$

  If this query returns $\top$, the contract claims the maximum bid with the query:

  function $\mathsf{claimMax}_{\mathsf{pubBid},\mathsf{bid},v,\mathsf{pk}}((\mathsf{pk}_s, b, a_o, \cdot, \mathsf{pk}_o, S, R))$
  assert $\mathsf{bid} \in S$
  return $((\mathsf{pk}_s, b, \mathsf{pubBid}, v, \mathsf{pk}, S \setminus \{\mathsf{bid}\}, R \cup \{(a_o, \mathsf{pk}_o)\}), \top)$

  If the original value test fails, on the other hand, instead the contract transfers the ownership of bid via the underlying asset system to pk, and runs:

  function $\mathsf{claimLoss}((\mathsf{pk}_s, b, a, v, \mathsf{pk}_o, S, R))$
  return $(\top, (\mathsf{pk}_s, b, a, v, \mathsf{pk}_o, S \setminus \{\mathsf{bid}\}, R))$

- Withdrawing: Given a public key pk, which the caller knows the corresponding secret key for, the contract will make an oracle query to determine which assets to transfer ownership of, and to un-record them:

  function $\mathsf{withdraw}((\mathsf{pk}_s, b, a, v, \mathsf{pk}_b, S, R))$
  if $\mathsf{pk} = \mathsf{pk}_s \wedge a \neq \varnothing$ **then return** $((\varnothing, b, \varnothing, \varnothing, \mathsf{pk}_b, S, R), a)$
  else if $\mathsf{pk} = \mathsf{pk}_b \wedge b \neq \varnothing$ **then return** $((\mathsf{pk}_s, \varnothing, a, v, \varnothing, S, R), b)$
  else if $\exists c : (c, \mathsf{pk}) \in R$ **then return** $((\mathsf{pk}_s, b, a, v, \mathsf{pk}_b, S, R \setminus \{(c, \mathsf{pk})\}), c)$

While this example does not fully handle all corner cases (such as a buyer bidding multiple times), it is hopefully sufficient to illustrate the core idea: The query made, and the response to it, will often be the same even if another party has since interacted. Two bid do not conflict, as the oracle queries each makes can be run even if the contract's state has since changed. We also do not need to

concern ourselves with the representation: Whether the sets are implemented as linked lists, arrays, or balanced trees does not matter – the abstract data type behaves the same, after all.

Transcript application. We define some shorthand notations for talking about transcripts: For a given transcript $\mathcal{T}$, state σ, and context z, we write $\mathcal{T}(\sigma, z)$ as a shorthand for the following operation:

function $\mathcal{T}(\sigma, z)$
  let $\mathcal{O} \leftarrow \mathcal{U}(\sigma, z)$
  for $(q_i, r_i) \in \mathcal{T}$ **do**
    send q_i **to** $\mathcal{O}$ and **receive the reply** r
    if $r \neq r_i$ **then return** $\bot$
  let $(\sigma', \cdot) \leftarrow \mathsf{state}(\mathcal{O})$
  return σ'

If $z = \varnothing$, we may write $\mathcal{T}(\sigma)$, and omit z. If $\mathcal{T}$ is malformed (i.e. not a sequence of pairs), we assume $\bot$ is returned.

We further define the application of transcript sequences: Given a sequence X of transcript and context pairs $(\mathcal{T}, z)$, we write $\mathcal{T}^*_X(\sigma)$ for the transcript sequence application. Specifically, we define $\mathcal{T}^*_\epsilon(\sigma) := \sigma$, and $\mathcal{T}^*_{X \parallel (\mathcal{T}, z)}(\sigma) := \mathcal{T}(\mathcal{T}^*_X(\sigma), z)$.

Transcript oracles. For a given transcript $\mathcal{T}$, we define the *transcript oracle* $\mathcal{O}(\mathcal{T})$ to act as an oracle with the following behaviour: Internally, it keeps a counter i, initialised to 1. Let $\mathcal{T} = (q_1, r_1) \parallel \ldots \parallel (q_n, r_n)$. Then, when receiving a query q, check if $q = q_i$. If so, increment i, and return r_i. If $q \neq q_i$, or $i > n$, abort. For transcript oracles, we define the additional function $\mathsf{consumed}(\mathcal{O})$, which returns $\top$ if and only if $i + 1 = n$, i.e. each recorded query has been processed by the oracle. We refer to a transcript as *minimal* in some usage context, if once the usage context, given oracle access to the transcript oracle returns, $\mathsf{consumed}$ holds.

Oracles, transcripts, and interchangeability. The core premise of Kachina relies on a few key observations on how transcripts relate with the original, universal state oracles. First, we observe that if $\sigma' = \mathcal{T}(\sigma, z) \neq \bot$, then σ' is indistinguishable to having been produced with the universal oracle $\mathcal{U}(\sigma, z)$, by the nature of the definition of transcript application. This lets us, given the transcript and context, reproduce the behaviour of some function which was parameterised by the transcript, even if we do not know *why* this function made the queries it did. In the protocol, this is primarily used to replicate the affects other users' queries have on the public state.

Further, we observe that if the transcript oracle $\mathcal{O}(\mathcal{T})$ doesn't abort when used as an oracle in some function, then it behaves identically to the original universal oracle that was used to generate the transcript. We use this fact to generate zero-knowledge proofs about transactions – we prove that each oracle query in a transcript was made, and that the behaviour is correct, *given the responses the transcript claims.* As a caveat here, we further prove that $\mathsf{consumed}(\mathcal{O})$, in

order to be able to claim that the transcript doesn't just start with the queries an honest execution would make, but matches them exactly.

To tie things together, the basic function is that Alice generates a transcript for the oracle accesses that her private computation performs. She proves this transcript correct, and minimal. She sends the transcript and proof to Bob, who is convinced of the correctness and minimality, and can therefore replicate the effects of the transcript by applying it to the state directly.

Inherent conflicts. While the approach of abstracting the interaction with the state away has many benefits, it relies on the contract itself making use of them in such a way that the original problems of resilience to reorderings are not simply pushed down the line. For instance, a contract which offers some currency to the first party to interact with it clearly is not resilient to reordering, by its very design. Nonetheless, two parties may both see the contract, see the funds it holds unclaimed, and both decide to claim them. This *inherent conflict* cannot be resolved, simply because the parties are acting concurrently.

It is further possible for a contract to be poorly designed, and introduce conflicts through that. Suppose a contract is auctioning an item, and two parties wish to bid. A poorly programmed design may have an array of bids in the public state, and may see both parties attempting to place their bid into the first cell of the array. A functionally equivalent approach, minimising conflicts, would instead instruct the public state to insert at the end of the array, regardless of how many items are occupied.

A note on context. The design of oracles for interacting with the state of the contract is convenient when this response of the oracles is unlikely to change over time. In some cases, however this is just not feasible, and the contract may need advice about highly volatile state. For instance, the Merkle tree roots used in the Zerocash [3] protocol change with every transaction processed, but need to have statements proved about them.

To solve this problem, we run the private state oracle in the context of the view of the submitting party at the point of transaction creation. Specifically, it can not only read and write the current private state, but can also read the public and private states at the time when the query was submitted. We also provide the contract access to "optimistic projections" of the state, which are the public and private states if all unconfirmed transactions were to appear in order immediately after the parties current ledger state. This allows the party to avoid conflicting with their own unconfirmed transactions – for instance in the Zerocash contract described in Section 5.2.

Finally, we provide a randomness source η as part of the context – this is chosen as well at the time of query, and ensures that – for honest parties – the private state oracle may behave nondeterministically. Formally, the context z in KACHINA is a tuple $(\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta)$, where (σ^o, ρ^o) is the querying parties view of the contract state at the time of submission, and (σ^π, ρ^π) is their projected view of the contract.

4.2 The Challenge of Dependencies

Consider two transactions, τ_1 and τ_2 being reordered. If τ_1 moves funds from Alice to Bob, and τ_2 from Bob to Charlie, it is clear that the sequence $\tau_2 \ldots \tau_1$ may not be valid, if Bob's transfer relies on the funds he receives from Alice. Fortunately, if conflicts occur on the public state, such as in this case, the ledger protocol catches them, and ensures that the invalid transactions are ignored.

The possible effects on a users private state are more challenging to understand, however. While two different parties cannot conflict with each other on private state changes, due to their domain separation, parties can encounter *internal* conflicts. Consider that p, starting with the private state ρ_1, makes the transaction τ_1 which advances their private state to ρ_2. Afterwards, they make transaction τ_2, ending in private state ρ_3. Now, if these transactions are made close to each other, it is possible for them to be reordered on the ledger. However the transaction τ_2 may – from the perspective of private state updates – make no sense coming first. Indeed, the corresponding private state transcript $\mathcal{T}_{\rho,2}$ may be such that $\mathcal{T}_{\rho,2}(\rho_1, z_2) = \bot$. What then? Should the user act as if the transactions had happened in the right order, potentially leading to confusing inconsistencies between the private state, and the shared public state? Should the user faithfully apply $\mathcal{T}_{\rho,2}$, and accept that the contract has encountered a catastrophic failure? Obviously both are not ideal, and the more practical solution is to claim that τ_2 *depends* on τ_1. This information can be published as part of the transaction, and their ordering enforced by the on-chain validation.

What remains unclear is what the dependencies of a new transaction should be. If a user has a set of unconfirmed transaction U, and is adding the new transaction τ in the ledger state Σ, naïvely what dependencies should capture a constraint over which permutations of the unconfirmed transactions are permissible. More precisely, suppose we are given a sequence of transactions Σ', where Σ is a prefix of Σ', and Σ' both contains all transactions in $U \cup \{\tau\}$, *and* they are all "valid". Then taking the permutation of $U \cup \{\tau\}$ of transactions from Σ', and applying the corresponding private state oracle transcripts in order to the current private state should give a sane, non-$\bot$, result regardless of the permutation.

As proving that some transactions are non-conflicting may be application specific, instead of having a general-purpose means to capture what a transactions dependencies are, we instead allow a dependency function dep to be a parameter, and constrain how this must behave. We can also provide a "fallback" dependency function, which is overly aggressive in which transactions it depends on.

Formal definition. More formally, dep is a pure function, over the following items: a) a set of unconfirmed transactions, their associated private state transcript, their associated context, and their respective dependencies. b) the new transaction's private state transcript, and c) the new transaction's context (including the confirmed private state, ρ^o. It returns a sequence of transactions which must

all proceed the new transaction, in the specified order. In mathematical notation, $D \leftarrow \mathsf{dep}(X, \mathcal{T}, z)$, where $\forall x \in X : \exists \tau, \mathcal{T}_\tau, z_\tau, D_\tau : x = (\tau, \mathcal{T}_\tau, z_\tau, D_\tau)$.

To begin with, we introduce some mathematical notation: First, we write S_X for the set of all permutations of the set X, where each permutation is assumed to be a list. Second, we write $X \sqsubseteq Y$ for the *subsequence relation*, where one list is a subsequence of another if and only if the items of the first are found in the second in the same order:

$$X \sqsubseteq Y := X \subseteq Y \wedge (\forall a, b \in X : \mathsf{idx}(X, a) < \mathsf{idx}(X, b) \implies \mathsf{idx}(Y, a) < \mathsf{idx}(Y, b))$$

To define the constraints of dep, we also define *dependency satisfaction*. In the actual protocol, a transactions dependencies are satisfied if two conditions hold: a) Each of the dependencies are confirmed transactions, and b) $D \sqsubseteq \Sigma$, where Σ is the ledger of transactions. In the context of generating dependencies, we have a more relaxed approach, and want to consider only the permutations of unconfirmed transactions which *could possibly* be satisfied. We therefore ignore dependencies outside of this set of unconfirmed transactions. For instance, if we have two unconfirmed transactions, a and b, where b depends on ac, we ignore the dependency on c, and we consider the sequence ab to satisfy dependencies, but ba to not do so. Formally, the predicate $\mathsf{sat}(X, U)$ takes some form of permutation of X in dep, and a set of unconfirmed transactions U. We define $\mathsf{sat}(\epsilon, U) := \top$, and $\mathsf{sat}(X \parallel (\cdot, \cdot, \cdot, D), U) := \mathsf{sat}(X, U) \wedge (D \cap U) \sqsubseteq \mathsf{map}(\mathsf{proj}_1, X)$.

We now define the invariant $J(X, \rho)$, stating that given the sequence X, and state ρ, for any permutation of any subset of X, if dependencies are satisfied, the corresponding private state transcripts can be applied in the permutation order without aborting: $J(X, \rho) := \forall Y \subseteq X, Z \in S_Y : \mathsf{sat}(Z, \mathsf{map}(\mathsf{proj}_1, X)) \implies \mathcal{T}^*_{\mathsf{map}(\mathsf{proj}_2 \times \mathsf{proj}_3, Z)}(\rho) \neq \bot$.

We can now define the constraints on dep:

1. If called with non-honestly generated transcripts or contexts, no constraints need to hold.
2. The result must be a subsequence of the transactions in X:
   $\mathsf{dep}(X, \mathcal{T}, z) \sqsubseteq \mathsf{map}(\mathsf{proj}_1, X)$
3. When adding a new transaction τ, with the corresponding transcript $\mathcal{T}$ and context z, the invariant J is preserved:
   $\textbf{let } Y = X \parallel (\tau, \mathcal{T}, z = (\cdot, \rho^o, \cdot, \cdot, \cdot), \mathsf{dep}(X, \mathcal{T}, z)) \textbf{ in } \mathcal{T}^*_{\mathsf{map}(\mathsf{proj}_2 \times \mathsf{proj}_3), Y}(\rho^o) \neq \bot \wedge J(X, \rho^o) \implies J(Y, \rho^o)$

Finally, we note that the following dependency function will always satisfy the constraints: $\mathsf{dep}(X, \mathcal{T}, z) = \mathsf{map}(\mathsf{proj}_1, X)$. This is as it maximally constrains the possible permutations which satisfy the dependencies.

4.3 The Contract Class

The class of contracts achievable by KACHINA, $\mathbb{C}_{\text{KACHINA}}$, is defined primarily by constraining them to a modified transition function, Γ. This transition function

is given oracle access to the calling user's private state ρ_p, and a shared public state σ, with oracle accesses functioning as described in Subsection 4.1, and Subsection 4.1. The adversary is permitted to program its own private state oracle arbitrarily – this corresponds to local computation, after all. In addition to Γ, the protocol is parameterised by a leakage descriptor desc, which receives the time t, the sequence of unconfirmed transactions, their private state transcripts and dependencies, X, both public and private state transcripts $\mathcal{T}_\sigma$ and $\mathcal{T}_\rho$, the original input w, and the context z. Finally, it is parameterised by a dependency function dep, which is under the constraints noted in Subsection 4.2.

Formally, $(\Delta_{\text{KACHINA}}, \Lambda_{\text{KACHINA}}) \in \mathbb{C}_{\text{KACHINA}}$ if and only if there exist corresponding Γ, desc, and dep, such that they are derived approximately as follows:

Transition Function Δ_{KACHINA} (sketch)

When receiving an input $((\sigma, \boldsymbol{\rho}), p, w, (\mathcal{T}_\sigma, z), \cdot)$*:*

if $\mathcal{T}_\sigma(\sigma) = \bot$ **then return** $(\bot, \bot)$

let $(\sigma, \cdot, \boldsymbol{\rho}[p], \cdot, y) \leftarrow \mathsf{run}\text{-}\Gamma(\sigma, \boldsymbol{\rho}[p], w, z, p \in \mathcal{H})$

return $((\sigma, \boldsymbol{\rho}), y)$

Where $\mathsf{run}\text{-}\Gamma(\sigma, \rho, w, z, \cdot)$ runs $\Gamma_{\mathcal{O}_\sigma, \mathcal{O}_\rho}(w)$, and returns $(\sigma', \mathcal{T}_\sigma, \rho', \mathcal{T}_\rho, y)$ (see Appendix C for a full specification).

Leakage Function Λ_{KACHINA} (sketch)

When receiving an input $(t, U, T, (\sigma^o, \boldsymbol{\rho}^o), p, w)$*:*

Simulate applying all unconfirmed transactions in order, for a new *projected state* (σ^π, ρ^π). Select a randomness stream η, and set the context z to the old state $(\sigma^o, \boldsymbol{\rho}^o[p])$, the projected state (σ^π, ρ^π), and η. Run Γ against this projected state and context, and retrieve the new states and transcripts $\mathcal{T}_\sigma$, $\mathcal{T}_\rho$. Compute the dependencies D and leakage description $\mathsf{description}$, and return $(\mathsf{description}, \mathcal{T}_\sigma, D, (\mathcal{T}_\sigma, z))$.

Some corner cases have been omitted; the full version of these functions may be found in Appendix C.

4.4 The KACHINA Protocol

The construction of the protocol itself is now fairly straightforward – as already noted, instead of proving statements about transition functions directly interacting with the state, it uses non-interactive zero-knowledge to prove statements about transition functions interacting with an oracle. Specifically, when creating a transaction, users prove that the generated transcript is consistent with the transition function and initial input. Instead of evaluating transactions, users apply the public (and, if available, private) state transcripts associated with them. We sketch the protocol here, the full details can be found in Appendix C.

Protocol KACHINA **(sketch)**

The KACHINA protocol realises the ideal smart contract functionality when parameterised by a transition function Γ, a leakage descriptor desc, and a dependency function dep, such that the corresponding (Δ, Λ) pair is in $\mathbb{C}_{\text{KACHINA}}$. It operates in the $(\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}, \mathcal{G}_{\mathsf{simpleLedger}})$-hybrid model, where $((\mathcal{T}_\sigma, \cdot), (w, \mathcal{T}_\rho)) \in \mathcal{L}$ iff $\Gamma_{\mathcal{O}(\mathcal{T}_\sigma), \mathcal{O}(\mathcal{T}_\rho)}(w)$ does not abort, and exists normally.

Executing a ledger state:

Starting with an initial state $(\sigma, \rho) \leftarrow (\varnothing, \varnothing)$, and an empty set of confirmed transactions, for each transaction in the ledger verify their dependencies, and that $\mathcal{T}_\sigma(\sigma) \neq \bot$. If both are satisfied, apply $\mathcal{T}_\sigma$, and if available, the corresponding $\mathcal{T}_\rho$, and mark the transaction as confirmed. Otherwise, skip it.

Prior to any interaction:

Compute which transactions have been rejected in the ledger state, and remove any unconfirmed transactions that – directly or indirectly – depend on them.

When receiving a message $(\text{POST-QUERY}, w)$ *from a party* p*:*

Read the ledger state, and compute the corresponding smart-contract state (σ^o, ρ^o). Create a projected contract state (σ^π, ρ^π) by applying in order the transcripts from unconfirmed transactions to the already computed contract state.

Select a randomness stream η, and set the context z to the old state (σ^o, ρ^o), the projected state (σ^π, ρ^π), and η. Run Γ against against this projected state and context, and retrieve the new states and transcripts $\mathcal{T}_\sigma$, $\mathcal{T}_\rho$, as well as the output y. Compute the dependencies D and leakage description $\mathsf{description}$.

Ask p if $\mathsf{description}$ is an acceptable leakage. If so, create a NIZK proof π that $((\mathcal{T}_\sigma, D), (\mathcal{T}_\rho, w)) \in \mathcal{L}$. Record $\mathcal{T}_\rho$ and z, and the result y, and publish on $\mathcal{G}_{\mathsf{ledger}}$, record as unconfirmed, and return the transaction $(\mathcal{T}_\sigma, D, \pi)$.

When receiving a message $(\text{CHECK-QUERY}, \tau)$ *from a party* p*:*

If τ is in the current view of the ledger, execute the ledger and check if τ is confirmed, and an output for it has been recorded. If so, return the recorded value.

Theorem 1. *For any contract* $(\Delta, \Lambda) \in \mathbb{C}_{\text{KACHINA}}$*,* KACHINA *UC-emulates* $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$*, in the* $\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}$*-hybrid world, in the presence of* $\mathcal{G}_{\mathsf{simpleLedger}}$*.*

We prove Theorem 1 through a detailed case-analysis of any action an environment, in conjunction with the dummy adversary, may take. The full case analysis may be found in Appendix D, however we provide a general intuition here. We define an invariant $\boldsymbol{I}$ between the real and ideal executions in the UC security statement, roughly encoding that "the real and ideal states are equivalent". This ranges from simple equivalences, such as them having the same ledger states, or the same NIZK proofs considered valid, to complex invariants, such as all unconfirmed honest transactions satisfying the sub-invariant for dependencies defined in Subsection 4.2. This invariant is used to argue that the environment, in combination with a dummy adversary, cannot distinguishing between the real and ideal worlds. Specifically, for any action the environment takes, $\boldsymbol{I}$ is pre-

served, and from $\boldsymbol{I}$ holding, we can conclude that the information revealed to it, or the dummy adversary, is insufficient to distinguish the two worlds.

The simulator for Kachina is quite straight forward; it simply creates simulated NIZK proofs for all honest transactions, and forces the adversary to reveal witnesses to the simulated NIZK functionality in time for these to be input to the ideal smart contract. Fundamentally, the security proof relies on state transcripts, as defined in Subsection 4.1, being interchangeable with full state oracles in the same setting, and this setting being enforced by both the protocol and functionality.

While a lot of factors must be formally considered, this is derived from receiving NIZK proofs as part of valid transactions, which prove precisely that if the preconditions for the transaction are met, then the update performed on the public state is the same. The private state is a little more tricky, but is guaranteed by the dependency invariant J holding for honest parties. This lets us similarly argue that the private state transcript will have the same effect as the ideal-world execution.

5 A Case Study: Private Payments

In order to demonstrate the versatility of the Kachina protocol, we take a closer look at a specific smart contract which is prone to many of the issues Kachina addresses: The token contract. Token contracts are well understood when no privacy is desired, with well established standards such as ERC-20 [28] defining their usage. While this standard has its complexities, the basic idea is simply to maintain a mapping of "addresses" (hashes of public keys) to balances in the contracts public state. It is clear that this can be used to construct *public* currencies in smart contracts, given some means of distributing tokens. We write the first *provably private* token contract to demonstrate the expressive power of Kachina.

Having a private token contract also means that we do not have assume the existence of an external private currency, simplifying our model. Without a currency to back it, smart contracts can hardly exist. First users would lack an asset to *write contracts about*. Second, the well established mechanism of transaction fees and gas costs for denial-of-service mitigation requires a currency system. Even with these, denial of service is a delicate act, as seen in attacks on a bad cost model in Ethereum [30]. A private token system allows such a mechanism to be implemented within our model, as we discuss in Subsection J.5.

5.1 Indirect Construction

Given that Kachina is concerned with privacy, it is a natural question whether it allows for constructing a private variant of such a token contract. Following the design of Zerocash [3], it is possible to write a contract that maintains the necessary Zerocash secrets: coin randomnesses, commitment openings, and secret keys. The private state oracle can then compute the off-chain information

required to make a Zerocash transaction: Merkle-paths to your own commitments, the selection of randomness for new coins, and the encryption of the secret information of these coins. This information can then be handed to the central, provable core of the contract, which computes a coin's serial number, verifies the Merkle-path, and verifies the integrity of the transaction. Finally, the serial number and new commitment are sent to the public state oracle, which ensures the former is new, and adds the latter to the current tree.

A reasonable objection to this design may be that the contract itself is no longer self-evidently correct – indeed specifying what this Zerocash contract does, and proving that it achieves this is a separate issue. This is something that applies in many cases – few problems fall neatly into the state separation model of KACHINA. We can see the KACHINA contract here as a middle ground in the UC emulation proof, and define ideal transition and leakage function for a private payments system. We can then show that $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$ parameterised with these is UC-emulated by $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$ parameterised with the Zerocash contract. Finally, the actual realisation is achieved by applying the KACHINA protocol itself to this contract – by the transitivity of UC-emulation.

This has the advantage of having a standardised format and interface for privacy preserving smart contracts, while acknowledging that often privacy is non-trivial, and requires substantial additional cryptographic work. Private payments systems are precisely such a scenario, where fortunately the work has already been done by Zerocash – which we can precisely capture in our model.

5.2 Simplified Construction of Private Payments

For simplicity, in particular on the external interface, we make use only of single denomination coins, although we note that the same technique – with some caveats about leakage – applies for the full Zerocash protocol.

Ideal Private Payments We present first the formal specification of the ideal private payments smart contract. This consists of the corresponding transition and leakage functions, Δ_{pp}, and Λ_{pp}, and supports the following operations: a) INIT, giving a party a unique public key, b) (SEND, pk), sending one of the parties own coins to the public key pk, c) MINT, creating a new coin for the calling party, and d) BALANCE, returning the calling party's balance.

Transition Function Δ_{pp} (sketch)

The state transition function for a private payments system. Parties have associated public keys, and balances. The payments system allows for parties without a public key to generate one, and for parties to transfer and mint single-denomination coins, as well as query their own balance.

When receiving an input $(\sigma, p, \text{INIT}, \cdot, \mathsf{pk})$*:*

Assert that p's public key is not set, and ensure the uniqueness of the adversarial input pk (e.g. by incrementing it until it is an unused public key). Record pk as p's public key, and return it.
When receiving an input $(\sigma, p, (\text{SEND}, \mathsf{pk}), \cdot, a)$*:*
If p is honest, spend from their associated public key. If not, the adversarial input is the public key to spend from, asserting it is not associated with any honest party. Assert the spending public key's balance is positive, and decrease it by 1. Increase the receiving public key pk's balance by 1.
When receiving an input $(\sigma, p, \text{MINT}, \mathsf{pk}, \cdot)$*:*
Increase pk's balance by 1.
When receiving an input $(\sigma, p, \text{BALANCE}, B, \cdot)$*:*
Return the balance B.

Leakage Function Λ_{pp} (sketch)

Each operation on Δ_{pp} has minimal leakage, revealing only which operation was performed, and in the case of a transfer, the time and the recipient – if and only if the recipient is corrupted.

When receiving an input (t, U, T, σ, p, w)*:*
Reject initialisation transaction if σ is already initialised, or a transaction in U is an initialising transaction. Reject spending transactions if the coins held in σ, minus the coins spend in each transaction in U is not greater than zero.
Leak the type of transaction (INIT, SEND, MINT, or BALANCE). If the transaction is SEND, leak the time t, and, if the receiving public key is adversarial, the recipient. There are no dependencies. In the case of minting, provide the calling party's public key as a context, in the case of balance queries, combine the available balance and provide this as a context.

The Zerocash KACHINA Contract The corresponding contract of Zerocash, which is both in the KACHINA class of contracts, and realises the private payments contract, is presented below, along with a proof of it belonging to the KACHINA class – mainly consisting of proving the lack of dependencies to be permissible.

Transition Function Γ_{zc} (sketch)

The state transition function for a Zerocash-based token contract.

When receiving an input INIT*:*
Instruct the private state oracle to sample new Zerocash secret keys, and record them in the private state. Return the corresponding public keys.
When receiving an input $(\text{SEND}, (\mathsf{pk}_z, \mathsf{pk}_e))$*:*
Instruct the private state oracle to process new messages in the public state context for both the confirmed and project state. Have the oracle select a coin held in both states to spend, and return all corresponding secrets, as well as a

Merkle tree root it is contained in, and a path to it. Assert the validity of the Merkle path, and the correct computation of the coin. Compute it's serial number, and instruct the public state oracle to mark this serial number as spent (asserting its uniqueness), and assert that the provided Merkle tree root exists. Finally, the private state oracle computes a coin commitment owned by pk_z, with the commitment secrets encrypted with pk_e. The public state oracle inserts this new commitment to the set of commitments, and appends the message to the list of messages, updating the set of past Merkle tree roots.

When receiving an input MINT*:*

Instruct the private state oracle to assert the existence of secret keys. It must then sample a new coin commitment owned by the recorded private key, and record the commitment and associated secrets as a held coin. The commitment is passed to the public state oracle, which is instructed to record it in the set of commitments, and update the list of past Merkle roots.

When receiving an input BALANCE*:*

Instruct the private state oracle to process new messages in the public state context for both the confirmed and projected state. The oracle then returns the size of the intersection of the sets of held coins in the confirmed and projected private states. This is returned by this query.

function $\mathsf{dep}_{\mathsf{zc}}(X, \mathcal{T}, z)$
  return ϵ

function $\mathsf{desc}_{\mathsf{zc}}(t, \cdot, \cdot, \cdot, w, \cdot)$
  if $w = \text{INIT}$ **then return** INIT
  else if $\exists \mathsf{pk} : w = (\text{SEND}, \mathsf{pk})$ **then return** $(\text{SEND}, t, \mathsf{pk})$
  else if $w = \text{MINT}$ **then return** MINT
  else if $w = \text{BALANCE}$ **then return** BALANCE
  else return $\bot$

Lemma 1. Γ_{zc}, $\mathsf{dep}_{\mathsf{zc}}$, $\mathsf{desc}_{\mathsf{zc}}$ *define* $(\Delta_{\mathsf{zc}}, \Lambda_{\mathsf{zc}}) \in \mathbb{C}_{\text{KACHINA}}$.

Proof (sketch). $\mathsf{desc}_{\mathsf{zc}}$ is a pure function, and Γ_{zc} is a function with oracle access to public and private state variables. More tricky is showing that $\mathsf{dep}_{\mathsf{zc}}$ satisfies its requirements. Transcripts generated by $\mathsf{run}\text{-}\Gamma$ fall into three categories: They set a private key (initialisation), they insert a coin (minting), or they remove a coin, and insert some number of coins (sending).

Consider first a new initialisation transaction. It does not affect the behaviour of unconfirmed minting and sending transactions, as these do not use the current private state's secret key. Further, it cannot co-exist with another unconfirmed initialisation transaction, as this would initialise the private keys, ensuring an abort, which violates the preconditions of dependencies.

If the new transaction is a minting or balance transaction, this happily functions independently of other transactions, not having any requirements on the current private state. Likewise for sending transactions, the state transcript itself only depends on ρ^o and ρ^π, not the dynamic ρ. Specifically, the only thing varying there is which coins get added and removed from $\rho.\vec{C}$, but this information is not directly used – its only purpose is to reduce the necessary re-computation the next time around. □

UC Emulation

Theorem 2. *$\mathcal{F}_{\mathsf{sc}}^{\Delta_{\mathsf{pp}},\Lambda_{\mathsf{pp}}}$ is UC-emulated by $\mathcal{F}_{\mathsf{sc}}^{\Delta_{\mathsf{zc}},\Lambda_{\mathsf{zc}}}$ in presence of $\mathcal{G}_{\mathsf{simpleLedger}}$.*

This proof can also be carried out via invariants. Here the invariant tracking is simple: The real and ideal world have the same coins owned by the same users at any time. Our simulator, described in Subsection C.3, has a lot of bookkeeping to do, mostly to conjure up fake commitments and encryptions for the real-world adversary, and replicating them in the real world. We provide a full proof sketch in Appendix E.

Corollary 1. *$\mathcal{F}_{\mathsf{sc}}^{\Delta_{\mathsf{pp}},\Lambda_{\mathsf{pp}}}$ is UC-emulated by* Kachina*, parameterised by Γ_{zc}, $\mathsf{dep}_{\mathsf{zc}}$, and $\mathsf{desc}_{\mathsf{zc}}$, in the $\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}$-hybrid world, in the presence of $\mathcal{G}_{\mathsf{simpleLedger}}$.*

6 Conclusion

We have shown in this paper how to build a large class of smart contracts with only zero-knowledge and distributed ledgers, and outline how this can be used and extended upon. To do so we have modelled formally what smart contracts with privacy are, represented as a state transition function that is fed inputs from a ledger, and a leakage function that decides what parts of the input are visible on this ledger. We have then defined which class of such contracts we will consider in this paper, and presented a protocol, Kachina, which utilises non-interactive zero-knowledge proofs and state oracles to achieve the desired smart contract behaviour, while leaking only part of the computation performed.

While the designs are largely theoretical and detached from any actual implementation, we stress that they were designed with real-life constraints in mind: The use of state oracles allows moving most computationally hard, or storage intensive operations outside of the NIZK itself, reducing their cost. While the NIZK must still be universal, zero-knowledge constructions with universal reference strings exist [24], and are practical to use in this setting, although they are not directly provable in the UC model.

The concrete semantics of programming a smart contract, as well as the corresponding oracles, have been deliberately left unspecified, as many candidates are possible, and a well suited language may be domain specific. That said, most existing (public) smart contracts are small and simple in size, and would translate and run well even within a zero knowledge proof, if compiled into appropriate constraints.

In ending this paper, we would like to make clear that this problem space is by no means solved. We have shown how to realise a specific class of privacy-preserving smart contracts, however privacy is not such a simple issue to be addressed by a single paper. In Appendix I, we sketch the relation of trust models with privacy, and we believe this taxonomy of trust, and how each level can be addressed, formalised, and brought into a unified model, is a crucial long-term research question for providing meaningful privacy to users of smart contract systems.

References

1. Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 913–930. ACM Press, October 2018.
2. Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 324–356. Springer, Heidelberg, August 2017.
3. 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 Computer Society Press, May 2014.
4. Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. *IACR Cryptology ePrint Archive*, 2018:962, 2018.
5. 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 William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 991–1008. USENIX Association, 2018.
6. Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. Cryptology ePrint Archive, Report 2019/191, 2019. `https://eprint.iacr.org/2019/191`.
7. Jan Camenisch, Robert R. Enderlein, Stephan Krenn, Ralf Küsters, and Daniel Rausch. Universal composition with responsive environments. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 807–840. Springer, Heidelberg, December 2016.
8. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.
9. Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 61–85. Springer, Heidelberg, February 2007.
10. Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *34th ACM STOC*, 2002.
11. Manuel Chakravarty, Roman Kireev, Kenneth MacKenzie, Vanessa McHale, Jann Müller, Alexander Nemish, Chad Nester, Michael Peyton Jones, Simon Thompsona, Rebecca Valentine, and Philip Wadler. Functional blockchain contracts. `https://iohk.io/research/papers/#functional-blockchain-contracts`, 2019.
12. Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual payment hubs over cryptocurrencies. In *2019 IEEE Symposium on Security and Privacy*, 2019.
13. Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. General state channel networks. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 949–966. ACM Press, October 2018.
14. The Enigma Project Team. What is Enigma? `https://enigma.co/discovery-documentation/`, 2019.

15. Etherscan. Ethereum sync (default) chart. https://etherscan.io/chartsync/chaindefault, 2019.
16. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.
17. Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *17th ACM STOC*, pages 291–304. ACM Press, May 1985.
18. Harry A. Kalodner, Miles Carlsten, Paul Ellenbogen, Joseph Bonneau, and Arvind Narayanan. An empirical study of namecoin and lessons for decentralized namespace design. In *14th Annual Workshop on the Economics of Information Security, WEIS 2015, Delft, The Netherlands, 22-23 June, 2015*, 2015.
19. Harry A. Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 1353–1370. USENIX Association, August 2018.
20. Thomas Kerber, Markulf Kohlweiss, Aggelos Kiayias, and Vassilis Zikas. Ouroboros crypsinous: Privacy-preserving proof-of-stake. In *2019 IEEE Symposium on Security and Privacy*, 2019.
21. Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. Fair and robust multi-party computation using a global transaction ledger. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 705–734. Springer, Heidelberg, May 2016.
22. Ahmed E. 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*, pages 839–858. IEEE Computer Society Press, May 2016.
23. Kevin Liao, Matthew A. Hammer, and Andrew Miller. Ilc: A calculus for composable, computational cryptography. Cryptology ePrint Archive, Report 2019/402, 2019. https://eprint.iacr.org/2019/402.
24. Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updateable structured reference strings. Cryptology ePrint Archive, Report 2019/099, 2019. https://eprint.iacr.org/2019/099.
25. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.
26. Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.
27. Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.
28. Fabian Vogelsteller and Vitalik Buterin. ERC-20 token standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md, 2015.
29. Jonathan Warren. Bitmessage: A peer-to-peer message authentication and delivery system. *white paper (27 November 2012), https://bitmessage. org/bitmessage. pdf*, 2012.
30. Jeffrey Wilcke. The Ethereum network is currently undergoing a DoS attack. https://blog.ethereum.org/2016/09/22/ethereum-network-currently-undergoing-dos-attack/, September 2016.
31. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014.
32. Guy Zyskind, Oz Nathan, and Alex Pentland. Enigma: Decentralized Computation Platform with Guaranteed Privacy. *arXiv e-prints*, June 2015.

A UC Conventions

This paper is modelled and proven in the Universal Composability Framework [8], with global functionalities [9]. Although we assume a basic familiarity with these concepts from the reader, the style of writing UC protocols and functionalities may differ greatly from author to author. As a result potentially important corner cases may be overlooked, as the exact behaviour of a given functionality is sometimes unclear. We adopt a more explicit style, while at the same time attempting to avoid writing unnecessary information in the definition of the functionalities. While the proofs, protocols and functionalities can be read and understood without explicit knowledge of the notation described in this section, we define what behaviour our notation leaves implicit in this section.

In this section we describe the implicit behaviour of the functionalities and protocols presented in this paper, as well as touching on miscellaneous conventions adopted to increase the formal clarity of the model. Further, we introduce some notations used throughout the paper that simplify our code but are unrelated to UC.

Flow of execution. Session identifiers are formally used in UC to shield a protocol from external calls, except when allowed by the control function. While they are effectively a technical detail of the description in UC, they are often replicated in the description of functionalities and protocols. We leave all session identifiers implicit instead. In a similar vein, it is often a convention to replicate (part of) the input to a functionality when returning the result, to ensure that it is clear which query is being answered. We omit this as well, in favour of simply stating the actual value returned. Both of these are how a protocol would be written in a channel-based communications model, such as that of [23], rather than the tape-based model of UC itself.

When a functionality is processing something, it is always processing *on behalf of some party*, which may be the adversary itself, or may be corrupted. Likewise when a protocol is processing something, it is processing this *on behalf of its owning party*. When a functionality or protocol hands off execution to another entity, by making a query to another functionality, or the adversary, execution *for this party* is suspended, and resumes only when the query returns. Attempts by the environment to make queries to a suspended protocol will be ignored. Likewise, if the environment attempts to query a functionality with a party which is currently suspended, the query will also be ignored. Crucially, the environment may still query a functionality with *another party* while one is suspended, ensuring that parties may still act concurrently. We observe that this behaves equally in protocols and functionalities, as the functionality is suspended in the same situation is the corresponding party's protocol is suspended. Finally, we assume that queries will eventually return – this is equivalent to queries which do not return, as we allow the environment and adversary to hold off indefinitely until returning. While this is possible, in practice, due to the implicit suspension mechanism described above, this means disabling a party permanently.

This above mechanism is not a great deviation from UC – it can easily be implemented by having a functionality or protocol record locally the suspension, and reject new queries from the suspended party until it receives an input of a specified form. We simply omit this mechanism when writing our protocols. Something similar is in fact necessary for our security, as parties could otherwise easily shoot themselves in the foot by concurrently creating conflicting transactions. Responsive environments [7] are a strictly stronger form of this idea.

We assume the existence of a set of all parties $\mathcal{P}$, of which there is a subset of honest parties $\mathcal{H} \subseteq \mathcal{P}$. We assume $\mathcal{H} \neq \varnothing$. Correspondingly, the set of corrupted parties is $\mathcal{P} \setminus \mathcal{H}$. In this paper, static corruption is assumed, and all functionalities are assumed to have knowledge of these sets.

As a slight note, we use a somewhat unconventional model of having multiple functionalities interacting with each other in the ideal world. This is largely to avoid monolithic functionality design, using composition as a software design primitive, rather than a security one.

Notation. In terms of notation, we will explicitly declare and initialize all state variables of functionalities and protocols, to make formal statements about them more precise. For the same reason, the behaviour of functionalities and protocols is described via detailed pseudocode, instead of text. Most of the notation is self-explanatory, however adversarial queries are not. As the adversary may respond arbitrarily to queries, we include with each query a well-formedness condition, and a fallback distribution. In particular, we write **query** $\mathcal{A}$ **with** x and **receive the reply** y, **satisfying** P, else **sampling from** D to mean the following: Send x to $\mathcal{A}$, then wait for the response y. If $P(y)$ does not hold, instead randomly sample y from D. This allows us to ensure responses are well-formed, while avoiding the common technique of aborting in the ideal world on receiving unexpected input, something we try to avoid, as it effectively permits denial-of-service in the "ideal" world. Finally, we use the period (".") as a membership access operator, to talk about variables of simulated functionalities, or in the proof, to talk about state variables of various functionalities and protocol instances. For instance, we write $\mathcal{F}.X$ to mean the state variable X within the (possibly simulated) functionality $\mathcal{F}$.

Functional Programming Constructs. Besides understanding the message passing mechanics of UC, we assume only the basic programming knowledge needed to read pseudo-code. We sometimes use functional programming expressions, including the following for precision:

- Lambda expressions: $(\lambda x : 2x)(2) = 4$
- List and tuple literals: $[1, 2]$, and $(1, 2)$.
- The higher-order function map: $\mathsf{map}(\lambda x : 2x, [1, 2]) = [2, 4]$
- The higher-order function filter: $\mathsf{filter}(\lambda x : x \equiv 0 \mod 2, [1, 2]) = [2]$
- The tuple projection function proj_i: $\mathsf{proj}_1((a, b)) = a$

We interpret maps as functions from keys to values. The symbols $\perp$ and $\varnothing$ are overloaded, with the former representing both "false", and "error/abort",

while the latter represents the empty set, empty map, and in the case of contract states, the initial state (which the contract itself may ascribe a different format to). Further, for a map M, we will write $k \in M$ to mean that the map contains the key k. A key is not in the map iff $M(k) := \bot$. For lists, we use ϵ to denote the empty list, and $\|$ to denote list concatenation. Single non-list items can be interpreted as a singleton list.

We will use the following functions throughout the paper. $\mathsf{prefix}(L, x)$ returns the longest prefix of L containing x, or L itself, if not such prefix exists. $\mathsf{idx}(L, x)$ returns the index of the first occurrence of x in L, or $\bot$ if $x \notin L$.

```
function prefix(L, x)
    let L' ← ϵ
    for y ∈ L do
        L' ← L' ‖ y
        if y = x then break
    return L'
function idx(L, x)
    let i ← 0
    for y ∈ L do
        if y = x then return i
        let i ← i + 1
    return ⊥
```

Finally, we write $a \prec b$ for the list prefix operation, where we assume reflexivity.

B Ledger Functionalities

As smart contracts utilize an inderection layer of an underlying ledger, for completeness we will define the behaviour of this underlying ledger. The key characteristics of the ledgers used here are that they maintain a sequence of transactions, which any user may submit new transactions to, which *may* then eventually appear in this sequence. The sequence itself is public, and can be read by anyone.

We note in particular that for the purposes of this paper, many common features of ledgers are not required – in particular liveness is optional, although the ideal-world guarantees for a ledger without liveness are naturally lessened. Further, we do not specify a validation predicate, instead performing the validation in our definition of smart contracts. This is solely to have a clean separation between the consensus system and the semantics of transactions – to prevent denial-of-service attacks in a real system, this line would need to be blurred, and transaction validation partially done in the consensus algorithm.

One feature typical ledger functionalities do not have, which we do explicitly use here, is that of transactional privacy. Typically ledgers leak which party created a new transaction, something we wish to avoid to strengthen the privacy of our protocols. It is worth noting that this leakage is entirely network based, and using a sender-anonymous network is sufficient to adapt "leaky" ledgers to this more private variant.

Finally, readers may be surprised that the "private ledger" of [20] is not used. This is largely for the same reason that validation predicates are avoided – the separation of the ledger and transaction semantics is not possible in this private ledger. While it would be possible to encode ideal smart contracts as specific ledger blinding functions, this does not accurately model what we intend to capture, and results in a large, monolithic, and hard-to-understand functionality – decidedly not ideal.

B.1 The Perfect Ledger

The perfect ledger is a Platonic ideal form of a distributed ledger – it allows any party to instantly append to its record, and allows any party to read the current sequence of recorded transactions. This ledger is very similar to that used in [21], however it is not realised by existing distributed ledger protocols – not least as there is by necessity some network delay.

Functionality $\mathcal{G}_{\mathsf{perfectLedger}}$

The perfect ledger is strictly *more* powerful than actual ledger implementations.

State variables and initialisation values:

Variable	Description
$\Sigma := \epsilon$	Authoritative ledger state

When receiving a message $(\textsc{submit}, \tau)$ *from a party* p*:*
  let $\Sigma \leftarrow \Sigma \parallel \tau$
When receiving a message READ *from a party* p*:*
  return Σ

B.2 The Simplified Practical Ledger

The simplified ledger captures the essence of the traditional persistence property of ledgers, although it does not capture liveness. Any user may post transactions, which are deemed unconfirmed. The adversary may decide when and which unconfirmed transactions to move to an append-only ledger, and may decide how long a prefix of this ledger honest parties see – provided it does not remove anything previously revealed to them.

While the liveness property is not captured by this ledger, due to the large amount of adversarial control, it is straightforward to see – although we will not demonstrate it here – that more complex ledgers, such as those defined in [2,1], UC-emulate $\mathcal{G}_{\mathsf{simpleLedger}}$. In particular, this means that if replaced in the ideal world with such a ledger, which *does* have the liveness property, we also in practice have liveness for our protocol. We discuss the issue of liveness more in Appendix F.

Functionality $\mathcal{G}_{\mathsf{simpleLedger}}$

The simplified interface to $\mathcal{G}_{\mathsf{ledger}}$ is strictly less powerful than actual ledger implementations, allowing reasoning about a less complex ledger functionality.

State variables and initialisation values:

Variable	Description
$\Sigma := \epsilon$	Authoritative ledger state
$M := \lambda p.\epsilon$	Mapping of parties to ledger states

When receiving a message (SUBMIT, τ) *from a party* p*:*

```
// The adversary is not required to ever put
// transactions on the ledger.
// Where it doesn't, the execution is unlikely
// to be interesting, however.
```

query $\mathcal{A}$ **with** $(\text{TRANSACTION}, \tau)$

When receiving a message READ *from a party* p*:*

if $p = \mathcal{A}$ **then return** Σ
else return $M(p)$

When receiving a message (EXTEND, Σ') *from* $\mathcal{A}$*:*

let $\Sigma \leftarrow \Sigma \parallel \Sigma'$

When receiving a message $(\text{ADVANCE}, p, \Sigma')$ *from* $\mathcal{A}$*:*

if $M(p) \prec \Sigma' \prec \Sigma$ **then let** $M(p) \leftarrow \Sigma'$.

C Fully Specified Functionalities and Protocols

C.1 Ideal World

Functionality $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$

The smart contract functionality $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$ allows parties to query a deterministic state machine determined by Δ and Λ in a ledger-specified order. The exact semantics of the call are subject to adversarial influence, who is provided some leakage, as defined in Λ.

State variables and initialisation values:

Variable	Description
$T := \varnothing$	Mapping from transactions to their executing components.
$U_p := \epsilon$	Sequence of unconfirmed transactions, for all parties p

When receiving a message $(\text{POST-QUERY}, w)$ *from an honest party* p*:*

let $\Sigma_p \leftarrow \mathsf{updateState}(p)$
let $(\mathsf{desc}, \mathsf{lkg}, D, z) \leftarrow \Lambda(|\Sigma_p|, U_p, \mathsf{filter}(\lambda(\tau, \cdot) : \tau \in U_p, T), \mathsf{execState}(\Sigma_p), p, w)$
if $\mathsf{desc} = \bot$ **then**

return REJECTED
send (LEAK, desc) **to** p and **receive the reply** b
if b **then**
query $\mathcal{A}$ **with** (TRANSACTION, lkg, D) and **receive the reply** (τ, a), **satisfying** $T(\tau) = \bot \wedge \tau \neq \bot$, else **sampling from** $(\{0,1\}^\kappa, \bot)$
let $T(\tau) \leftarrow (p, w, z, a, D); U_p \leftarrow U_p \parallel \tau$
send (SUBMIT, τ) **to** $\mathcal{G}_{\mathsf{ledger}}$ **on behalf of** p
return (POSTED, τ)
else
return REJECTED

When receiving a message (CHECK-QUERY, τ) *from an honest party* p*:*

let $\Sigma_p \leftarrow \mathsf{updateState}(p)$
if $\tau \in \Sigma_p$ **then**
if $T(\tau) = (p, \ldots)$ **then return** $\mathsf{execResult}(\mathsf{prefix}(\Sigma_p, \tau))$
else return $\bot$
else return NOT-FOUND

Helper procedures:

procedure $\mathsf{updateState}(p)$
send READ **to** $\mathcal{G}_{\mathsf{ledger}}$ **through** p and **receive the reply** Σ_p
let $C \leftarrow \mathsf{execConfirmed}(\Sigma_p)$
let $U_p' \leftarrow U_p$
repeat
let $U_p \leftarrow U_p'$
for $\tau \in U_p$ **do**
let $(\ldots, D) \leftarrow T(\tau)$
if $D \not\subseteq (C \cup U_p) \vee (D \cap C) \not\sqsubseteq \Sigma_p$ **then let** $U_p' \leftarrow U_p' \setminus \{\tau\}$
until $U_p = U_p'$
return Σ_p
procedure $\mathsf{execState}(\Sigma) =$ **let** $(\sigma, \cdot, \cdot) \leftarrow \mathsf{exec}(\Sigma)$ **in return** σ
procedure $\mathsf{execResult}(\Sigma) =$ **let** $(\cdot, y, \cdot) \leftarrow \mathsf{exec}(\Sigma)$ **in return** y
procedure $\mathsf{execConfirmed}(\Sigma) =$ **let** $(\cdot, \cdot, C) \leftarrow \mathsf{exec}(\Sigma)$ **in return** C
procedure $\mathsf{exec}(\Sigma)$
let $\sigma \leftarrow \varnothing; y \leftarrow \bot; C \leftarrow \varnothing$
for $\tau \in \Sigma$ **do**
if $\tau \in C$ **then continue**
if $T(\tau) = \bot$ **then**
query $\mathcal{A}$ **with** (INPUT, τ) and **receive the reply** $x = (p, w, z, a, D)$, **satisfying** $p \notin \mathcal{H}$, else **sampling from** {NONE}
if $T(\tau) = \bot$ **then**
let $T(\tau) \leftarrow x$
$y \leftarrow \bot$
if $T(\tau) =$ NONE **then continue**
let $(p, w, z, a, D) \leftarrow T(\tau)$
if $D \setminus C \neq \varnothing \vee D \not\sqsubseteq \Sigma$ **then continue**
let $(\sigma', y) \leftarrow \Delta(\sigma, p, w, z, a)$

if $\sigma' \neq \bot$ **then let** $\sigma \leftarrow \sigma'; C \leftarrow C \cup \{\tau\}$
return (σ, y, C)

C.2 KACHINA

Transition Function Δ_{KACHINA}

The KACHINA transition function, running an internal transition function Γ with oracle access to the public contract state, and the private state of the party making the query. The query has an associated context z, which the private state oracle may access, and an associated public state transcript $\mathcal{T}_\sigma$, which must be consistent with the current public state in order for the query to run successfully.

When receiving an input $((\sigma, \boldsymbol{\rho}), p, w, (\mathcal{T}_\sigma, z), \cdot)$*:*
if $\mathcal{T}_\sigma(\sigma) = \bot$ **then return** $(\bot, \bot)$
let $(\sigma, \cdot, \boldsymbol{\rho}[p], \cdot, y) \leftarrow \mathsf{run\text{-}\Gamma}(\sigma, \boldsymbol{\rho}[p], w, z, p \in \mathcal{H})$
return $((\sigma, \boldsymbol{\rho}), y)$

Helper procedures:
function $\mathsf{run\text{-}\Gamma}(\sigma, \rho, w, z, h)$
$\mathcal{O}_\sigma \leftarrow \mathcal{U}(\sigma, \varnothing); \mathcal{O}_\rho \leftarrow \mathcal{U}(\rho, z)$
if $\neg h$ **then let** $\mathcal{O}_\rho \leftarrow z$
$y \leftarrow \Gamma_{\mathcal{O}_\sigma, \mathcal{O}_\rho}(w)$
$(\sigma, \mathcal{T}_\sigma) \leftarrow \mathsf{state}(\mathcal{O}_\sigma); (\rho, \mathcal{T}_\rho) \leftarrow \mathsf{state}(\mathcal{O}_\rho)$
return $(\sigma, \mathcal{T}_\sigma, \rho, \mathcal{T}_\rho, y)$

Leakage Function Λ_{KACHINA}

The KACHINA leakage function reveals the public state transcript generated by Γ during the projected transition. This projected transition takes the state of the contract as the party currently sees it, and first replays all currently unconfirmed transactions from the same party. Both the initial (latest confirmed) contract state, as well as the projected state, and a randomness stream are considered the transaction's context.

When receiving an input $(t, U, T, (\sigma^o, \boldsymbol{\rho}^o), p, w)$*:*
let $(\sigma^\pi, \rho^\pi) \leftarrow (\sigma^o, \boldsymbol{\rho}^o[p])$
for $u \in U$ **do**
let $(p', w', (\mathcal{T}_\sigma, z), \cdot, \cdot, D) \leftarrow T(u)$
if $\mathcal{T}_\sigma(\sigma^\pi) = \bot$ **then**
return $(\bot, \bot, \bot, \bot, \bot)$
let $(\sigma^\pi, \cdot, \rho^\pi, \mathcal{T}, \cdot) \leftarrow \mathsf{run\text{-}\Gamma}(\sigma^\pi, \rho^\pi, w', z, p' \in \mathcal{H})$
let $X \leftarrow X \parallel (u, \mathcal{T}, z, D)$
let η be a randomness stream.
let $z \leftarrow (\sigma^o, \boldsymbol{\rho}^o[p], \sigma^\pi, \rho^\pi, \eta)$

let $(\sigma, \mathcal{T}_\sigma, \rho, \mathcal{T}_\rho, \cdot) \leftarrow \mathsf{run}\text{-}\Gamma(\sigma^\pi, \rho^\pi, w, z, \top)$
if $\sigma = \bot \vee \rho = \bot$ **then**
  return $(\bot, \bot, \bot, \bot, \bot)$
else
  let $D \leftarrow \mathsf{dep}(X, \mathcal{T}_\rho, z)$
  return $(\mathsf{desc}(t, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, w, z), \mathcal{T}_\sigma, D, (\mathcal{T}_\sigma, z))$

Protocol KACHINA

The KACHINA protocol realises the ideal smart contract functionality when parameterised by a transition function Γ, a leakage descriptor desc, and a dependency function dep, such that the corresponding (Δ, Λ) pair is in $\mathbb{C}_{\text{KACHINA}}$. It operates in the $(\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}, \mathcal{G}_{\mathsf{simpleLedger}})$-hybrid model, where $\mathcal{L}$ is defined below.

$((\mathcal{T}_\sigma, \cdot), (w, \mathcal{T}_\rho)) \in \mathcal{L}$ if and only if, where $\mathcal{O}_\sigma \leftarrow \mathcal{O}(\mathcal{T}_\sigma)$, and $\mathcal{O}_\rho \leftarrow \mathcal{O}(\mathcal{T}_\rho)$, $\Gamma_{\mathcal{O}_\sigma, \mathcal{O}_\rho}(w) \neq \bot$, and after it is run, $\mathsf{consumed}(\mathcal{O}_\sigma) \wedge \mathsf{consumed}(\mathcal{O}_\rho)$ holds.

State variables and initialisation values:

Variable	Description
$T := \varnothing$	Mapping from transactions to their private state transcripts and contexts.
$Y := \varnothing$	Mapping from transactions to their outputs.
$U := \epsilon$	Sequence of unconfirmed transactions.

When receiving a message $(\text{POST-QUERY}, w)$ *from a party* p*:*

let $\Sigma \leftarrow \mathsf{updateState}()$
let $(\sigma^o, \rho^o) \leftarrow \mathsf{execState}(\Sigma)$
let $\sigma^\pi \leftarrow \sigma^o; \rho^\pi \leftarrow \rho^o; X \leftarrow \epsilon$
for $u = (\mathcal{T}_\sigma, D, \cdot) \in U$ **do**
  let $(\mathcal{T}_\rho, z) \leftarrow T(u)$
  let $\sigma^\pi \leftarrow \mathcal{T}_\sigma(\sigma^\pi); \rho^\pi \leftarrow \mathcal{T}_\rho(\rho^\pi, z)$
  let $X \leftarrow X \parallel (u, \mathcal{T}_\rho, z, D)$
let η be a randomness stream.
let $z \leftarrow (\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta)$
let $(\sigma, \mathcal{T}_\sigma, \rho, \mathcal{T}_\rho, y) \leftarrow \mathsf{run}\text{-}\Gamma(\sigma^\pi, \rho^\pi, w, z)$
if $\sigma = \bot \vee \rho = \bot$ **then**
  return REJECTED
let $D \leftarrow \mathsf{dep}(X, \mathcal{T}_\rho, z)$
send $(\text{LEAK}, \mathsf{desc}(|\Sigma|, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, w, z))$ **to** p and **receive the reply** b
if b **then**
  send $(\text{PROVE}, (\mathcal{T}_\sigma, D), (w, \mathcal{T}_\rho))$ **to** $\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}$ and **receive the reply** π
  let $\tau \leftarrow (\mathcal{T}_\sigma, D, \pi)$
  let $T(\tau) \leftarrow (\mathcal{T}_\rho, z); Y(\tau) \leftarrow y; U \leftarrow U \parallel \tau$
  send (SUBMIT, τ) **to** $\mathcal{G}_{\mathsf{simpleLedger}}$
  return (POSTED, τ)
else
  return REJECTED

When receiving a message $(\text{CHECK-QUERY}, \tau)$ *from a party* p*:*

let $\Sigma \leftarrow \mathsf{updateState}()$
if $\tau \in \Sigma$ **then return** $\mathsf{execResult}(\mathsf{prefix}(\Sigma, \tau))$
else return NOT-FOUND

Helper procedures:

procedure $\mathsf{updateState}(p)$
  send READ **to** $\mathcal{G}_{\mathsf{simpleLedger}}$ and **receive the reply** Σ
  let $C \leftarrow \mathsf{execConfirmed}(\Sigma)$
  let $U' \leftarrow U$
  repeat
    let $U \leftarrow U'$
    for $\tau = (\cdot, D, \cdot) \in U$ **do**
      if $D \nsubseteq (C \cup U) \vee (D \cap C) \not\sqsubseteq \Sigma$ **then let** $U' \leftarrow U' \setminus \{\tau\}$
  until $U = U'$
  return Σ
procedure $\mathsf{execState}(\Sigma) =$ **let** $(\sigma, \cdot, \cdot) \leftarrow \mathsf{exec}(\Sigma)$ **in return** σ
procedure $\mathsf{execResult}(\Sigma) =$ **let** $(\cdot, y, \cdot) \leftarrow \mathsf{exec}(\Sigma)$ **in return** y
procedure $\mathsf{execConfirmed}(\Sigma) =$ **let** $(\cdot, \cdot, C) \leftarrow \mathsf{exec}(\Sigma)$ **in return** C
procedure $\mathsf{exec}(\Sigma)$
  let $\sigma \leftarrow \varnothing; \rho \leftarrow \varnothing; y \leftarrow \bot; C \leftarrow \varnothing$
  for $\tau = (\mathcal{T}_\sigma, D, \pi) \in \Sigma$ **do**
    if $\tau \in C$ **then continue**
    let $y \leftarrow \bot$
    send $(\text{VERIFY}, (\mathcal{T}_\sigma, D), \pi)$ **to** $\mathcal{F}^{\mathcal{L}}_{\mathsf{nizk}}$ and **receive the reply** b
    if $\neg b$ **then continue**
    if $D \setminus C \neq \varnothing \vee D \not\sqsubseteq \Sigma$ **then continue**
    if $\mathcal{T}_\sigma(\sigma) \neq \bot$ **then**
      let $\sigma \leftarrow \mathcal{T}_\sigma(\sigma); C \leftarrow C \cup \{\tau\}$
      if $T(\tau) \neq \bot$ **then**
        let $(\mathcal{T}, z) \leftarrow T(\tau)$
        let $\rho \leftarrow \mathcal{T}(\rho, z)$
        let $y \leftarrow Y(\tau)$
  return $((\sigma, \rho), y, C)$

C.3 Private Payments

Transition Function Δ_{pp}

The state transition function for a private payments system. Parties have associated public keys, and balances. The payments system allows for parties without a public key to generate one, and for parties to transfer and mint single-denomination coins, as well as query their own balance.

State variables and initialisation values:

Variable	Description
$K := \varnothing$	Mapping of parties to public keys
$B := \lambda p : 0$	Mapping of parties to their spendable coins

When receiving an input $(\sigma, p, \text{INIT}, \cdot, \mathsf{pk})$*:*

if $\sigma.K(p) = \bot$ **then**
  while $\exists p' : \mathsf{pk} = \sigma.K(p') \vee \mathsf{pk} \in \{\varnothing, \bot\}$ **do let** $\mathsf{pk} \xleftarrow{R} \{0,1\}^\kappa$
  let $\sigma.K(p) \leftarrow \mathsf{pk}$
  return (σ, pk)
else
  return $(\bot, \bot)$

When receiving an input $(\sigma, p, (\text{SEND}, \mathsf{pk}), \cdot, a)$*:*

if $p \notin \mathcal{H} \wedge a \neq \varnothing$ **then**
  let $\mathsf{pk}' \leftarrow a$
  assert $\nexists p' \in \mathcal{H} : \mathsf{pk}' = \sigma.K(p')$
else if $\sigma.K(p) \neq \bot$ **then let** $\mathsf{pk}' \leftarrow \sigma.K(p)$
else return $(\bot, \bot)$
if $\sigma.B(\mathsf{pk}') > 0$ **then**
  let $\sigma.B(\mathsf{pk}') \leftarrow \sigma.B(\mathsf{pk}') - 1$
  let $\sigma.B(\mathsf{pk}) \leftarrow \sigma.B(\mathsf{pk}) + 1$
  return $(\sigma, \top)$
else return $(\bot, \bot)$

When receiving an input $(\sigma, p, \text{MINT}, \mathsf{pk}, \cdot)$*:*

let $\sigma.B(\mathsf{pk}) \leftarrow \sigma.B(\mathsf{pk}) + 1$
return $(\sigma, \top)$

When receiving an input $(\sigma, p, \text{BALANCE}, B, \cdot)$*:*

return (σ, B)

Leakage Function Λ_{pp}

Each operation on Δ_{pp} has minimal leakage, revealing only which operation was performed, and in the case of a transfer, the time and the recipient – if and only if the recipient is corrupted.

When receiving an input (t, U, T, σ, p, w)*:*

let $\sigma^\pi \leftarrow \sigma$
let $B^- \leftarrow 0$
for $u \in U$ **do**
  let $(\cdot, w', z, a, \cdot, \cdot) \leftarrow T(u)$
  if $w' = (\text{SEND}, \cdot)$ **then let** $B^- \leftarrow B^- + 1$
  let $(\sigma^\pi, \cdot) \leftarrow \Delta_{\mathsf{pp}}(\sigma^\pi, p, w', z, a)$
if $w = \text{INIT}$ **then**
  if $\sigma.K(p) = \sigma^\pi.K(p) = \bot$ **then**
    return $(\text{INIT}, \text{INIT}, \epsilon, \varnothing)$
  else return $(\bot, \bot, \bot, \bot)$

else if $\exists \mathsf{pk} : w = (\text{SEND}, \mathsf{pk})$ **then**
  let $c \xleftarrow{R} \{0,1\}^\kappa$
  if $\sigma.B(\sigma.K(p)) - B^- > 0 \wedge \sigma.K(p) = \sigma^\pi(p) \neq \bot$ **then**
    let $\mathsf{lkg} \leftarrow t$
    if $\nexists p' \in \mathcal{H} : \mathsf{pk} = \sigma.K(p')$ **then let** $\mathsf{lkg} \leftarrow (t, \mathsf{pk})$
    return $((\text{SEND}, t, \mathsf{pk}), \mathsf{lkg}, \epsilon, \varnothing)$
  else return $(\bot, \bot, \bot, \bot)$
else if $w = \text{MINT} \wedge \sigma.K(p) \neq \bot$ **then**
  return $(\text{MINT}, \text{MINT}, \epsilon, \sigma.K(p))$
else if $w = \text{BALANCE} \wedge \sigma.K(p) \neq \bot$ **then**
  return $(\text{BALANCE}, \text{BALANCE}, \epsilon, \sigma.B(\sigma.K(p)) - B^-)$
else
  return $(\bot, \bot, \bot, \bot)$

Transition Function Γ_{zc}

The state transition function for a Zerocash-based token contract. In green are parts run in the public state oracle, in red are parts run in the private state oracle.

Public state variables and initialisation values:

Variable	Description
$\mathsf{cms} := \varnothing$	Public coin commitment set
$\mathsf{sns} := \varnothing$	Public serial number set
$\vec{R} := \epsilon$	Vector of commitment Merkle tree roots
$\vec{M} := \epsilon$	Vector of encrypted messages

Private state variables and initialisation values:

Variable	Description
$i := 0$	Index of $\vec{M}$ processed.
$\vec{C} := \epsilon$	Vector of coins available.
$K_e := \bot$	Encryption secret key.
$K_z := \bot$	Zero-knowledge secret key.

When receiving an input INIT*:*
  send INIT **to** $\mathcal{O}_\rho$ and **receive the reply** pk
  return pk
When receiving an input $(\text{SEND}, (\mathsf{pk}_z, \mathsf{pk}_e))$*:*
  send $(\text{SEND}, \mathsf{pk}_e)$ **to** $\mathcal{O}_\rho$ and **receive the reply** $(p, r, K_z, p', r', \mathsf{rt}, \mathsf{path}, M)$
  assert path is a valid Merkle tree path with root rt, to the element
    $\mathsf{comm}_r((\mathsf{prf}^{\mathsf{pk}}_{K_z}(1), p))$
  let $\mathsf{sn} \leftarrow \mathsf{prf}^{\mathsf{sn}}_{K_z}((p, r))$
  let $\mathsf{cm} \leftarrow \mathsf{comm}_{r'}(pk_z, p')$
  send $(\text{SPEND}, \mathsf{sn}, \mathsf{rt})$ **to** $\mathcal{O}_\sigma$
  send (MSG, M) **to** $\mathcal{O}_\sigma$

send $(\textsc{mint}, \mathsf{cm})$ **to** $\mathcal{O}_\sigma$
return $\top$

When receiving an input MINT*:*
send MINT **to** $\mathcal{O}_\rho$ and **receive the reply** cm
send $(\textsc{mint}, \mathsf{cm})$ **to** $\mathcal{O}_\sigma$
return $\top$

When receiving an input BALANCE*:*
send BALANCE **to** $\mathcal{O}_\rho$ and **receive the reply** B
return B

When receiving an private oracle query INIT*:*
assert $\rho^\pi.K_e = \bot \wedge \rho^\pi.K_z = \bot$
let $\rho.K_z \xleftarrow{R} \{0,1\}^\kappa$
let $(\rho.K_e, \mathsf{pk}_e) \leftarrow \mathsf{keyGen}(1^\kappa)$
return $(\mathsf{prf}^{\mathsf{pk}}_{\mathsf{sk}_z}(1), \mathsf{pk}_e)$

When receiving an private oracle query $(\textsc{send}, \mathsf{pk}_e)$*:*
let $\rho^o \leftarrow \mathsf{update}(\rho^o, \sigma^o)$
let $\rho^\pi \leftarrow \mathsf{update}(\rho^\pi, \sigma^\pi)$
let $\rho \leftarrow \mathsf{update}(\rho, \sigma^\pi)$
assert $(\rho^o.\vec{C} \cap \rho^\pi.\vec{C}) \neq \epsilon$
let $(p, r) \leftarrow (\rho^o.\vec{C} \cap \rho^\pi.\vec{C})[0]$
let $\rho.\vec{C} \leftarrow \rho.\vec{C} \setminus \{(p, r)\}$
let $\mathsf{rt} \leftarrow \mathsf{merkleroot}(\sigma^o.\mathsf{cms})$
let $\mathsf{path} \leftarrow \mathsf{merklepath}(\mathsf{comm}_r((\mathsf{prf}^{\mathsf{pk}}_{\rho^o.K_z}(1), p)), \mathsf{rt})$
let $(p', r') \xleftarrow{R} \{0,1\}^\kappa \times \{0,1\}^\kappa$
let $M \leftarrow \mathsf{enc}((r', p'), \mathsf{pk}_e)$
let $K_z \leftarrow \rho^o.K_z$
return $(p, r, \rho^o.K_z, p', r', \mathsf{rt}, \mathsf{path}, M)$

When receiving an private oracle query MINT*:*
assert $\rho^o.K_e \neq \bot \wedge \rho^o.K_z \neq \bot$
let $(p, r) \xleftarrow{R} \{0,1\}^\kappa \times \{0,1\}^\kappa$
let $\mathsf{cm} \leftarrow \mathsf{comm}_r(\mathsf{prf}^{\mathsf{pk}}_{\rho^o.K_z}(1), p)$
let $\rho.\vec{C} \leftarrow \rho.\vec{C} \parallel (p, r)$
return cm

When receiving an private oracle query BALANCE*:*
let $\rho^o \leftarrow \mathsf{update}(\rho^o, \sigma^o)$
let $\rho^\pi \leftarrow \mathsf{update}(\rho^\pi, \sigma^\pi)$
return $|\rho^o.\vec{C} \cap \rho^\pi.\vec{C}|$

When receiving an public oracle query $(\textsc{spend}, \mathsf{sn}, \mathsf{rt})$*:*
assert $\mathsf{sn} \notin \sigma.\mathsf{sns}$
assert $\mathsf{rt} \in \sigma.\vec{R}$
let $\sigma.\mathsf{sns} \leftarrow \sigma.\mathsf{sns} \cup \{\mathsf{sn}\}$

When receiving an public oracle query $(\textsc{msg}, M)$*:*

let $\sigma.\vec{M} \leftarrow \sigma.\vec{M} \parallel M$

When receiving an public oracle query (MINT, cm)*:*

let $\sigma.\mathsf{cms} \leftarrow \sigma.\mathsf{cms} \cup \{\mathsf{cm}\}$

let $\sigma.\vec{R} \leftarrow \sigma.\vec{R} \parallel \mathsf{merkleroot}(\sigma.\mathsf{cms})$

Helper procedures:

function $\mathsf{update}(\rho, \sigma)$

let $\vec{N} \leftarrow \sigma.\vec{M}[\rho.i{:}]; \rho.i \leftarrow \mathsf{max}(\rho.i, |\sigma.\vec{M}|)$

for $M \in \vec{N}$ **do**

if $\exists r, p : (r, p) = \mathsf{dec}(M, \rho.K_e)$ **then**

if $\mathsf{comm}_r((\mathsf{prf}^{\mathsf{pk}}_{\rho.K_z}(1), p) \notin \sigma.\mathsf{cms}$ **then continue**

if $\mathsf{prf}^{\mathsf{sn}}_{\rho.K_z}(p) \in \sigma.\mathsf{sns}$ **then continue**

let $\rho.\vec{C} \leftarrow \rho.\vec{C} \parallel (r, p)$

return ρ

Simulator $\mathcal{S}_{\mathsf{zc}}$

The fully detailed Zerocash simulator.

State variables and initialisation values:

Variable	Description
$B := \varnothing$	Unspent adversarial coins.
$K := \varnothing$	Honest public/private key pairs.
$T := \varnothing$	Mapping of transactions to created coin commitments.

When receiving a message (TRANSACTION, x, D) *from* $\mathcal{F}_{\mathsf{sc}}^{\Delta_{\mathsf{pp}}, \Lambda_{\mathsf{pp}}}$*:*

if $x =$ INIT **then**

let $\mathcal{T}_\sigma \leftarrow \epsilon$

query $\mathcal{A}$ **with** (TRANSACTION, $\mathcal{T}_\sigma, D$) and **receive the reply** $(\tau, \cdot)$, **satisfying** $T(\tau) = \bot \wedge \tau \neq \bot$, else **sampling from** $(\{0,1\}^\kappa, \bot)$

let $(K_e, \mathsf{pk}_e) \leftarrow \mathsf{keyGen}(1^\kappa)$

let $K_z \xleftarrow{R} \{0,1\}^\kappa$

let $\mathsf{pk}_z \leftarrow \mathsf{prf}^{\mathsf{pk}}_{K_z}(1)$

let $K \leftarrow K \cup \{((K_z, K_e), (\mathsf{pk}_z, \mathsf{pk}_e))\}$

let $T(\tau) \leftarrow \varnothing$

return $(\tau, (\mathsf{pk}_z, \mathsf{pk}_e))$

else if $x = (\text{SEND}, t, (\mathsf{pk}_z, \mathsf{pk}_e))$ **then**

let $p \xleftarrow{R} \{0,1\}^\kappa; r \xleftarrow{R} \{0,1\}^\kappa$

let $\mathsf{sn} \xleftarrow{R} \mathsf{prf}^{\mathsf{sn}}_{\{0,1\}^\kappa}(\{0,1\}^\kappa \times \{0,1\}^\kappa)$

let $\mathsf{rt} \leftarrow \mathsf{root}(t)$

let $\mathsf{cm} \leftarrow \mathsf{comm}_r(\mathsf{pk}_z, p)$

let $B \leftarrow B \cup \{(\mathsf{pk}_z, \mathsf{pk}_e, \mathsf{cm})\}$

let $M \leftarrow \mathsf{enc}((r, p), \mathsf{pk}_e)$

let $\mathcal{T}_\sigma \leftarrow ((\text{SPEND}, \mathsf{sn}, \mathsf{rt}), \varnothing) \parallel ((\text{MSG}, M), \varnothing) \parallel ((\text{MINT}, \mathsf{cm}), \varnothing)$

query $\mathcal{A}$ **with** $(\text{TRANSACTION}, \mathcal{T}_\sigma, D)$ and **receive the reply** $(\tau, \cdot)$, **satisfying** $T(\tau) = \bot \wedge \tau \neq \bot$, else **sampling from** $(\{0,1\}^\kappa, \bot)$
let $T(\tau) \leftarrow \{\mathsf{cm}\}$
return $(\tau, \varnothing)$
else if $x = (\text{SEND}, t)$ **then**
let $(\cdot, \mathsf{pk}) \xleftarrow{R} \mathsf{keyGen}(1^\kappa)$
let $\mathsf{rt} \leftarrow \mathsf{root}(t)$
let $\mathsf{cm} \xleftarrow{R} \mathsf{comm}_{\{0,1\}^\kappa}(\mathsf{prf}^{\mathsf{pk}}_{\{0,1\}^\kappa}(1), \{0,1\}^\kappa)$
let $\mathsf{sn} \xleftarrow{R} \mathsf{prf}^{\mathsf{sn}}_{\{0,1\}^\kappa}(\{0,1\}^\kappa \times \{0,1\}^\kappa)$
let $M \xleftarrow{R} \mathsf{enc}((\{0,1\}^\kappa, \{0,1\}^\kappa), \mathsf{pk})$
let $\mathcal{T}_\sigma \leftarrow ((\text{SPEND}, \mathsf{sn}, \mathsf{rt}), \varnothing) \parallel ((\text{MSG}, M), \varnothing) \parallel ((\text{MINT}, \mathsf{cm}), \varnothing)$
query $\mathcal{A}$ **with** $(\text{TRANSACTION}, \mathcal{T}_\sigma, D)$ and **receive the reply** $(\tau, \cdot)$, **satisfying** $T(\tau) = \bot \wedge \tau \neq \bot$, else **sampling from** $(\{0,1\}^\kappa, \bot)$
let $T(\tau) \leftarrow \{\mathsf{cm}\}$
return $(\tau, \varnothing)$
else if $x = \text{MINT}$ **then**
let $\mathsf{cm} \xleftarrow{R} \mathsf{comm}_{\{0,1\}^\kappa}(\mathsf{prf}^{\mathsf{pk}}_{\{0,1\}^\kappa}(1), \{0,1\}^\kappa)$
let $\mathcal{T}_\sigma \leftarrow ((\text{MINT}, \mathsf{cm}), \varnothing)$
query $\mathcal{A}$ **with** $(\text{TRANSACTION}, \mathcal{T}_\sigma, D)$ and **receive the reply** $(\tau, \cdot)$, **satisfying** $T(\tau) = \bot \wedge \tau \neq \bot$, else **sampling from** $(\{0,1\}^\kappa, \bot)$
let $T(\tau) \leftarrow \{\mathsf{cm}\}$
return $(\tau, \varnothing)$
else if $x = \text{BALANCE}$ **then**
let $\mathcal{T}_\sigma \leftarrow \epsilon$
query $\mathcal{A}$ **with** $(\text{TRANSACTION}, \mathcal{T}_\sigma, D)$ and **receive the reply** $(\tau, \cdot)$, **satisfying** $T(\tau) = \bot \wedge \tau \neq \bot$, else **sampling from** $(\{0,1\}^\kappa, \bot)$
return $(\tau, \varnothing)$
else abort
return (τ, a')

When receiving a message (INPUT, τ) *from* $\mathcal{F}_{\mathsf{sc}}^{\Delta_{\mathsf{pp}}, \Lambda_{\mathsf{pp}}}$*:*

send (INPUT, τ) **to** $\mathcal{A}$ and **receive the reply** $(p, w, (\mathcal{T}_\sigma, \mathcal{O}_\rho), \cdot, D)$
if $T(\tau) \neq \bot$ **then return** NONE
let $T(\tau) \leftarrow \varnothing$
if $w = (\text{SEND}, (\mathsf{pk}_z, \cdot))$ **then**
if $\mathcal{T}_\sigma = ((\text{SPEND}, \mathsf{sn}, \mathsf{rt}), \varnothing) \parallel ((\text{MSG}, M), \varnothing) \parallel ((\text{MINT}, \mathsf{cm}'), \varnothing)$ **then**
send $(\text{SEND}, \mathsf{pk}_e)$ **to** $\mathcal{O}_\rho$ and
receive the reply $(p, r, K_z, p', r', \mathsf{rt}', \mathsf{path}, M')$
let $\mathsf{cm} \leftarrow \mathsf{comm}_r((\mathsf{prf}^{\mathsf{pk}}_{K_z}, p))$
let $b \leftarrow \top$
send READ **to** $\mathcal{G}_{\mathsf{ledger}}$ and **receive the reply** Σ
if $\nexists t : 0 \leq t \leq |\Sigma| \wedge \mathsf{rt} = \mathsf{root}(t) \wedge \exists \tau : (T(\tau) = \mathsf{cm} \wedge \tau \in \Sigma[:t])$ **then**
let $b \leftarrow \bot$
if $\mathsf{sn} \neq \mathsf{prf}^{\mathsf{sn}}_{K_z}((p, r)) \vee \mathsf{rt} \neq \mathsf{rt}' \vee M \neq M'$ **then let** $b \leftarrow \bot$
if $\mathsf{cm}' \neq \mathsf{comm}_{r'}(\mathsf{pk}_z, p')$ **then let** $b \leftarrow \bot$
if $\neg b$ **then return** NONE

```
        // We now know the transaction is valid.
        // We must determine if M can be honestly decrypted,
        // and which adversarial coin is being spent.
        if ∃((·, K_e), (pk_z, pk_e)) ∈ K then
            let d = dec(M, K_e)
            if d = (r', p') then
                let w ← (SEND, (pk_z, pk_e))
            else
                let w ← (SEND, (SIMKEY, ⊥))
        else
            let B ← B ∪ {(SIMKEY, ⊥, cm')}
            let w ← (SEND, (SIMKEY, ⊥))
        if ∃(pk'_z, pk'_e, cm) ∈ B : pk'_z = prf^pk_{K_z} then
            let a ← (pk'_z, pk'_e)
        else abort
        let T(τ) ← {cm'}
        let z ← ∅
    else return NONE
else if w = MINT ∧ T_σ = ((MINT, cm), ∅) then
    let B ← B ∪ {(SIMKEY, ⊥, cm)}
    let T(τ) ← {cm'}
    let z ← (SIMKEY, ⊥); a ← ∅
else return NONE
return (p, w, z, a, D)
```

Helper procedures:

```
procedure root(t)
    let cms ← ∅
    send READ to G_ledger and receive the reply Σ
    for τ ∈ Σ[:t] do
        let cms ← cms ∪ T(τ)
    return merkleroot(cms)
```

D Security Analysis

Proof (of Theorem 1). If an environment can distinguish between the ideal and real executions in presence of our simulator (see Subsection D.1), then there must exist some polynomial sequence of interactions permitting it to distinguish with a non-negligible advantage. Broadly, each of the environment's actions fall into one of three categories: a) Honestly interacting with the protocol. b) Honestly interacting with the ledger. c) Commanding the adversary to perform some action in the real world. We will consider the responses the environment makes to queries given to the dummy adversary separately, in each case at the point where the query is made.

We will consider in parallel two random variables of the state of the ideal world execution, and that of the real world execution at any time. We leave out

of our analysis the "stack" of partial executions (as described in Appendix A), except to show that the flow of each party – i.e. when it is waiting for which query to be answered – is the same in both worlds. In particular, we note that the state of the ideal world has the following functionalities' states as a part of it: 1. the state of the simulator, $\mathcal{S}$, 2. the state of the smart contract functionality, $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$, and finally 4. the state of the ledger $\mathcal{G}_{\mathsf{L}}^{i}$. In the real world, for each $p \in \mathcal{H}$, p's protocol state, which we refer to as ϕ_p, is part of the state, along with the (shared) NIZK hybrid functionality $\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L},r}$, and the real-world ledger $\mathcal{G}_{\mathsf{L}}^{r}$. For convenience, we will often talk about these states as concrete variables, and not random variables.

We will prove inductively that any action the environment takes will do two things: First, it will preserve an invariant $\boldsymbol{I}$, which holds after the state of both worlds at any point during the two experiments. Second, if the invariant holds, the environment gains at most negligible advantage in distinguishing from its next action. To begin, we will specify the simulator, the invariant $\boldsymbol{I}$, followed by a few lemmas helpful in the proof. Finally, we will perform the induction itself.

D.1 The Simulator

The simulator for Kachina has fairly little work to do. It creates simulated transactions by creating a simulated NIZK proof, and attaching it to the leakage x. Secondly, when presented with an unknown transaction, and asked for the corresponding input, it attempts to extract the input from the simulated zero-knowledge functionality.

Simulator $\mathcal{S}_{\text{Kachina}}$

The simulator $\mathcal{S}_{\text{Kachina}}$ has two main points of interaction in the ideal world: First, it gets notified of the leakage of honest submissions, in the form of the new public state σ', and decides their format on the ledger. Second, it gets queried when an adversarial transaction is seen on the ledger, and must assign meaning to them. Furthermore, it simulates the non-global functionality $\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}$, which the adversary may interact with.

State variables and initialisation values:

Variable	Description
$\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}$	Simulation of $\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}$

When receiving a message $(\text{TRANSACTION}, \mathcal{T}_\sigma, D)$ *from* $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$*:*
  query $\mathcal{A}$ **with** $(\text{PROVE}, (\mathcal{T}_\sigma, D))$ and **receive the reply** π,
    satisfying $\pi \neq \bot \wedge (\cdot, \pi) \notin \mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}.\Pi \wedge (x, \pi) \notin \mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}.\overline{\Pi}$, else
    sampling from $\{0,1\}^\kappa$, **on behalf of** $\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}$
  let $\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}.\Pi \leftarrow \mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}.\Pi \cup \{((\mathcal{T}_\sigma, D), \pi)\}$
  return $((\mathcal{T}_\sigma, D, \pi), \varnothing)$
When receiving a message $(\text{INPUT}, (\mathcal{T}_\sigma, D, \pi))$ *from* $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$*:*

simulate sending $(\text{VERIFY}, (\mathcal{T}_\sigma, D), \pi)$ **to** $\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}$ and **receive the reply** b
if $b \wedge \exists w, \mathcal{T}_\rho : \mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}.W((\mathcal{T}_\sigma, D), \pi) = (w, \mathcal{T}_\rho)$ **then**
  return $(\mathcal{A}, w, (\mathcal{T}_\sigma, \mathcal{O}(\mathcal{T}_\rho)), \varnothing, D)$
else
  return NONE

Forward all queries to $\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}$ *to the simulated instance. Forward all queries, to global functionalities directly.*

D.2 The Invariant $\boldsymbol{I}$.

We define $\boldsymbol{I}$ as the conjunction of each of the constraints over the state variables over the two executions listed below.

(1) The ledgers are indistinguishable:
$\mathcal{G}_{\mathsf{L}}^{i}.\Sigma = \mathcal{G}_{\mathsf{L}}^{r}.\Sigma \wedge \forall p \in \mathcal{H} : \mathcal{G}_{\mathsf{L}}^{i}.M(p) = \mathcal{G}_{\mathsf{L}}^{r}.M(p)$

(2) The simulated and real NIZKs consider the same statement/proof pairs valid and invalid:
$\mathcal{S}.\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}.\Pi = \mathcal{F}_{\mathsf{nizk}}^{\mathcal{L},r}.\Pi \wedge \mathcal{S}.\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}.\overline{\Pi} = \mathcal{F}_{\mathsf{nizk}}^{\mathcal{L},r}.\overline{\Pi}$

(3) Real world witnesses have a corresponding ideal world witness:
$\forall \mathcal{T}_\sigma, D, \pi : \exists \mathcal{T}_\rho, w : \mathcal{F}_{\mathsf{nizk}}^{\mathcal{L},r}.W(((\mathcal{T}_\sigma, D), \pi)) = (\mathcal{T}_\rho, w) \implies \mathcal{S}.\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}.W(((\mathcal{T}_\sigma, D), \pi)) = (\mathcal{T}_\rho, w) \vee [\exists p \in \mathcal{H}, z = (\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta) : \phi_p.T((\mathcal{T}_\sigma, D, \pi)) = (\mathcal{T}_\rho, z) \wedge \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T((\mathcal{T}_\sigma, D, \pi)) = (p, w, (\mathcal{T}_\sigma, z), \varnothing, D) \wedge \mathsf{run\text{-}\Gamma}(\sigma^\pi, \rho^\pi, w, z, \top) = (\cdot, \mathcal{T}_\sigma, \cdot, \mathcal{T}_\rho, \phi_p.Y((\mathcal{T}_\sigma, D, \pi)))]$

(4) Recorded transactions are proven, and only adversarial witnesses are known by the simulator:
$\forall \mathcal{T}_\sigma, D, \pi, p : \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T((\mathcal{T}_\sigma, D, \pi)) = (p, \ldots) \implies ((\mathcal{T}_\sigma, D), \pi) \in \mathcal{F}_{\mathsf{nizk}}^{\mathcal{L},r}.\Pi \wedge (p \notin \mathcal{H} \iff ((\mathcal{T}_\sigma, D), \pi) \in \mathcal{S}.\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}.W)$

(5) Honest parties record transactions correctly:
$\forall p \in \mathcal{H}, \tau : \tau \in \phi_p.T \iff \tau \in \phi_p.Y \iff \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T(\tau) = (p, \ldots) \wedge \tau \in \phi_p.U \implies \tau \in \phi_p.T$

(6) All recorded transactions respect dependencies and transcripts:
$\forall \tau \in \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T : \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T(\tau) = \text{NONE} \vee (\exists \mathcal{T}, D, \pi : \tau = (\mathcal{T}, D, \pi) \wedge \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T(\tau) = (\cdot, \cdot, (\mathcal{T}, \cdot), \varnothing, D))$

(7) Recorded as rejected transactions are disproven:
$\forall \mathcal{T}_\sigma, D, \pi : \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T((\mathcal{T}_\sigma, D, \pi)) = \text{NONE} \implies ((\mathcal{T}_\sigma, D), \pi) \in \mathcal{F}_{\mathsf{nizk}}^{\mathcal{L},r}.\overline{\Pi}$

(8) The dependency invariant J holds for all honest unconfirmed transactions:
$\forall p \in \mathcal{H}$, let Σ be the longest prefix of $\mathcal{G}_{\mathsf{L}}^{r}.M(p)$ such that $\Sigma \cap \phi_p.U = \varnothing$; define $X(u = (\cdot, D, \cdot)) :=$ let $(\mathcal{T}, z) = \phi_p.T(u)$ in $(u, \mathcal{T}, z, D)$, and $((\cdot, \rho), \cdot, C) := \phi_p.\mathsf{exec}(\Sigma)$. Then $J(\mathsf{map}(X, \phi_p.U), \rho) \wedge \mathsf{sat}(\mathsf{map}(X, \phi_p.U), \phi_p.U) \wedge \forall (\cdot, D, \cdot) \in \phi_p.U : D \setminus C \setminus \phi_p.U = \varnothing$ holds.

(9) Transactions owned by an honest party, and not in their view of the ledger, are considered unconfirmed, or can never be accepted: Let Σ be the longest prefix of $\mathcal{G}_{\mathsf{L}}^{r}.M(p)$ such that $\forall \tau \in \Sigma : \tau \in \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T$.
$\forall p \in \mathcal{H}, \tau \notin \Sigma : \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T(\tau) = (p, \ldots) \implies \tau \in \phi_p.U \vee (\nexists \Sigma' \succ \mathcal{G}_{\mathsf{L}}^{r}.M(p) : \tau \in \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.\mathsf{execConfirmed}(\Sigma' \parallel \tau)$

(10) All results and state updates are consistent with the input and transcripts: $\forall p \in \mathcal{H}, \mathcal{T}_\sigma, \tau = (\mathcal{T}_\sigma, \cdot, \cdot), w, z : \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T(\tau) = (p, w, z, \varnothing, \cdot) \implies [\exists \mathcal{T}_\rho : \phi_p.T(\tau) = (\mathcal{T}_\rho, z) \wedge \phi_p.Y(\tau) = \Gamma_{\mathcal{O}(\mathcal{T}_\sigma),\mathcal{O}(\mathcal{T}_\rho)}(w) \wedge [\forall \sigma, \rho : \mathcal{T}_\sigma(\sigma) \neq \bot \wedge \mathcal{T}_\rho(\rho, z) \neq \bot \implies (\mathcal{T}_\sigma(\sigma), \cdot, \mathcal{T}_\rho(\rho, z), \cdot, \phi_p.Y(\tau)) = \mathsf{run\text{-}\Gamma}(\sigma, \rho, w, z, \top)]]$

(11) Execution results should be equivalent for prefixes *and* extensions of the ledger state containing no new adversarial transactions: $\forall \Sigma, p \in \mathcal{H} : ((\Sigma \prec \mathcal{G}_{\mathsf{L}}^r.\Sigma \vee \mathcal{G}_{\mathsf{L}}^r.\Sigma \prec \Sigma) \wedge \forall \tau \in \Sigma : \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T(\tau) \neq \bot) \implies$ let $((\sigma^i, \boldsymbol{\rho}^i), y^i, C^i) \leftarrow \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.\mathsf{exec}(\Sigma); ((\sigma^r, \rho^r), y^r, C^r) \leftarrow \phi_p.\mathsf{exec}(\Sigma)$ in $\sigma^i = \rho^i \wedge \boldsymbol{\rho}^i[p] = \rho^r \wedge C^i = C^r \wedge$ if $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T(\Sigma[-1]) = (p, \ldots)$ then $y^r = y^i$ else $y^r = \bot$

(12) Recorded transactions which are canonically preceeded by a (yet) unrecorded transaction, are honest, and considered unconfirmed by their owner: $\forall \tau \in (\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T \cap \mathcal{G}_{\mathsf{L}}^r.\Sigma), \tau' \in (\mathcal{G}_{\mathsf{L}}^r.\Sigma \setminus \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T), p \in \mathcal{H} : \mathsf{idx}(\mathcal{G}_{\mathsf{L}}^r.\Sigma, \tau') < \mathsf{idx}(\mathcal{G}_{\mathsf{L}}^r.\Sigma, \tau) \wedge \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T(\tau) = (p, \ldots) \implies \tau \in \phi_p.U \vee (\nexists \Sigma' \succ \mathcal{G}_{\mathsf{L}}^r.\Sigma : \tau \in \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.\mathsf{execConfirmed}(\Sigma' \parallel \tau))$

(13) The ledger is ahead of any party's ledger: $\forall p \in \mathcal{H} : \mathcal{G}_{\mathsf{L}}^r.M(p) \prec \mathcal{G}_{\mathsf{L}}^r.\Sigma$

(14) The same transactions are unconfirmed in both worlds: $\forall p \in \mathcal{H} : \phi_p.U = \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.U_p$

(15) NIZK proofs have witnesses: $\forall x, \pi : (x, \pi) \in \mathcal{F}_{\mathsf{nizk}}^{\mathcal{L},r}.\Pi \iff (\exists w : \mathcal{F}_{\mathsf{nizk}}^{\mathcal{L},r}.W((x, \pi)) = w \wedge \mathcal{S}.\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}.W((x, \pi)) \in \{w, \bot\} \wedge (x, w) \in \mathcal{L})$

(16) Recorded transactions are either on the ledger, considered unconfirmed by an honest party, or can never be satisfied: $\forall \tau \in \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T : \tau \in \mathcal{G}_{\mathsf{L}}.\Sigma \vee (\exists p \in \mathcal{H} : \tau \in \phi_p.U \wedge \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T(\tau) = (p, \ldots)) \vee (\nexists \Sigma' \succ \mathcal{G}_{\mathsf{L}}.\Sigma : \tau \in \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.\mathsf{execConfirmed}(\Sigma' \parallel \tau))$

Often many of these parts of the invariant are trivially preserved due to the state variables constrained in them being left unchanged. Such trivial cases will be omitted in our analysis.

D.3 Supporting Lemmas

Lemma 2 is the kernel of the proof of Kachina, but its formal statement is not particularly intuitive. The purpose is quite simple: Both $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$ and Kachina have exec functions, which executes an entire ledger state given to it. Lemma 2 is a generalisation of invariant (11), and simply states that this execution will preserve the invariant, and return the same values in the real and ideal world.

Lemma 2. *For any $p \in \mathcal{H}, \Sigma$ where $\Sigma \prec \mathcal{G}_{\mathsf{L}}^r.\Sigma \vee \mathcal{G}_{\mathsf{L}}^r.\Sigma \prec \Sigma$, and after sending the message $(\text{EXTEND}, \Sigma \setminus \mathcal{G}_{\mathsf{L}}^r.\Sigma)$ to $\mathcal{G}_{\mathsf{L}}$, $((\sigma^i, \boldsymbol{\rho}^i), y^i, C^i)$ is the result of running $\mathsf{exec}(\Sigma)$ in $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$, and $((\sigma^r, \rho^r), y^r, C^r)$ is the result of running $\mathsf{exec}(\Sigma)$ in ϕ_p, these interactions preserve $\boldsymbol{I}$, and the returned values are equivalent: $\sigma^i = \sigma^r \wedge \boldsymbol{\rho}^i[p] = \rho^r$. If the last transaction τ in Σ is owned by p (i.e. $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T(\tau) = (p, \ldots)$), then $y^i = y^r$, otherwise $y^r = \bot$.*

Proof. First, we consider the EXTEND call. This will only extend if Σ is longer than $\mathcal{G}_{\mathsf{L}}.\Sigma$ – otherwise it extends with ϵ, which is a no-op. This call preserves $\boldsymbol{I}$, as demonstrated in Subsection D.4.

We prove by induction over Σ. In the base case, $\Sigma = \epsilon$. The invariant is trivially preserved, and the returned values are equivalent (when $\varnothing$ is interpreted as public/private state pairs). In the induction step, we proceed by case analysis for the new transaction $\tau = (\mathcal{T}, D, \pi)$:

Case 1. The $\tau \in \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T$, and all processed transactions so far have been also been recorded (are in $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T$). If so, then by (11), the return values are equivalent. Further, this iteration does not change the state in the ideal world. By (4) and (7), we also know that the transaction is either in $\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L},r}.\Pi$, or $\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L},r}.\overline{\Pi}$. As a result, no state changes will be made in the real-world execution either, trivially preserving $\boldsymbol{I}$.

Case 2. $\tau \notin \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T$, but $((\mathcal{T}, D), \pi) \in \mathcal{F}_{\mathsf{nizk}}^{\mathcal{L},r}.\overline{\Pi}$. In this case, the real world will skip this transaction, and set y to $\perp$. In the ideal world, the simulator will ensure that $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T(\tau)$ is set to NONE, and equally this transaction is skipped, with y set to $\perp$. This affects and preserves the following invariants:

(3) As by (15), τ has no witness.
(4) As $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T(\tau) = \text{NONE}$, not satisfying the precondition.
(5) As τ was not in $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T$ in the induction hypothesis, and is not associated with an honest party.
(6) By $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T(\tau)$ being NONE, satisfying the postcondition.
(7) Due to $((\mathcal{T}, D), \pi) \in \mathcal{F}_{\mathsf{nizk}}^{\mathcal{L},r}.\overline{\Pi}$.
(9) As $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T(\tau) = \text{NONE}$, not satisfying the precondition.
(10) As τ was not in $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T$ in the induction hypothesis, it cannot be in any $\phi_p.Y$, by (5).
(11) By the output equivalence part of the induction step holding.
(12) By τ being previously unrecorded, further restricting the quantification domain, and $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T(\tau) = \text{NONE}$, not satisfying the precondition.
(16) By the newly recorded transaction being in the ledger state, as this has been extended if necessary.

Case 3. $\tau \notin \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T$, but $((\mathcal{T}, D), \pi) \in \mathcal{F}_{\mathsf{nizk}}^{\mathcal{L},r}.\Pi$. In this case, by (15) a witness must be recorded, and by (3) this witness must be accessible to the simulator. As a result, the simulator will ensure that $T(\tau)$ is set to $(\mathcal{A}, w, (\mathcal{T}_\sigma, \mathcal{O}(\mathcal{T}_\rho)), \varnothing, D)$. As this is an adversarial transactions, the ρ-value of the adversary is not constrained, and neither is the output y-value. As a result, to show the execution equivalence holds, it suffices to show that both worlds will have the same σ-value after this new transaction is the same in both worlds. In the real world, $\mathcal{T}_\sigma(\sigma)$ is applied, or if this is $\perp$, the transaction is skipped. In the ideal world, $\mathcal{T}_\sigma(\sigma)$ is first tested to be $\perp$, and if it is, the transaction is also skipped. Otherwise, $\mathsf{run}\text{-}\Gamma(\sigma, \cdot, w, \mathcal{O}(\mathcal{T}_\rho), \perp)$ is run. Since $\mathcal{T}_\sigma(\sigma) \neq \perp$, and $((\mathcal{T}_\sigma, D), (\mathcal{T}_\rho, w)) \in \mathcal{L}$ (by (15)), we know that the public state oracle used in $\mathsf{run}\text{-}\Gamma$ can be replaced with

$\mathcal{O}(\mathcal{T}_\sigma)$, and that the call $\Gamma_{\mathcal{O}(\mathcal{T}_\sigma),\mathcal{O}(\mathcal{T}_\rho)}(w)$ will terminate successfully. As a result, the value σ returned in the ideal world is the same as $\mathcal{T}_\sigma(\sigma)$, that is returned in the real world. As $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T$ is set, the following parts of the invariant are affected and preserved:

(3) By the left hand side of the disjunction already being satisfied.
(4) By the transaction being recorded in the NIZK, and the simulator knowing its witness.
(5) As τ was not in $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T$ in the induction hypothesis, and is not associated with an honest party.
(6) By the newly recorded transaction satisfying the postcondition.
(7) By the newly recorded transaction not being recorded as rejected.
(9) By the newly recorded transaction not being honestly owned, not satisfying the precondition.
(10) As τ was not in $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T$ in the induction hypothesis, it cannot be in any $\phi_p.Y$, by (5).
(11) By the output equivalence part of the induction step holding.
(12) By τ being previously unrecorded, further restricting the quantification domain, and $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T(\tau) = (\mathcal{A}, \ldots)$, not satisfying the precondition.
(16) By the newly recorded transaction being in the ledger state, as this has been extended if necessary.

Case 4. The transaction has not been previously seen – that is $\tau \notin \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T$, and $((\mathcal{T}, D), \pi) \notin \mathcal{F}_{\mathsf{nizk}}^{\mathcal{L},r}.\Pi \cup \mathcal{F}_{\mathsf{nizk}}^{\mathcal{L},r}.\overline{\Pi}$. In this case, both the real and ideal worlds will attempt the same NIZK verification (simulated in the ideal world). By (2), they will both query the adversary for a NIZK witness in the same way, handing off execution. By the induction hypothesis, $\boldsymbol{I}$ holds as the point of execution transfer, and as the query made is the same in both worlds, the environment gains no means to distinguish.

As NIZK verification is the first thing done in both worlds, and NIZK verification is agnostic as to which party is verifying, this is equivalent to the environment *first* manually verifying the same statement/proof pair. As will be shown in Subsection D.4, this preserves the invariant, and returns the same result in both worlds. Therefore, Case 4 is equivalent to either Case 2 (if the NIZK verification failed), or Case 3 (if the NIZK verification succeeded), as if the NIZK verification is done externally beforehand, the statement/proof pair *must* be either in $\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L},r}.\Pi$, or in $\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L},r}.\overline{\Pi}$.

Case 5. $\tau \in \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T$, however (11) cannot be applied, as other transactions have since been added. By (12), we know that τ belongs to an honest party p', and that $\tau \in \phi_{p'}.U$. We will use (8) to argue that, where $(\mathcal{T}_\rho, z) = \phi_{p'}.T(\tau)$, either $\mathcal{T}_\rho(\boldsymbol{\rho}^i[p'], z) \neq \bot$, or the transaction is skipped in both worlds.

First, we consider the possibility that $\tau \notin \phi_{p'}.U$. By (9) we know that τ cannot ever be confirmed by a suffix of the ledger state referred to in the invariant. As this is a prefix of $\mathcal{G}_{\mathsf{L}}^r.\Sigma$, such that it contains no unrecorded transactions, the current induction is necessarily a suffix of it. As a result, we know that the ideal

world execution will fail. As transactions are rejected in both worlds under the same conditions – either dependencies are not satisfied, or $\mathcal{T}_\sigma(\sigma) = \bot$, we can conclude that these transactions are also skipped in the real world, preserving $\boldsymbol{I}$ as no state variables are changed, and satisfying all conditions by the induction hypothesis. We will now focus on the case that $\tau \in \phi_{p'}.U$

Next, we determine that, given $\tau \in \phi_{p'}.U$, the longest prefix Σ^* referred to in (8) is a prefix of the ledger state Σ we are currently performing induction over. We know it to be a prefix of $\mathcal{G}_\mathsf{L}.M(p')$, such that this prefix contains none of the transactions in $\phi_{p'}.U$. As $\tau \in \phi_{p'}.U$, and is either a prefix or extension of $\mathcal{G}^r_\mathsf{L}.\Sigma$, of which $\mathcal{G}^r_\mathsf{L}.M(p')$ is itself a prefix by (13), we can conclude that $\Sigma^* \prec \Sigma$.

As, to apply (8), we are only concerned with the parties private state ρ, we can observe all transactions in $\Sigma \setminus \Sigma^*$ are either not owned by p', will not be accepted in any context, or are in $\phi_{p'}.U$. We can ignore the first possibility, as the real world execution of them will not affect ρ, regardless. The second can also be ignored, as these will be skipped by the ideal world execution, and by induction hypothesis, by the real world execution as well. Next, we consider which of the transaction in $\Sigma \setminus \Sigma^*$ owned by p' have been successfully processed. exec provides replay protection, ensuring that each unconfirmed transaction has been processed at most once. By induction hypothesis, the sequence A of such transactions that are accepted, and therefore confirmed, is the same in both worlds. By virtue of exec rejecting transactions which would set the state to $\bot$, we know that $\mathcal{T}^*_{\mathsf{map}(\phi_{p'}.T,A)}(\boldsymbol{\rho}^*[p']) \neq \bot$, where $\boldsymbol{\rho}^*$ is taken to be the private state corresponding to the prefix Σ^* in the ideal world. As each transaction in A got confirmed in order, we know that $\mathsf{sat}(\mathsf{map}(X, A), \phi_{p'}.U)$ holds, where X is the function defined in (8), as this performs a more relaxed variant of the confirmation check.

Now that τ is processed, we know by (8) that its dependencies are either in confirmed, and in order, in Σ^*, or in $\phi_{p'}.U$. In either case, τ is skipped in both worlds if it is a replayed transaction, or if would set $\mathcal{T}_\sigma(\sigma) = \bot$. In the ideal world, additionally it gets skipped if it would set the contract state – either σ^i or $\boldsymbol{\rho}^i$ – to $\bot$.

As $\tau \in \phi_{p'}.U$, and is not a replay, $B = A \parallel \tau$ is a permutation of a subset of $\phi_{p'}.U$. As a result, by (8), we know that $\mathcal{T}^*_{\mathsf{map}(\phi_{p'}.T,B)}(\boldsymbol{\rho}^*[p']) \neq \bot$. As we've previously established this holds for A, by definition of $\mathcal{T}^*$, this implies that $\mathcal{T}_\rho(\boldsymbol{\rho}[p']^i, z) \neq \bot$, where $\boldsymbol{\rho}[p']$ is the same as the ideal world private state for the induction hypothesis, by repeated application of (10). Likewise, (10) allows us to conclude that σ^i will also be non-$\bot$, as the update applied to it will be equivalent to $\mathcal{T}_\sigma(\sigma^i)$, which by case analysis is not $\bot$.

If the transaction is skipped in both worlds, the induction hypothesis still applies. Otherwise, both $\mathcal{T}_\sigma(\sigma^i) \neq \bot$ and $\mathcal{T}_\rho(\boldsymbol{\rho}^i[p'], z) \neq \bot$. As previously noted in Subsection 4.1, we can conclude that the updated states in the ideal world are equal to $\sigma^i \leftarrow \mathcal{T}_\sigma(\sigma^i)$ and $\boldsymbol{\rho}^i[p'] \leftarrow \mathcal{T}_\rho(\boldsymbol{\rho}^i[p'], z)$. Likewise, (10) applies (as by (5), $\phi_{p'}.Y(\tau)$ is set), and we know the ideal-world result $y^i = \phi_{p'}.Y(\tau)$.

As in the real world $\sigma^r \leftarrow \mathcal{T}_\sigma(\sigma^r)$, their equality is preserved. If $p = p'$, by (5), $\phi_p.T$ is defined, and as a result, the same update is carried out to ρ

in the real world, as to $\boldsymbol{\rho}^i[p]$ in the ideal world. Further, it will return the same result y^r as the ideal world, as $\phi_p.Y(\tau) = y^i$. If $p \neq p'$, the ideal world update does not affect $\boldsymbol{\rho}^i[p]$, and the correctness of the returned private state is guaranteed by the induction hypothesis. $y^r = \bot$ is returned, which satisfies the requirements. Finally, in both cases, as the transaction is not skipped, it is added to C, ensuring the returned C is the same in both worlds. Neither world makes any state updates, trivially preserving $\boldsymbol{I}$. □

Lemma 3. *If $\boldsymbol{I}$ holds, then for all $p \in \mathcal{H}$, running* $\mathsf{updateState}(p)$ *in $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$, and running* updateState *in ϕ_p preserves $\boldsymbol{I}$.*

Proof. To begin with, both worlds retrieve the same value Σ / Σ_p from $\mathcal{G}_{\mathsf{L}}$, due to (1). As seen in Subsection D.4, this preserves $\boldsymbol{I}$. Next, by Lemma 2, both worlds receive the same value C, and the execConfirmed call preserves $\boldsymbol{I}$. The worlds now iterate over $\phi_p.U$ and $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.U_p$ respectively, which by (14) are equal in value. The operations performed are almost identical, with the exception of the real world deconstructing $u = (\cdot, D, \cdot)$ for each $u \in U$, while the ideal world extracts $(\ldots, D) = \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T(u)$ instead. By (6), if $u \in \mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T$, the two are equivalent, and by (5), as $u \in \phi_p.U$, it is also in both $\phi_p.T$, and $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T$. We conclude that both worlds perform the same operations. Updated is only $\phi_p.U$ and $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.U_p$ respectively. The following parts of the invariant are affected and preserved:

(5) By reducing the scope of $\phi_p.U$.
(8) This consists of three sub-parts: The satisfaction of J, that of sat, and that $D \backslash C \backslash U = \varnothing$. The first is trivial: J makes a statement about all permutations of subsets. A smaller initial set merely reduces the scope of the quantifiers. The second holds due to updateState ensuring that if a transaction is removed from U, any transactions that depend on it are also removed, with the remaining transactions being in the same order as before. As a result, a previously satisfied transaction is either removed itself, or still satisfied, as it does not depend on any removed transactions, and dependencies still in U being in the same order as before. Finally, $D \setminus C \setminus U = \varnothing$ is also preserved due to the recursive removal. Specifically, if $D \nsubseteq (C \cup U)$ the corresponding transaction is removed. As a result, only transactions satisfying this condition will remain.
(9) As the removed transactions either fail confirmation directly (it depends on a transaction rejected in Σ_p, or a different transaction order than got enforced), or depends on a transaction which fails. In either case, any state Σ', of which Σ_p is a prefix, cannot accept these for the same reasons.
(12) As in (9).
(14) By equal update. □

D.4 Proof of Theorem 1

We proceed with the main inductive proof of Theorem 1. We consider the base case of the system initializations in the real and ideal worlds. The induction

hypothesis is that after $k < 2^\kappa$ interactions with any environment, the state of both worlds satisfy the invariant $\boldsymbol{I}$, and the environment has not gained a non-negligible advantage in distinguishing. We will assume, without loss of generality, the adversary being a dummy adversary. We provide a concrete list of actions the environment may take before taking the induction step. We note that as at any point the environment cannot distinguish, we can assume that it takes the same action in both worlds without loss of generality.

Base Case.

Proof. Most base cases hold either due to equal initialization of variables constrained to be equal, or due to initialization leaving forall quantifiers to quantify over the empty set. The former is the case for: (1), (2), (5), and (14). The latter is the case for: (3), (4), (6), (7), (9), (10), (12), and (16). The remaining hold for the following reasons:

(8) At initialisation, the only prefix of $\mathcal{G}^r_{\mathsf{L}}.M(p)$ is ϵ. $\phi_p.\mathsf{execState}(\epsilon) = (\varnothing, \varnothing)$. The base case therefore holds iff $J(\varnothing, \varnothing)$ holds. This in turn holds iff $\mathcal{T}^*_\epsilon(\varnothing) \neq \bot$, or $\varnothing \neq \bot$.
(11) At initialisation, the only ledger state Σ which satisfies the condition that $\forall \tau \in \Sigma : \mathcal{F}^{\Delta,\Lambda}_{\mathsf{sc}}.T(\tau) \neq \bot$ is ϵ. For this, as both world's are initialised to equivalent contract states, the outputs of exec will be equal.
(13) By the reflexivity of $\prec$. □

Induction Step.

Proof. We observe that the environment is capable of the following queries:

- $\forall p \in \mathcal{H}, w$:[4] Sending $(\textsc{post-query}, w)$ to $\mathcal{F}^{\Delta,\Lambda}_{\mathsf{sc}}$ or Kachina.
- $\forall p \in \mathcal{H}, \tau$: Sending $(\textsc{check-query}, \tau)$ to $\mathcal{F}^{\Delta,\Lambda}_{\mathsf{sc}}$ or Kachina.
- $\forall p \in \mathcal{P}, \tau$: Sending $(\textsc{submit}, \tau)$ to $\mathcal{G}_{\mathsf{L}}$.
- $\forall p \in \mathcal{P}$: Sending $\textsc{read}$ to $\mathcal{G}_{\mathsf{L}}$.
- $\forall \Sigma'$: Sending $(\textsc{extend}, \Sigma')$ to $\mathcal{G}_{\mathsf{L}}$.
- $\forall p, \Sigma'$: Sending $(\textsc{advance}, p, \Sigma')$ to $\mathcal{G}_{\mathsf{L}}$.
- $\forall p \in \mathcal{P} \setminus \mathcal{H}$: Sending $(\textsc{prove}, x, w)$ to $\mathcal{F}^{\mathcal{L}}_{\mathsf{nizk}}$.
- $\forall p \in \mathcal{P}$:[5] Sending $(\textsc{verify}, x, \pi)$ to $\mathcal{F}^{\mathcal{L}}_{\mathsf{nizk}}$.

We will prove that $\boldsymbol{I}$ is preserved across any of these queries, and that they reveal the same information in both worlds.

[4] We omit without loss of generality the environments ability to make honest queries with corrupted parties. The environment may simulate running the honest protocol to replicate these.

[5] Technically, as in $\textsc{prove}$, the environment can only instruct corrupted parties to verify. As verification for honest parties preserves the invariant as well, and is a useful lemma, we prove the more general statement.

Case (POST-QUERY, w). We proceed by sub-case analysis. We identify the following cases: 1. The transaction is rejected by the contract. 2. The transaction is rejected by the user. 3. The transaction is posted. In all cases, $\mathsf{updateState}$ is first run. By Lemma 3, this preserves the invariant, and also ensures that the returned value $\Sigma_p = \mathcal{G}_{\mathsf{L}}.M(p)$ is the value returned in both worlds (by (1)). In the ideal world, Λ is called. The real world largely emulates the same, computing most of the same values identically. Of note are the values σ^o and $\rho^o/\boldsymbol{\rho}^o[p]$, which are computed in both worlds using $\mathsf{execState}(\Sigma_p)$. By Lemma 2, this preserves the invariant, and returns the same values.

The only place where the two worlds diverge in their computation is in handling the unconfirmed transactions – the ideal world executes $\mathsf{run}\text{-}\Gamma$, and updates σ^π, ρ^π, and X accordingly, while the real world applies $\mathcal{T}_\sigma$ and $\mathcal{T}_\rho$ to the states. Before we go into the main three cases, we will argue that, if the transaction is *not* rejected by the contract, then these two approaches will yield the same result, and that they will reject equally.

To begin with, the ideal world *does* also apply the public state transcript, specifically to check that the result is non-$\bot$. If this is not satisfied, it immediately results in the new query being rejected. Correspondingly, the real world will actually apply this transcript, resulting in $\sigma^\pi = \bot$. As a $\bot$ state will remain $\bot$, this will further in the execution result in the final public state σ also being $\bot$, and the real world execution rejecting this query.

Further, we observe that in the real world, the final value of ρ^π *cannot* be $\bot$ – to begin with, $\mathsf{updateState}$ guarantees that $\Sigma_p \cap \phi_p.U = \varnothing$. This in turn, along with (8) ensures that $J(X)$ holds, as well as that $\mathsf{sat}(X, U)$ holds. It follows $\mathcal{T}^*(\rho^o) \neq \bot$, where $\mathcal{T}^*$ performs the same repeated applications of $\mathcal{T}_\rho(\rho, z)$ as the loop in the main protocol, using the same values. Further, by (3) and (4), we can conclude that the transcripts $\mathcal{T}_\rho$, and contexts z are the same in both worlds. By (10) , we can conclude that the final ρ^π values are also the same in both worlds. Further, as $\mathcal{T}_\rho$ and z are equal in both worlds, and by (6) D is also equal, the sequence X is also equal. Subsequently, σ, ρ, and D are computed equivalently in both worlds.

We now consider the main case analysis: If the contract rejects the transaction in the ideal world, the returned description is $\bot$. This happens if and only if $\sigma = \bot \vee \rho = \bot$, the same condition as the real world protocol has for rejecting the transaction before querying the user. If the transaction is rejected, no variables are modified, preserving $\boldsymbol{I}$, and the same value is returned in both worlds, giving the environment no means to distinguish.

If the contract does not automatically reject the query, the leakage descriptor is computed equally in both worlds, and sent to the party to acknowledge. The party has the opportunity to accept the described leakage, or cancel the transaction. At the point of handing over execution to the environment, no state has been modified, trivially preserving $\boldsymbol{I}$, and as the same leakage descriptor is given, it has no means to distinguishing.

In the case of the environment subsequently cancelling the transaction, both worlds immediately return with REJECTED, again trivially preserving $\boldsymbol{I}$, and giving no means to distinguish.

Finally, if the environment accepts the leakage, both worlds obtain the transaction identifier τ: The simulator ensures that the real-world adversary is queried for the same NIZK proof as it is in the real-world, and that the transaction format matches that of real-world transactions. At the time of the proof query, no state has been modified, trivially preserving $\boldsymbol{I}$. As the same statement is queried for, the environment gains no information to distinguish.

Subsequently, both worlds record the transaction's information (in $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T$ and $\phi_p.T$), and note is as unconfirmed (in $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.U$ and $\phi_p.U$). In the real world, the result is further recorded in $\phi_p.Y$. The following parts of $\boldsymbol{I}$ are affected and preserved (including the PROVE query):

(2) By $((\mathcal{T}_\sigma, D), \pi)$ being added to both worlds' Π equally.
(3) As $\phi_p.T$, $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T$, and $\phi_p.Y$ are appropriately set to satisfy the RHS of the disjunction.
(4) As for the newly added transaction, $p \in \mathcal{H}$, and $((\mathcal{T}_\sigma, D), \pi) \notin \mathcal{S}.\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L}}.W$ (by the uniqueness of statement/proof pairs).
(5) As the newly added transaction is added to all of $\phi_p.T$, $\phi_p.U$, and $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T$, where it is associated with p.
(6) As the newly added transaction does consist of transcript, dependencies and proof, and the former two are recorded in $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T$ correctly, and $\mathcal{S}$ returns $\varnothing$ for a.
(7) As the newly recorded transaction is not recorded as NONE.
(8) By J being preserved when appending a new transaction, J holds after the induction step (as ρ remains unaffected). sat holds by induction hypothesis, and as $D \sqsubseteq U \setminus \{\tau\}$. For the new transaction, $D \setminus C \setminus \phi_p.U = \varnothing$ as $D \subseteq \phi_p.U$; for previously transactions this still holds, as $\phi_p.U$ is expanded.
(9) As the newly added transaction is also unconfirmed by the owning party.
(10) By $\mathcal{T}_\sigma$, $\mathcal{T}_\rho$, and y having been extracting from $\mathsf{run}\text{-}\Gamma$, operating in the context of w, z, and some σ, ρ, with these values being recorded in the corresponding state variables (except σ and ρ). As the transcripts are transcripts of oracle evaluations against w, and y is the result of Γ operating with these oracles, executing $\Gamma_{\mathcal{O}(\mathcal{T}_\sigma),\mathcal{O}(\mathcal{T}_\rho)}(w)$ has the same effect. Further, as the transcripts accurately reproduce the state change in the original state context, by definition of transcript execution, and if an applied transcript is not $\perp$, it is indistinguishable from making the original queries to the state oracle, combined with the sequence of queries made depending only on w and the state oracle itself, we can conclude that the transcript application is the same as executing against the state oracles, regardless of which initial state the transcript could be successfully applied to.
(11) As a new transaction has been recorded, we must now additionally consider transaction sequences Σ which contain this new transaction at some point. We cannot directly use Lemma 2, however we can make use of its induction: If we can show that any Σ *ending* with the new transaction τ satisfies the

execution equivalences, then induction from Lemma 2 can apply on that as a base case (in particular, the precondition for Case 5 applies for all subsequent transactions). The execution equivalence holds for this new base case, as we know that this new transaction is both honest, and considered unconfirmed for this party. Therefore, the argument for Case 5 holds for τ itself as well. As the execution equivalence defined in Lemma 2 is the same as that of (11), this part of the invariant is preserved.

(12) By the newly added transaction being unconfirmed, it satisfies any quantification where τ is set to it. By now being recorded, the range of quantifications for τ' is restricted, relaxing the condition.

(14) By equal update.

(16) By the newly recorded transaction being considered unconfirmed by an honest party.

Finally, both worlds submit to the ledger the same transaction τ, which simply sent to the adversary. At this point $\boldsymbol{I}$ holds as argued above, and as the same transaction is sent, the environment cannot distinguish. Finally, (POSTED, τ) is returned, giving the environment no information to distinguish for the same reasons.

Case $(\text{CHECK-QUERY}, \tau)$. After running $\mathsf{updateState}$, Lemma 3 preserves the invariant, but also ensures that $\Sigma_p = \mathcal{G}_{\mathsf{L}}.M(p)$, where Σ_p is the value returned in both worlds (by (1)).

We consider three cases: 1. $\tau \notin \Sigma_p$, 2. $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}.T(\tau) = (p, \ldots)$, and 3. otherwise. In Case 1, both worlds return NOT-FOUND without updating any state, not allowing the environment to distinguish, and preserving $\boldsymbol{I}$. In Case 2, both worlds run $\mathsf{execResult}(\mathsf{prefix}(\Sigma_p, \tau))$, preserving $\boldsymbol{I}$ according to Lemma 2, and returning the same value in both worlds, giving the environment no information to distinguish. Finally, in Case 3, only the real world runs $\mathsf{execResult}$, while the ideal world returns $\perp$. As previously in $\mathsf{updateState}$ the sub-function $\mathsf{execConfirmed}$ was run, we know that all NIZK-verifications performed in this exec call have previously been made – as a result the call modifies no state, and preserved $\boldsymbol{I}$. Further, by Lemma 2, it returns $\perp$, as in the ideal world, giving the environment no information to distinguish.

Case (SUBMIT, τ). In both worlds τ is handed to the adversary, and no other action is taken. As the same information is relayed, the environment cannot distinguish, and as no state is changed, $\boldsymbol{I}$ is preserved.

Case READ. By (1), both worlds will return the same result; therefore the environment cannot distinguish. As no state is changed, $\boldsymbol{I}$ is preserved.

Case (EXTEND, Σ'). As nothing is returned, the environment gains no information allowing it to distinguish. By (1), the updates done are the same in both worlds. The parts of the invariant affected and preserved are the following:

(1) By equal update.

(11) Extending $\mathcal{G}_{\mathsf{L}}.\Sigma$ further constrains the possible Σ values quantified over.
(12) Without loss of generality, we can assume single-transaction appends to $\mathcal{G}^r_{\mathsf{L}}.\Sigma$. If a new unrecorded transaction is added, it (at first) does not precede any transactions, leaving the quantification unchanged, and relaxing the non-existance quantifier. If a recorded honest transaction is added, then by (9) and (13), this transaction satisfies the conditions.
(13) By the append-only nature of EXTEND.
(16) By relaxing the constraint.

Case (ADVANCE, p, Σ'). As nothing is returned, the environment gains no information allowing it to distinguish. By (1), the updates done are the same in both worlds. The parts of the invariant affected and preserved are the following:

(1) By equal update.
(8) Without loss of generality, we can assume single-transaction advances. If $\mathcal{G}_{\mathsf{L}}.M(p) \cap \phi_p.U \neq \varnothing$, or the newly added transaction $\tau \in \phi_p.U$, this is preserved as the longest prefix remains equal. Otherwise, Σ in the induction step is that of the induction hypothesis, with one transaction $\tau \notin \phi_p.U$ appended. If τ is not owned by p, by (5), $\phi_p.T(\tau) = \bot$, and therefore $\mathsf{execState}(\Sigma \parallel \tau)$ returns the same ρ as $\mathsf{execState}(\Sigma)$, preserving the invariant. If τ is owned by p, by (9), this transaction will be rejected, likewise returning the same ρ. Further, as $D \setminus C \setminus U$ is already $\varnothing$ for all dependency lists D, and extending Σ can only lead to C growing, this condition remains satisfied.
(9) By further restricting all-quantification and non-existance quantification.
(13) By condition that $\mathcal{G}^r_{\mathsf{L}}.M(p) \prec \mathcal{G}^r_{\mathsf{L}}.\Sigma$.

Case (PROVE, x, w). In the ideal world, this query is handled by the simulated functionality $\mathcal{S}.\mathcal{F}^{\mathcal{L}}_{\mathsf{nizk}}$. If $(x, w) \notin \mathcal{L}$ the call returns immediately with $\bot$ in both worlds, and no variables are modified, giving the environment no information to distinguish, and preserving $\boldsymbol{I}$. Otherwise, the adversary is immediately queried with (PROVE, x) in both worlds. Again, at this point no variables have been modified, preserving $\boldsymbol{I}$, and the information handed to the adversary is the same in both worlds, giving the environment no information to distinguish. The adversary will eventually respond with a proof π, which is verified against constraints in both worlds, and randomly sampled if it does not meet them. By (2), the constraints are identical in both worlds. Finally Π and W are set, and π returned in both worlds, giving no distinguishing information to the environment. The following parts of the invariant are affected and preserved:

(2) By equal update.
(3) By equal insertion into $\mathcal{F}^{\mathcal{L},r}_{\mathsf{nizk}}.W$, and $\mathcal{S}.\mathcal{F}^{\mathcal{L}}_{\mathsf{nizk}}.W$.
(4) By relaxing the constraint.
(9) As the possible results of executing transactions consisting of unrecorded statement/proof pairs is constrained – the environment can no longer decide if they should be processed or not.
(11) As in (9).

(12) As in (9).
(15) As only members of $\mathcal{L}$ are recorded.
(16) As in (9).

Case (VERIFY, x, π). The flow for verification is only slightly more complex than that for proving. At a high level, the adversary may be given a chance to produce a last-moment witness for the statement being verified. If it refuses to do so, the proof is recorded as definitively invalid. We consider three sub-cases: 1. The statement/proof pair is recorded as either valid or invalid. 2. The adversary returns a valid witness. 3. The adversary does not return a valid witness.

In Case 1, VERIFY returns the same value in both worlds by (2), giving the environment no means to distinguish. Case 2 is equivalent to the adversary first sending a PROVE query for the given statement, and supplying it with the corresponding proof, and then running the VERIFY query. We therefore refer to Case 1, and the case of PROVE. In Cases 1 and 2, no state is changed, preserving $\boldsymbol{I}$. Finally, for Case 3, $\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L},r}.\overline{\Pi}$ is updated equally in both worlds, and $\perp$ is returned in both worlds, giving the environment no information to distinguish. In this case, the following parts of the invariant are affected and preserved:

(2) By equal update.
(7) By relaxing the condition on $\mathcal{F}_{\mathsf{nizk}}^{\mathcal{L},r}.\overline{\Pi}$.
(9) As the possible results of executing transactions consisting of unrecorded statement/proof pairs is constrained – the environment can no longer decide if they should be processed or not.
(11) As in (9).
(12) As in (9).
(16) As in (9). □

As the environment cannot with non-negligible probability between the real and ideal world in any single action if $\boldsymbol{I}$ is preserved, and as $\boldsymbol{I}$ is preserved with overwhelming probability across each action by the environment, and holds at protocol initialization, we conclude that the environment cannot distinguish in the UC game. □

E Proof Sketch of the Zerocash Contract

Proof (sketch of Theorem 2). To begin with, observe that from the collision resistance of PRFs, commitments, and sampling from $\{0,1\}^{\kappa}$, all coin commitments, serial numbers, and public keys will be unique with overwhelming probability.

The environment can perform the following primary actions: a) For any honest party, run (POST-QUERY, w). b) For any honest party, run (CHECK-QUERY, τ). c) For any party, run (SUBMIT, τ) against $\mathcal{G}_{\mathsf{L}}$. d) For any party, run READ against $\mathcal{G}_{\mathsf{L}}$. e) Run (ADVANCE, p, Σ') against $\mathcal{G}_{\mathsf{L}}$, and e) for any party, run (EXTEND, Σ') against $\mathcal{G}_{\mathsf{L}}$.

All but the first two of these are trivial. The simulator forwards all queries to $\mathcal{G}_{\mathsf{L}}$, and the state of $\mathcal{G}_{\mathsf{L}}$ depends on no other functionality (transactions "submitted" in the ideal functionality are only passed to the adversary). As a direct result, the state and return value of $\mathcal{G}_{\mathsf{L}}$ follow the same distribution in both worlds, giving the environment no means to distinguish.

During the running of CHECK-QUERY the environment does have a significant additional means of input, in the form of being able to assign meaning to adversarial transactions *as they get executed for the first time*. It is sufficient to show the following: a) From the ideal-world leakage, the simulator can create indistinguishable real-world leakage. b) Ideal-world transactions have the same leakage descriptions sent to the environment (and are rejected under the same conditions). c) An invariant holds between the ideal and real-world contract state, such that it is preserved across both honest and adversarial transactions' transition function executions.

We omit the full detail of this invariant. To sketch the idea behind it, we must prove that the following are preserved: The public keys recorded in the ideal contract state, and the simulator must correspond directly to secret keys recorded in the real contract state, and the same public keys returned by the real contract. Further, the coins held by honest parties in the real contract should be valid at any time, and correspond directly to the balance of the same party in the ideal contract. Honest unconfirmed transactions in both the real and ideal contracts should still be valid when they are finally executed (also implying they do not conflict with each other).

These are preserved across honest INIT calls, as the simulator ensures the keys it stores, and the public keys returned in the ideal contract, are generated in the same way as in the real contract. They are preserved across honest SEND calls, as they remove one commitment from an honest party's coins, and potentially add it to the respective recipient party. Further, the leakage functions of honest SENDs in the real contract ensures the same coin cannot be spend again. They are preserved across honest MINTs, as again the balance is incremented alongside a new coin being recorded. For adversarial transactions, as the simulator has all honest private keys, it can, and does, check if an honest party would register receiving a new coin. If a coin is sent, but no honest party receives it, the simulator records it as adversarial – even if it may not be spendable by the real-world adversary. Further, the simulator manages which real-world coin commitments are associated with which adversarial public key in the ideal world. This ensures it can always spend a corresponding ideal coin to whatever was spent in the real world (assuming the real world adversary doesn't spend a coin he doesn't own, violating the one-wayness of the PRF).

Transactions remaining valid in the ideal world is guaranteed by ensuring the balance of a party cannot fall below zero – by assuming the worst case of only balance removing transactions becoming confirmed. Likewise, in the real world, the coins eligible for spending are those received in confirmed transactions, but not spent in unconfirmed ones, ensuring they will not conflict. In both cases, key

generation will be refused if one is currently unconfirmed. MINT and BALANCE queries both only require initialisation to have taken place in either world.

To observe that the simulator creates indistinguishable leakage, we first note that the leakage for real-world INIT transactions is an empty transcript, which the simulator indeed recreates. For SEND transactions, the simulator creates a public state transcript following the same structure of one in the real contract execution – spending a coin, creating a new one, and sending a message. Here there are two cases: either the recipient is adversarial, or it is honest. In the case of an honest recipient, the simulator does not know the exact public key of the recipient. Fortunately, however, the environment does not know their secret keys for the same reason. As a result, it is sufficient to commit to an arbitrary coin, and encrypt arbitrary secrets. Due to the hiding of the commitments, and the key-privacy of the encryption scheme, the environment cannot distinguish this from a real transaction. The simulator creates a random serial number – revealing nothing due to collision resistance, and from the leakage of the length of the ledger, can reconstruct the corresponding Merkle tree root, revealing the same root as the corresponding real-world transaction.

If the adversary *is* the recipient, the simulator is given the actual public keys – and can use these directly as in the real protocol, creating a valid spendable commitment, and a message the adversary can decrypt. Minting is similar to the case of sending to an honest party – except no message is encrypted. For the same reason, the leakage is indistinguishable. Finally, honest balance queries have no leakage in the real world.

For honest parties, the leakage descriptor the environment is asked to sign off on is identical – for INIT, MINT, and BALANCE consisting of just this string, and for SEND, it is $(\text{SEND}, t, \mathsf{pk})$, where pk is the recipient, if it is adversarial, and otherwise is omitted. In each case, assertions made about the current and projected states are satisfied in either both worlds, or neither, ensuring the transaction is rejected or posted equally in both worlds. Specifically, all have tests for whether keys are initialised (asserting negatively in INIT, and positively everywhere else). During spending, a positive spendable balance is also asserted in both worlds. These holding simultaneously is guaranteed by the invariant holding.

Finally, the transaction outputs the environment receives are the same in both worlds: For INIT, the simulator ensures it sees equally distributed public keys. For BALANCE, the equal distribution is guaranteed by the invariant. For all other messages, it will only see if they are in the ledger state – as the honest transactions cannot fail, and return nothing. □

F Liveness of Smart Contracts

We have presented KACHINA in a model without liveness guarantees. In this section, we discuss how to model, and prove KACHINA secure in the presence of such liveness guarantees. Similar principles can be used to argue for the security of KACHINA for further modifications to the ledger protocol, such as adding consensus-level validation predicates.

F.1 The δ-delay Ledger

While the simplified ledger $\mathcal{G}_{\mathsf{simpleLedger}}$ is nice for the analysis of protocols building on it as a global functionality, in practice users would like to take advantage of some liveness guarantees. We present a ledger $\mathcal{G}^{\delta}_{\mathsf{delayLedger}}$, which annotates transactions with a time at which they were received. This time is never returned to parties, however it asserts that every party can see a transaction, once it is δ time slots in the past. This ledger operates under the assumption of a global clock $\mathcal{G}_{\mathsf{clock}}$, which is also presented here. We posit without proof that $\mathcal{G}^{\delta}_{\mathsf{delayLedger}}$ UC-emulates $\mathcal{G}_{\mathsf{simpleLedger}}$, by virtue of the latter having far greater adversarial power.

Functionality $\mathcal{G}^{\delta}_{\mathsf{delayLedger}}$

The δ-delay ledger adds liveness guarantees to $\mathcal{G}_{\mathsf{simpleLedger}}$, ensuring that sufficiently old transactions are always visible to honest parties.

State variables and initialisation values:

Variable	Description
$\Sigma := \epsilon$	Authoritative ledger state
$M := \lambda p.\epsilon$	Mapping of parties to ledger states
$U := \varnothing$	Multiset of unconfirmed transactions

When receiving a message $(\textsc{submit}, \tau)$ *from a party* p*:*

send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t
let $U \leftarrow U \cup \{(\tau, t)\}$
query $\mathcal{A}$ **with** $(\textsc{transaction}, \tau, t)$

When receiving a message READ *from a party* p*:*

assert liveness
return $\mathsf{map}(\mathsf{proj}_1, M(p))$

When receiving a message $(\textsc{extend}, \Sigma')$ *from* $\mathcal{A}$*:*

if $\Sigma' \subseteq U$ **then**
  let $U \leftarrow U \setminus \Sigma'$
  let $\Sigma \leftarrow \Sigma \parallel \Sigma'$

When receiving a message $(\textsc{advance}, p, \Sigma')$ *from* $\mathcal{A}$*:*

if $M(p) \prec \Sigma' \prec \Sigma$ **then**
  let $M(p) \leftarrow \Sigma'$.

Helper procedures:

procedure liveness
  send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t
  if $\exists(\tau, t') \in U : |t - t'| > \delta$ **then return** $\bot$
  else if $\exists(\tau, t') \in \Sigma : |t - t'| > \delta \wedge \exists p \in \mathcal{H} : (\tau, t') \notin M(p)$ **then return** $\bot$
  else return $\top$

Functionality $\mathcal{G}_{\mathsf{clock}}$

The global clock allows parties to agree on some discrete notion of time.

State variables and initialisation values:

Variable	Description
$t := 0$	Current time
$T := \varnothing$	Timekeepers
$A := \varnothing$	Agreements to advance

When receiving a message REGISTER *from a party p:*
  let $T \leftarrow T \cup \{p\}$
When receiving a message DEREGISTER *from a party p:*
  let $T \leftarrow T \setminus \{p\}$
When receiving a message UPDATE *from a party p:*
  let $A(p) \leftarrow \top$
  if $\forall p \in T.A(t)$ **then**
    let $t \leftarrow t + 1; A \leftarrow \lambda p.\bot$
    query $\mathcal{A}$ **with** TICK-TOCK
When receiving a message READ *from a party p:*
  return t

F.2 Commutativity of Ledger Realisations

It is of note that since the ledger exists in both the ideal and real world, we would ideally wish to be able to utilise the stronger δ-delay ledger (and others) in the ideal world as well. This is not trivial, however – the UC proof presented in this paper holds for the simple ledger, and while the transitivity and composability of UC proofs implies that the simple ledger can be replaced by the stronger ledger in the real world, this is not the only goal.

In order to enable ideal-world replacement, we consider when UC replacements are commutative. Specifically, consider we have four functionalities, $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$, such that: a) $\mathcal{A}$ and $\mathcal{B}$ both have $\mathcal{C}$ as a global functionality, b) $\mathcal{A}$ is UC-emulated by $\mathcal{B}$ with the simulator $\mathcal{S}_{\mathcal{B}}$, and c) $\mathcal{C}$ is UC-emulated by $\mathcal{D}$ with the simulator $\mathcal{S}_{\mathcal{D}}$. Observe that this is a generalisation of our situation, where $\mathcal{A}$ is $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$, $\mathcal{B}$ is KACHINA, $\mathcal{C}$ is $\mathcal{G}_{\mathsf{simpleLedger}}$, and $\mathcal{D}$ is some other ledger $\mathcal{G}_{\mathsf{L}}$.

When can we conclude that $\mathcal{A}$ is still UC-emulated by $\mathcal{B}$ even if the global functionality is replaced by $\mathcal{D}$ in both worlds? I.e. when can we perform the inner UC-replacement *first*, and still be able to perform the outer one?

Theorem 3. *Given $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{D}$, $\mathcal{S}_{\mathcal{B}}$, $\mathcal{S}_{\mathcal{C}}$ as defined in Subsection F.2, if $\mathcal{S}_{\mathcal{B}}$ forwards all adversarial queries to $\mathcal{C}$ unchanged, and makes no queries of its own to $\mathcal{C}$, then $\mathcal{A}$ is UC-emulated by $\mathcal{B}$ with the global functionality $\mathcal{D}$ in place of $\mathcal{C}$.*

Proof. We will provide this proof largely visually. The environment has a number of actions it can perform in any given world, in tandem with the dummy adversary. We will represent these as unconnected wires in a circuit representation of the different UC functionalities. Each wire is coloured in accordance with its purpose; these colours serve only to differentiate the wires. We visualise the preconditions of Theorem 3 in Figure 4 and Figure 5. This crucially includes the precondition that $\mathcal{S}_{\mathcal{B}}$ forwards adversarial queries to $\mathcal{C}$, which is represented equivalently by these queries being made directly instead.

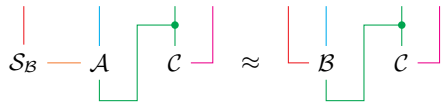

Fig. 4. The first part of the precondition: $\mathcal{B}$ UC-emulates $\mathcal{A}$.

Fig. 5. The second part of the precondition: $\mathcal{D}$ UC-emulates $\mathcal{C}$.

By the UC emulation theorem, for all environments, executions with the simulator and the ideal-world protocol are equivalent to executions with the real-world protocol. Due to the all-quantification of the environment, we can replace any part of a circuit diagram which matches exactly one of the two sides of the equivalence with the other – this is the foundation of the compositionality in UC.

We first make use of this in the non-standard direction, of making our ideal-world protocol *more ideal*. Specifically, replace the right-hand side of Figure 5 with the left. We start similarly to the left-hand side of Figure 4, however using $\mathcal{D}$ instead of $\mathcal{C}$. Visually, Figure 6 demonstrates the result, which includes two independent simulators.

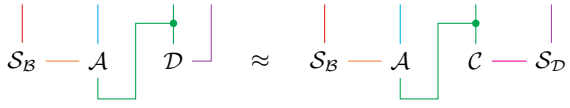

Fig. 6. Visual equivalence for idealising the sub-protocol $\mathcal{D}$.

From here, we can realise both the ideal-world functionalities, provided it is in the correct order: We must first realise $\mathcal{A}$, as it relies on the presence of $\mathcal{C}$. We can directly apply the equivalence of Figure 4, as can be seen in Figure 7.

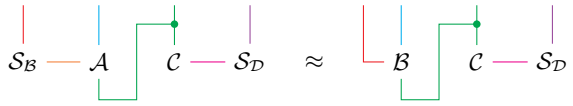

Fig. 7. Visual equivalence for substituting the main protocol $\mathcal{B}$.

Finally, nothing stands in the way of realising $\mathcal{C}$ with $\mathcal{D}$, using the equivalence of Figure 5 again, this time in the more typical direction. As a result, we get in Figure 8 the final step, leading us to the intended equivalence, and proving the related UC-emulation statement.

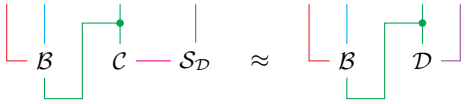

Fig. 8. Visual equivalence for re-substituting $\mathcal{D}$.

As a final remark, we observe that KACHINA satisfies the preconditions for commutativity, as $\mathcal{S}_{\text{KACHINA}}$ does not make adversarial queries to $\mathcal{G}_{\mathsf{ledger}}$.

G Enforcing Private State Consistency

The protocol presented so far allows an adversary to arbitrarily set his own private state. Often it may be desirable to ensure that parties must follow the rules of the contract, even when it comes to the private state, however. This is possible, although it also introduces extra costs, and has the caveat of not functioning with nondeterminism.

The core idea is to store commitments to private states within the public state of the contract. The contract itself can then verify that the private state is consistent with this commitment, update it, and then re-commit to the new state, proving the correctness of every step along the way. Clearly this adds more work to be verified about the contract, however a more worrying change is that again the contract needs to be able to process the entirety of the private contract state. Fortunately using slightly more complex updateable cryptographic datastructures, such as Merkle trees, can mitigate this problem – although it cannot be eliminated entirely, as computation which aggregates the entire private state will still be as costly.

H Non-Atomic Executions

Smart contracts are typically closely linked with transactions made on the underlying ledger, and indeed we explicitly make the same link in this paper. That being said, there are numerous applications which do not rely on a single transaction per interaction with a contract, from Hawk [22], which requires at least two transactions per round of interaction, to state channels [13], which have many of the same properties of smart contracts, but may (under optimal conditions) not require transactions at all.

While the model of smart contracts presented in Section 3 technically excludes both of these, and a full treatment of both would require further work, it is nonetheless worth considering how they can be – albeit imperfectly – embedded in this model. First, let us consider contract queries which require multiple on-chain interactions to "complete". As an example from Hawk, consider Alice posts a query to a Hawk-style contract. Naturally, this will not immediately return – even if Alice's transaction has made it on-chain. Instead, the transaction could return a "future object" – a concept often used in concurrent programming design, essentially just being a reference ID, and a promise to associate some data with it later. Both Alice and the manager party would have to regularly poll the contract – e.g. send a contract query POLL every 10 minutes. On the manager's next POLL query, he would update the Hawk private state, and encrypt and post the result for Alice. Finally, when Alice next polls, she would retrieve the result, and associate it with the previous "future object" as an output. This sketches a protocol running *on top of* KACHINA, which achieves this style of interaction. It is worth noting that this requirement for end-users to interact is also a limitation of the underlying model of universal composability: The environment must manually instruct parties to resume, or messages to be forwarded by the adversary.

In a similar vein, we can observe that some transactions need not be placed on a ledger. In particular, if the shared, public state is not used, the transaction is essentially "offchain", and there is no need to publicly post it. Further, if the public state is used for message passing (such as in the construction of a Zerocash contract above), this part of the transaction need not be on-chain – sending an out of band message is cheaper. Using the same UC-based approach described above, it would therefore be possible, for example, to first define an ideal payment-channel contract, and prove that this is UC-emulated by a contract implementing, for instance, Perun [12]. Finally, we can argue that most transactions in this contract can be omitted from the ledger, as they are just two-party channel interactions. This is a rather roundabout means of constructing off-chain communication, however it brings a crucial guarantee with it, namely that it behaves the same under ledger reorderings as a purely on-chain contract.

I Meta-Parties and Their Relation to Alternative Trust Models

So far we have presented, and argued for, a model of smart contracts with clear black-and-white privacy: Users have their own perfectly private local state, and access to a perfectly public shared state. While we believe this to be the best starting point for approaching the issue of privacy in smart contracts, reality is not so simple: Often users have more complex relations with each other.

To consider this more carefully, we can consider that any piece of data in a smart contract must have a set of *owners* $\mathbb{O}$, who can interact with it. Furthermore, in any real system, there are parties which can, together, decipher the actual data itself, and break the privacy of it. Let us refer to the set of all combinations of parties able to decipher the data as $\mathbb{T}$. While not strictly necessary, in general it is reasonable to assume that the owners are also the users able to break privacy, that is $\mathbb{T} \subseteq 2^{\mathbb{O}}$. While clearly there are many possible combinations here, a few stand out as interesting, and we observe that they all relate to some interpretation of privacy-preserving smart contracts:

- $\mathbb{O} = \mathcal{P}$, $\mathbb{T} = 2^{\mathcal{P}}$: This is the setting of Ethereum, and of the σ used in this work. Data is public, but can be interacted with by all.
- $\mathbb{O} = \{p\}$, $\mathbb{T} = \{p\}$: This is the setting of ρ used in this work. Data is private, but cannot be interacted with by anyone else.
- $\mathbb{O} = \mathcal{P}$, $\mathbb{T}$ is all subsets of $\mathcal{P}$ with a resource majority (regardless of work, stake, or what other honest majority assumption is being made): This setting is feasible by running MPC across the honest majority of the underlying consensus protocol.
- $\mathbb{O}$ is a fixed-size set, $\mathbb{T} = \{m\}$: This is the setting of Hawk [22], in which a single party is trusted with privacy.
- $\mathbb{O}$ is a fixed-size set, $\mathbb{T} = \{\mathbb{O}\}$: This is the setting of privacy-preserving state-channels, in which parties run MPC out-of-band to agree on updates.
- $\mathbb{O}$ is a fixed-size set, $\mathbb{T} = 2^{\mathbb{O}}$: This is the setting of public state-channels, in which parties run Arbitrum-like protocols out-of-band to agree on updates.

In particular, this work only directly concerns itself with the first two of these. It is clear, however that different problems call for different solutions, and ideally a smart contract system would encode all of these trust systems, not just one, or a few. Part of the reason for the choice of the first two is that they are sufficient for constructing the rest, being the exteremes of the spectrum.

The case of Hawk, for instance, was already described above in Appendix H. We will sketch how state channels might be modelled on top of Kachina, although we stress that a full formal treatment of this, and other settings, will be left for future work.

A state channel between two users can be interpreted as the two users constituting a "metaparty" – a single entity consisting of multiple parties. This is subject to some access control for when the constituent parties can act on behalf of both – commonly requiring agreement from all constituent parties. If Alice

and Bob open a new state channel, this can be seen as creating a new combined party of (Alice, Bob).

In Kachina, this party again has its own private state, and for state channels, this can track the most recent update of the channel. Updates are now operations that only affect the private state of this combined party, and as argued in Appendix H can be left off the ledger entirely. Interestingly, the access structure for closing channels, and reading the current state is more permissive in most state or payment channels – requiring only one user to initiate it.

Given a state channel system, most of it can be implemented in a Kachina smart contract. It is not new for state or payment channels to use smart contracts, however this is typically only for the opening and closing of the channel. We observe that in Kachina the update of the channel can also be modelled.

This approach of metaparties is useful, but not optimal. For instance, a contract cannot interact with both Alice's private state, and the state channel between Alice and Bob at the same time, as presented here. Further, how the constituents of a metaparty reach consensus on whether an action is permitted or not is unclear, and varies from case to case. We leave as future work giving first-class treatment to data owned by multiple parties.

J Smart Contract Systems

To construct complex systems of multiple smart contracts, not much additional machinery is required. In this section, we incrementally construct a complex system with similar functionality to Ethereum [31]. We begin by multiplexing between a fixed set of transition functions, and expand this with the ability to allow new transition functions to be registered, transition functions to call each other, registered contracts to hold and transfer funds, and combine in a setting where computation has an associated cost, which must be paid by the caller. We finally show how access to the underlying ledger may be modelled.

It is worth noting that we only concern ourselves with the "real world" of Kachina contracts. A reasonable question is how to transfer a proof such as the one we presented in Section 5 into this setting. While we don't go into the details here, we observe that (with one exception for the specific token contract used), only the smart contract's own transition function affects its state. Running a multiplexed smart contract is equivalent to running many small smart contracts independently – only interpreting the ledger differently. This is no longer true once contracts may call each other – in which case it is sufficient to reason about the closure of contracts able to call each other instead.

J.1 Multiplexing Contracts

The basic multiplexing contract takes n different sub-contracts as inputs. Each party supplies not only the input, but the index i of the contract they wish to call. The public and private states of the multiplexer consist of the product of the corresponding sub-contract states, and oracle queries are re-written to

address the correct part of the state. To do some, new oracles $\mathcal{O}'_\sigma$ and $\mathcal{O}'_\rho$ are constructed, which rewrite queries made to them. Then, the requested transition function is run with these oracles, instead of the original ones.

Transition Function Γ_{mux}

The multiplexing transition function Γ_{mux} is parameterised by n transition functions $\Gamma_1, \ldots, \Gamma_n$, and allows a users to address any one of them.

Public state variables and initialisation values:

Variable	Description
$\forall i \in \mathbb{Z}_n : \sigma_i := \varnothing$	Public states for each sub-contract

---

Private state variables and initialisation values:

Variable	Description
$\forall i \in \mathbb{Z}_n : \rho_i := \varnothing$	Private states for each sub-contract

When receiving an input (i, w):

assert $i \in \mathbb{Z}_n$
let $\mathcal{O}'_\sigma \leftarrow \lambda q : \mathcal{O}_\sigma(\mathsf{muxPubOracle}(i, q))$
let $\mathcal{O}'_\rho \leftarrow \lambda q : \mathcal{O}_\rho(\mathsf{muxPrivOracle}(i, q))$
return $\Gamma_{i,\mathcal{O}'_\sigma,\mathcal{O}'_\rho}(w)$

Helper procedures:

function $\mathsf{muxPubOracle}(i, q, \sigma, \varnothing)$
  let $\sigma' \leftarrow \sigma.\sigma_i$
  let $(\sigma', y) \leftarrow q(\sigma', \varnothing)$
  if $\sigma' = \bot$ **then return** $(\bot, y)$
  else
    let $\sigma.\sigma_i \leftarrow \sigma'$
    return (σ, y)
function $\mathsf{muxPrivOracle}(i, q, \rho, (\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta))$
  let $\rho' \leftarrow \rho.\rho_i$
  let $z' \leftarrow (\sigma^o.\sigma_i, \rho^o.\rho_i, \sigma^\pi.\sigma_i, \rho^\pi.\rho_i, \eta)$
  let $(\rho', y) \leftarrow q(\rho', z')$
  if $\rho' = \bot$ **then return** $(\bot, y)$
  else
    let $\rho.\rho_i \leftarrow \rho'$
    return (ρ, y)

We assume the existence of $\mathsf{unmuxPubOracle}$ and $\mathsf{unmuxPrivOracle}$, which take an oracle transcript to an Oracle produced by a multiplexed oracle, and return the pair $(i, \mathcal{T}')$, where i is the address used in the original multiplexing, and $\mathcal{T}'$ is the equivalent un-multiplexed transcript.

function $\mathsf{unmuxZ}_{\mathsf{mux}}((\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta), i)$

return $(\sigma^o.\sigma_i, \rho^o.\rho_i, \sigma^\pi.\sigma_i, \rho^\pi.\rho_i, \eta)$

function $\mathsf{unmuxX}_{\mathsf{mux}}(X, i)$
  let $X' \leftarrow \epsilon$
  for $(u, \mathcal{T}, z, D) \in X$ **do**
    if $\exists \mathcal{T}'$: $\mathsf{unmuxPrivOracle}(\mathcal{T}) = (i, \mathcal{T}')$ **then**
      let $X' \leftarrow X' \parallel (u, \mathcal{T}', \mathsf{unmuxZ}_{\mathsf{mux}}(z, i), D)$
  return X'

function $\mathsf{desc}_{\mathsf{mux}}(t, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, (i, w), z)$
  let $(\cdot, \mathcal{T}'_\sigma) \leftarrow \mathsf{unmuxPubOracle}(\mathcal{T}_\sigma); (\cdot, \mathcal{T}'_\rho) \leftarrow \mathsf{unmuxPrivOracle}(\mathcal{T}_\rho)$
  let $X' \leftarrow \mathsf{unmuxX}_{\mathsf{mux}}(X, i); z' \leftarrow \mathsf{unmuxZ}_{\mathsf{mux}}(z, i)$
  return "Calling sub-contract i: "+$\mathsf{desc}_i(t, X', \mathcal{T}'_\sigma, \mathcal{T}'_\rho, w, z')$

function $\mathsf{dep}_{\mathsf{mux}}(X, \mathcal{T}_\rho, z)$
  if $\mathcal{T}_\rho = \epsilon$ **then return** $\varnothing$
  else
    let $(i, \mathcal{T}'_\rho) \leftarrow \mathsf{unmuxPrivOracle}(\mathcal{T}_\rho)$
    let $X' \leftarrow \mathsf{unmuxX}_{\mathsf{mux}}(X, i); z' \leftarrow \mathsf{unmuxZ}_{\mathsf{mux}}(z, i)$
    return $\mathsf{dep}_i(X', \mathcal{T}'_\rho, z')$

J.2 Multiplexing With Registration

To allow registering new contracts in the multiplexer, it is possible to include the full contract's description as part of its *address* A. In practice it may make more sense to maintain a mapping from addresses to contract code, however we observe that this is not required. The only other large changes is that, since contracts are created on the fly, we cannot rely on their states to have been initialised at any point. Therefore, this initialisation takes place at any point where the multiplexed state is accessed.

function $\mathsf{forceInitMaps}(((M_1, \ldots, M_n), k, v))$
  for $i \in \{1, \ldots, n\}$ **do**
    if $k \notin M_i$ **then let** $M_i(k) \leftarrow v$
  return $(M_1, \ldots, M_n)$

Transition Function Γ_{regmux}

The multiplexing with registration transition function Γ_{regmux} allows addressing any pair of address and sub-transition function (A, Γ). It uses the specified transition function on whatever state is associated with this pair, or a new, empty state for the first use.

Public state variables and initialisation values:

Variable	Description
$\Sigma := \varnothing$	Mapping from address pairs to public states

Private state variables and initialisation values:

Variable	Description
$P := \varnothing$	Mapping from address pairs to private states

When receiving an input $(A = (i, \Gamma, \mathsf{desc}, \mathsf{dep}), w)$*:*

let $\mathcal{O}'_\sigma \leftarrow \lambda q : \mathcal{O}_\sigma(\mathsf{muxPubOracle}(A, q))$
let $\mathcal{O}'_\rho \leftarrow \lambda q : \mathcal{O}_\rho(\mathsf{muxPrivOracle}(A, q))$
return $\Gamma_{\mathcal{O}'_\sigma, \mathcal{O}'_\rho}(w)$

Helper procedures:

function $\mathsf{muxPubOracle}(A, q, \sigma, \varnothing)$
  if $A \notin \sigma.\Sigma$ **then let** $\sigma.\Sigma(A) \leftarrow \varnothing$
  let $\sigma' \leftarrow \sigma.\Sigma(A)$
  let $(\sigma', y) \leftarrow q(\sigma', \varnothing)$
  if $\sigma' = \bot$ **then return** $(\bot, y)$
  else
    let $\sigma.\Sigma(A) \leftarrow \sigma'$
    return (σ, y)

function $\mathsf{muxPrivOracle}(A, q, \rho, (\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta))$
  let $(\rho.P, \sigma^o.\Sigma.\rho^o.P, \sigma^\pi.\Sigma, \rho^\pi.P) \leftarrow$
    $\mathsf{forceInitMaps}((\rho.P, \sigma^o.\Sigma, \rho^o.P, \sigma^\pi.\Sigma, \rho^\pi.P), A, \varnothing)$
  let $\rho' \leftarrow \rho.P(A)$
  let $z' \leftarrow (\sigma^o.\sigma_i, \rho^o.\rho_i, \sigma^\pi.\sigma_i, \rho^\pi.\rho_i, \eta)$
  let $(\rho', y) \leftarrow q(\rho', z')$
  if $\rho' = \bot$ **then return** $(\bot, y)$
  else
    let $\rho.P(A) \leftarrow \rho'$
    return (ρ, y)

We assume the existence of $\mathsf{unmuxPubOracle}$ and $\mathsf{unmuxPrivOracle}$, which take an oracle transcript to an Oracle produced by a multiplexed oracle, and return the pair $(A, \mathcal{T}')$, where $A = (i, \Gamma, \mathsf{desc}, \mathsf{dep})$ is the address used in the original multiplexing, and $\mathcal{T}'$ is the equivalent un-multiplexed transcript.

function $\mathsf{unmuxZ}_{\mathsf{regmux}}((\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta), A)$
  let $(\sigma^o.\Sigma, \rho^o.P, \sigma^\pi.\Sigma, \rho^\pi.P) \leftarrow \mathsf{forceInitMaps}((\sigma^o.\Sigma, \rho^o.P, \sigma^\pi.\Sigma, \rho^\pi.P), A, \varnothing)$
  return $(\sigma^o.\Sigma(A), \rho^o.P(A), \sigma^\pi.\Sigma(A), \rho^\pi.P(A), \eta)$

function $\mathsf{unmuxX}_{\mathsf{regmux}}(X, A)$
  let $X' \leftarrow \epsilon$
  for $(u, \mathcal{T}, z, D) \in X$ **do**
    if $\exists \mathcal{T}' : \mathsf{unmuxPrivOracle}(\mathcal{T}) = (A, \mathcal{T}')$ **then**
      let $X' \leftarrow X' \parallel (u, \mathcal{T}', \mathsf{unmuxZ}_{\mathsf{regmux}}(z, A), D)$
  return X'

function $\mathsf{desc}_{\mathsf{regmux}}(t, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, (A = (\cdot, \cdot, \mathsf{desc}, \cdot), w), z)$
  let $(\cdot, \mathcal{T}'_\sigma) \leftarrow \mathsf{unmuxPubOracle}(\mathcal{T}_\sigma); (\cdot, \mathcal{T}'_\rho) \leftarrow \mathsf{unmuxPrivOracle}(\mathcal{T}_\rho)$
  let $X' \leftarrow \mathsf{unmuxX}_{\mathsf{regmux}}(X, A); z' \leftarrow \mathsf{unmuxZ}_{\mathsf{regmux}}(z, A)$
  return "Calling sub-contract A: "+$\mathsf{desc}(t, X', \mathcal{T}'_\sigma, \mathcal{T}'_\rho, w, z')$

function $\mathsf{dep}_{\mathsf{regmux}}(X, \mathcal{T}_\rho, (\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta))$
  if $\mathcal{T}_\rho = \epsilon$ **then return** $\varnothing$

else
  let $(A = (\ldots, \mathsf{dep}), \mathcal{T}'_\rho) \leftarrow \mathsf{unmuxPrivOracle}(\mathcal{T}_\rho)$
  let $(\sigma^o.\Sigma.\rho^o.P, \sigma^\pi.\Sigma, \rho^\pi.P) \leftarrow$
    $\mathsf{forceInitMaps}((\sigma^o.\Sigma, \rho^o.P, \sigma^\pi.\Sigma, \rho^\pi.P), A, \varnothing)$
  let $z' \leftarrow (\sigma^o.\Sigma(A), \rho^o.P(A), \sigma^\pi.\Sigma(A), \rho^\pi.P(A), \eta)$
  let $X' \leftarrow \epsilon$
  for $(u, \mathcal{T}_\rho, (\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta), D) \in X$ **do**
    if $\exists \mathcal{T}'_\rho : \mathsf{unmuxPrivOracle}(\mathcal{T}_\rho) = (A, \mathcal{T}'_\rho)$ **then**
      let $(\sigma^o.\Sigma.\rho^o.P, \sigma^\pi.\Sigma, \rho^\pi.P) \leftarrow$
        $\mathsf{forceInitMaps}((\sigma^o.\Sigma, \rho^o.P, \sigma^\pi.\Sigma, \rho^\pi.P), A, \varnothing)$
      let $X' \leftarrow X' \parallel (u, \mathcal{T}'_\rho, (\sigma^o.\sigma_i, \rho^o.\rho_i, \sigma^\pi.\sigma_i, \rho^\pi.\rho_i, \eta), D)$
  return $\mathsf{dep}(X', \mathcal{T}'_\rho, z')$

J.3 Loopback Multiplexing

Smart contract systems truly become interesting when contracts are allowed to *call each other*. This is not a technically difficult operation: Contracts simply need to have an additional exit and entry point to allow new queries to other contracts to be made, and these queries to be responded to. Specifically, we require contracts to *either* return $(\textsc{return}, y)$, *or* $(\textsc{call}, A, M)$, with the latter invoking a separate contract. we associate a special return value structure with indicating a new contract address and input to call, and require contracts to process a specific RESUME message.

As for the first time, it is possible for multiple separate contracts to get called, we domain-separate the randomness source η.

function $\mathsf{unxmuZ}_{\mathsf{loopmux}}((\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta), A)$
  let $(\sigma^o.\Sigma, \rho^o.P, \sigma^\pi.\Sigma, \rho^\pi.P) \leftarrow \mathsf{forceInitMaps}((\sigma^o.\Sigma, \rho^o.P, \sigma^\pi.\Sigma, \rho^\pi.P), A, \varnothing)$
  Let η' be a randomness source determinstically and collision-resistantly derived from the pair (η, A).
  return $(\sigma^o.\Sigma(A), \rho^o.P(A), \sigma^\pi.\Sigma(A), \rho^\pi.P(A), \eta')$

Transition Function $\Gamma_{\mathsf{loopmux}}$

The multiplexing with registration and loopback transition function $\Gamma_{\mathsf{loopmux}}$ allows addressing any pair of address and sub-transition function (A, Γ). These sub-transition functions may, return values of either $(\textsc{call}, A, M)$, or $(\textsc{return}, y)$. In the former case, a different sub-transition function is invoked, and the value it eventually returns is fed back into the original one, by re-invoking it with $(\textsc{resume}, y)$.

Public state variables and initialisation values:

Variable	Description
$\Sigma := \varnothing$	Mapping from address pairs to public states

Private state variables and initialisation values:

Variable	Description
$P := \varnothing$	Mapping from address pairs to private states

When receiving an input $(A = (i, \Gamma, \mathsf{desc}, \mathsf{dep}), w)$*:*

let $\mathcal{O}'_\sigma \leftarrow \lambda q : \mathcal{O}_\sigma(\mathsf{muxPubOracle}(A, q))$
let $\mathcal{O}'_\rho \leftarrow \lambda q : \mathcal{O}_\rho(\mathsf{muxPrivOracle}(A, q))$
repeat
  let $y \leftarrow \Gamma_{\mathcal{O}'_\sigma, \mathcal{O}'_\rho}(w)$
  if $\exists A', M : y = (\textsc{call}, A', M)$ **then**
    let $w \leftarrow \left(\textsc{resume}, \Gamma_{\mathsf{loopmux}, \mathcal{O}_\sigma, \mathcal{O}_\rho}((A', M))\right)$
until $\exists y' : y = (\textsc{return}, y')$
return y'

Helper procedures:

function $\mathsf{muxPubOracle}(A, q, \sigma, \varnothing)$
  if $A \notin \sigma.\Sigma$ **then let** $\sigma.\Sigma(A) \leftarrow \varnothing$
  let $\sigma' \leftarrow \sigma.\Sigma(A)$
  let $(\sigma', y) \leftarrow q(\sigma', \varnothing)$
  if $\sigma' = \bot$ **then return** $(\bot, y)$
  else
    let $\sigma.\Sigma(A) \leftarrow \sigma'$
    return (σ, y)
function $\mathsf{muxPrivOracle}(A, q, \rho, z)$
  let $z' \leftarrow \mathsf{unmuxZ}_{\mathsf{loopmux}}(z, A)$
  if $A \notin \rho.P$ **then let** $\rho.P(A) \leftarrow \varnothing$
  let $\rho' \leftarrow \rho.P(A)$
  let $(\rho', y) \leftarrow q(\rho', z')$
  if $\rho' = \bot$ **then return** $(\bot, y)$
  else
    let $\rho.P(A) \leftarrow \rho'$
    return (ρ, y)

Unlike before, we cannot invert the multiplexing on an entire transcript, as the transcript may consist of multiple, separate, sub-contract calls. Instead, we can invert multiplexing each query/response pair in the transcript itself. We assume the existence of $\mathsf{unmuxOracle}$, which take a query-response pair (q, r), where the query is $\mathsf{muxPubOracle}(A, q')$ or $\mathsf{muxPrivOracle}(A, q')$, and maps it to $(A, (q', r))$.

As far as descriptions go, it is crucial to note that the leakage description of a contract is no longer in isolation: what the contract may leak, depends on what this contract calls. We will assume instead that each sub-contracts leakage descriptor is aware that it is being run in a loopback system – and therefore we give it the full transcripts, even of sub-contracts being called. The assumption here is that the contract directly called by the user is also trusted by this user – the descriptor it gives should be trusted, not necessarily that of any further contracts it invoked. It is worth noting that this change of setting for the descriptor function does not preclude using contracts designed without loopback systems

in mind: As this cannot invoke other contracts, their old descriptor function can be easily lifted to this setting, as seen in Subsection J.2 (a slight caveat is that either the old descriptor needs to be capable of tolerating unconfirmed transaction transcripts over multiple calls to the underlying function, or there should exist a function which splits transcripts into these individual calls).

function $\mathsf{liftDesc}(A, \mathsf{desc})(t, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, w, z)$
  let $\mathcal{T}'_\sigma \leftarrow \mathsf{map}(\mathsf{proj}_2 \circ \mathsf{unmuxOracle}, \mathcal{T}_\sigma)$
  let $\mathcal{T}'_\rho \leftarrow \mathsf{map}(\mathsf{proj}_2 \circ \mathsf{unmuxOracle}, \mathcal{T}_\rho)$
  let $X' \leftarrow \mathsf{unmuxX}_{\mathsf{regmux}}(X, A); z' \leftarrow \mathsf{unmuxZ}_{\mathsf{regmux}}(z, A)$
  return $\mathsf{desc}(t, X', \mathcal{T}'_\sigma, \mathcal{T}'_\rho, w, z')$

function $\mathsf{unmuxT}(\mathcal{T}, A)$
  return $\mathsf{map}(\mathsf{proj}_2, \mathsf{filter}(\lambda(A', \cdot) : A = A', \mathsf{map}(\mathsf{unmuxOracle}, \mathcal{T})))$

function $\mathsf{unmuxX}_{\mathsf{loopmux}}(X, A)$
  let $X' \leftarrow \epsilon$
  for $(u, \mathcal{T}, z, D) \in X$ **do**
    let $\mathcal{T}' \leftarrow \mathsf{unmuxT}(\mathcal{T}, A); z' \leftarrow \mathsf{unmuxZ}_{\mathsf{loopmux}}(z, A)$
    let $X' \leftarrow X' \parallel (u, \mathcal{T}', z', D')$
  return X'

function $\mathsf{desc}_{\mathsf{loopmux}}(t, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, (A = (\cdot, \cdot, \mathsf{desc}, \cdot), w), z)$
  return "Calling sub-contract A: "+$\mathsf{desc}(t, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, w, z)$

function $\mathsf{dep}_{\mathsf{loopmux}}(X, \mathcal{T}_\rho, z)$
  let $S \leftarrow \varnothing$
  for $(q, r) \in \mathcal{T}_\rho$ **do**
    let $(A, \cdot) \leftarrow \mathsf{unmuxOracle}((q, r))$
    let $S \leftarrow S \cup \{A\}$
  let $D \leftarrow \varnothing$
  for $A = (\cdot, \cdot, \cdot, \mathsf{dep}) \in S$ **do**
    let $\mathcal{T}'_\rho \leftarrow \mathsf{unmuxT}(\mathcal{T}_\rho, A)$
    let $z' \leftarrow \mathsf{unmuxZ}_{\mathsf{loopmux}}(z, A)$
    let $X' \leftarrow \mathsf{unmuxX}_{\mathsf{loopmux}}(X, A)$
    let $D \leftarrow D \cup \mathsf{dep}(X', \mathcal{T}'_\rho, z')$
  return $\mathsf{map}(\mathsf{proj}_1, X) \cap D$

J.4 Integrated Payments Systems

Smart contract systems typically have an associated, native "asset", which can be traded not only by users, but by contracts as well. This asset is further typically tied to a public key, which can be used as an identity of end users, providing a means to authenticate to contracts. We demonstrate a simple means of achieving this: We construct a "simple payments" contract, which allows payments by end users through demonstrating knowledge of secret keys, and arbitrary payments which will be restricted to system usage. It is worth noting that this *could* be done in a privacy-preserving means, as presented in Section 5, although significant changes would have to be made, as there would be situations where a contract should publicly own funds, and be able to transfer them, and the simplified single-denomination design is not ideal.

Transition Function Γ_{sp}

The state transition function for a simple payments system. Parties have associated public/private keys, and balances. The payments system allows for parties without a key pair to generate one, and for parties to transfer and mint coins, as well as query their own balance.

Public state variables and initialisation values:

Variable	Description
$B := \lambda \mathsf{pk} : 0$	Mapping of public keys to their spendable coins

---

Private state variables and initialisation values:

Variable	Description
$\mathsf{sk} := \varnothing$	The party's secret key

When receiving an input INIT*:*
send INIT **to** $\mathcal{O}_\rho$ and **receive the reply** sk
let $\mathsf{pk} \leftarrow \mathsf{prf}^{\mathsf{pk}}_{\mathsf{sk}}(1)$
return pk

When receiving an input $(\text{SEND}, \mathsf{recv}, v)$*:*
send SECRETKEY **to** $\mathcal{O}_\rho$ and **receive the reply** sk
let $\mathsf{pk} \leftarrow \mathsf{prf}^{\mathsf{pk}}_{\mathsf{sk}}(1)$
send $(\text{SEND}, \mathsf{pk}, \mathsf{recv}, v)$ **to** $\mathcal{O}_\sigma$
return pk

When receiving an input $(\text{SYSTEM-SEND}, \mathsf{snd}, \mathsf{recv}, v)$*:*
send $(\text{SEND}, \mathsf{snd}, \mathsf{recv}, v)$ **to** $\mathcal{O}_\sigma$

When receiving an input (MINT, v)*:*
send SECRETKEY **to** $\mathcal{O}_\rho$ and **receive the reply** sk
let $\mathsf{pk} \leftarrow \mathsf{prf}^{\mathsf{pk}}_{\mathsf{sk}}(1)$
send $(\text{MINT}, \mathsf{pk}, v)$ **to** $\mathcal{O}_\sigma$

When receiving an input BALANCE*:*
send BALANCE **to** $\mathcal{O}_\rho$

When receiving an private oracle query INIT*:*
assert $\rho^\pi.\mathsf{sk} = \varnothing$
let $\rho.\mathsf{sk} \xleftarrow{R} \{0,1\}^\kappa$
return $\rho.\mathsf{sk}$

When receiving an private oracle query SECRETKEY*:*
return $\rho.\mathsf{sk}$

When receiving an private oracle query BALANCE*:*
return $\sigma^\pi.B(\mathsf{prf}^{\mathsf{pk}}_{\rho^\pi.\mathsf{sk}}(1))$

When receiving an public oracle query $(\textsc{send}, \mathsf{pk}, \mathsf{recv}, v)$*:*
  assert $\sigma.B(\mathsf{pk}) \geq v$
  let $\sigma.B(\mathsf{pk}) \leftarrow \sigma.B(\mathsf{pk}) - v$
  let $\sigma.B(\mathsf{recv}) \leftarrow \sigma.B(\mathsf{recv}) + v$
When receiving an public oracle query $(\textsc{mint}, \mathsf{pk}, v)$*:*
  let $\sigma.B(\mathsf{pk}) \leftarrow \sigma.B(\mathsf{pk}) + v$

function $\mathsf{desc}_{\mathsf{sp}}(t, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, w, z)$
  if $\mathcal{T}_\sigma = (\textsc{init}, \mathsf{pk})$ **then return** $(\textsc{init}, \mathsf{pk})$
  else if $\mathcal{T}_\sigma = ((\textsc{send}, \mathsf{snd}, \mathsf{recv}, v), \cdot)$ **then return** $(\textsc{send}, \mathsf{snd}, \mathsf{recv}, v)$
  else if $\mathcal{T}_\sigma = ((\textsc{mint}, \mathsf{pk}, v), \cdot)$ **then return** $(\textsc{mint}, \mathsf{pk}, v)$
  else return $\bot$
function $\mathsf{dep}_{\mathsf{sp}}(X, \mathcal{T}, z)$
  return ϵ

Once given such a payments system, the multiplexing system can ensure that for each call, a transfer to the called contract is initiated *first*, with the value of the transfer, and the source address being passed into the contract being called. Likewise, if this calls another contract, this call may transfer funds from one contract to another.

Transition Function Γ_{paymux}

The multiplexing with registration, loopback, and payments transition function Γ_{paymux} allows addressing any pair of address and sub-transition function (a, Γ). These sub-transition functions may, return values of either $(\textsc{call}, A, M)$, or $(\textsc{return}, y)$. In the former case, a different sub-transition function is invoked, and the value it eventually returns is fed back into the original one, by re-invoking it with $(\textsc{resume}, y)$.

Public state variables and initialisation values:

Variable	Description
$\Sigma := \varnothing$	Mapping from address pairs to public states

Private state variables and initialisation values:

Variable	Description
$P := \varnothing$	Mapping from address pairs to private states

When receiving an input $(\textsc{token}, w)$*:*
  assert $w \neq (\textsc{system-send}, \ldots)$
  let $\mathcal{O}'_\sigma \leftarrow \lambda q : \mathcal{O}_\sigma(\mathsf{muxPubOracle}(\textsc{token}, q))$
  let $\mathcal{O}'_\rho \leftarrow \lambda q : \mathcal{O}_\rho(\mathsf{muxPrivOracle}(\textsc{token}, q))$
  return $\Gamma_{\mathsf{sp}, \mathcal{O}'_\sigma, \mathcal{O}'_\rho}(w)$
When receiving an input $(\textsc{call}, v, A = (i, \Gamma, \mathsf{desc}, \mathsf{dep}), w)$*:*

let $\mathsf{pk} \leftarrow \Gamma_{\mathsf{paymux},\mathcal{O}_\sigma,\mathcal{O}_\rho}(\textsc{token}, (\textsc{send}, A, v))$
return $\mathsf{call}_{\mathcal{O}_\sigma,\mathcal{O}_\rho}(v, \mathsf{pk}, A, w)$

Helper procedures:

function $\mathsf{subCall}_{\mathcal{O}_\sigma,\mathcal{O}_\rho}(v, A, A' = (i, \Gamma, \mathsf{desc}, \mathsf{dep}), w)$
  assert $A' \neq \textsc{token}$
  let $\mathcal{O}'_\sigma \leftarrow \lambda q : \mathcal{O}_\sigma(\mathsf{muxPubOracle}(\textsc{token}, q))$
  let $\mathcal{O}'_\rho \leftarrow \lambda q : \mathcal{O}_\rho(\mathsf{muxPrivOracle}(\textsc{token}, q))$
  run $\Gamma_{\mathsf{sp},\mathcal{O}'_\sigma,\mathcal{O}'_\rho}(\textsc{system-send}, A, A', v)$
  return $\mathsf{call}_{\mathcal{O}_\sigma,\mathcal{O}_\rho}(v, A, A', w)$

function $\mathsf{call}_{\mathcal{O}_\sigma,\mathcal{O}_\rho}(v, A, A' = (\cdot, \Gamma, \cdot, \cdot), w)$
  let $\mathcal{O}'_\sigma \leftarrow \lambda q : \mathcal{O}_\sigma(\mathsf{muxPubOracle}(A', q))$
  let $\mathcal{O}'_\rho \leftarrow \lambda q : \mathcal{O}_\rho(\mathsf{muxPrivOracle}(A', q))$
  repeat
    let $y \leftarrow \Gamma_{\mathcal{O}'_\sigma,\mathcal{O}'_\rho}(\textsc{call}, A, v, w)$
    if $\exists v', A'', w' : y = (\textsc{call}, v', A'', w')$ **then**
      let $w \leftarrow \big(\textsc{resume}, \mathsf{subCall}_{\mathcal{O}_\sigma,\mathcal{O}_\rho}(v', A', A'', w')\big)$
  until $\exists y' : y = (\textsc{return}, y')$
  return y'

function $\mathsf{muxPubOracle}(A, q, \sigma, \varnothing)$
  if $A \notin \sigma.\Sigma$ **then let** $\sigma.\Sigma(A) \leftarrow \varnothing$
  let $\sigma' \leftarrow \sigma.\Sigma(A)$
  let $(\sigma', y) \leftarrow q(\sigma', \varnothing)$
  if $\sigma' = \bot$ **then return** $(\bot, y)$
  else
    let $\sigma.\Sigma(A) \leftarrow \sigma'$
    return (σ, y)

function $\mathsf{muxPrivOracle}(A, q, \rho, (\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta))$
  let $(\rho.P, \sigma^o.\Sigma.\rho^o.P, \sigma^\pi.\Sigma, \rho^\pi.P) \leftarrow$
    $\mathsf{forceInitMaps}((\rho.P, \sigma^o.\Sigma, \rho^o.P, \sigma^\pi.\Sigma, \rho^\pi.P), A, \varnothing)$
  let $\rho' \leftarrow \rho.P(A)$
  let $z' \leftarrow (\sigma^o.\sigma_i, \rho^o.\rho_i, \sigma^\pi.\sigma_i, \rho^\pi.\rho_i, \eta)$
  let $(\rho', y) \leftarrow q(\rho', z')$
  if $\rho' = \bot$ **then return** $(\bot, y)$
  else
    let $\rho.P(A) \leftarrow \rho'$
    return (ρ, y)

---

function $\mathsf{desc}_{\mathsf{paymux}}(t, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, M, z)$
  if $\exists w : M = (\textsc{token}, w)$ **then**
    return "Calling token contract:"+$\mathsf{desc}_{\mathsf{sp}}(t, X, \mathsf{map}(\mathsf{proj}_2 \circ \mathsf{unmuxOracle}, \mathcal{T}_\sigma),$
    $\mathsf{map}(\mathsf{proj}_2 \circ \mathsf{unmuxOracle}, \mathcal{T}_\rho), w, z)$
  else if $\exists v, A = (\cdot, \cdot, \mathsf{desc}, \cdot), w : M = (\textsc{call}, v, A, w)$ **then**
    return "Calling sub-contract A with pay-in v: "+$\mathsf{desc}(t, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, (\bot, v, w), z)$
  else return $\bot$

$\mathsf{dep}_{\mathsf{paymux}} = \mathsf{dep}_{\mathsf{loopmux}}$

J.5 Fees and Cost Models

In order to prevent denial-of-service attacks, the computations performed by the network in verifying a transaction must be paid for in some way. In public currencies, there is typically a *cost model*, which maps each step of computation to a cost, often referred to as *gas*. Each transaction declares a limit on how much gas it is willing to pay, and what each unit's value should be. It then pays the corresponding amount into a fees pool, and while executing the transaction, the gas usage is counted. If the limit is reached, the transaction is rejected, otherwise any spare gas is refunded.

We assume the existence of three cost models, $\$_{\mathsf{zk}}$, $\$_{\mathsf{std}}$, and $\mathrm{E}_{\mathsf{std}}$, calculating the cost of a transition function execution, a calculating the cost of a public state oracle execution, and estimating the cost of a public state oracle execution respectively.

The oracle cost model $\$_{\mathsf{std}}$ takes a state σ, a gas budget gas, and an oracle query q. It returns the amount of spare gas spare, and the query results σ', and r, ensuring the computation does not exceed the allotted gas limit: $(\mathsf{spare}, \sigma', r) \leftarrow \$_{\mathsf{std}}(\sigma, \mathsf{gas}, q)$. If this limit is reached, it returns $(0, \bot, \bot)$, otherwise, $(\sigma', r) = q(\sigma, \varnothing)$.

The cost oracle cost estimator $\mathrm{E}_{\mathsf{std}}$ takes a state σ, and an oracle query q as inputs, and returns an estimate e for the gas $\$_{\mathsf{std}}$ will require: $e \leftarrow \mathrm{E}_{\mathsf{std}}(\sigma, q)$. This is used to allocate the gas budget of each given query.

The transition function cost model $\$_{\mathsf{zk}}$ is parameterised with oracle access to $\mathcal{O}_\sigma$ and $\mathcal{O}_\rho$, and given a transition function Γ and input w as inputs. It returns afixed cost c, and a result y: $(c, y) \leftarrow \$_{\mathsf{zk}}(\mathcal{O}_\sigma, \mathcal{O}_\rho, \Gamma, w)$, where $y = \Gamma_{\mathcal{O}_\sigma, \mathcal{O}_\rho}(w)$. It further guarantees that for each oracle query q made in Γ, the following code is executed instead:

let $e \leftarrow \mathcal{O}_\rho(\lambda(\rho, \cdot) : \text{let } (\rho.\sigma^E, \cdot) \leftarrow q(\rho.\sigma^E, \varnothing) \text{ in } (\rho, \mathrm{E}_{\mathsf{std}}(\rho.\sigma^E, q)))$
let $r \leftarrow \mathcal{O}_\sigma(\lambda(\sigma, \cdot) : \text{let } (\mathsf{sp}, \sigma', r) \leftarrow \$_{\mathsf{std}}(\sigma, e, q); \sigma'.\mathsf{spare} \leftarrow \sigma'.\mathsf{spare} + \mathsf{sp} \text{ in } (\sigma', r))$

The cost c returned must be at least the sum of all values e for each above queries.

This directly has the effect that a) the estimate e is not exceeded by any query, and b) that $\sigma'.\mathsf{spare}$ is precisely the difference between estimates and actual costs. We assume here that $\sigma.\mathsf{spare}$, and $\rho.\sigma^E$ are not touched by Γ, and are initialised to 0 and σ^π respectively. As a result, the smart contract system using this cost model can ensure that spare funds are refunded, and that the user pays the necessary fees.

It is worth noting that this is not entirely on par with existing fee models in smart contract systems, due to one key issue: A failed transaction in Kachina has no effect by definition. This also means that no miner can be awarded gas fees for a failed transaction, as the fee payment is itself an effect of the transaction. There are a few ways this could be solved – for instance introducing multiple failure states with different effects, or splitting transactions into linked fee-payment and contract call sub-transactions. For instance, each transaction could consist of both a contract call transaction as listed below, and a fee-payment transaction of the same value, with the consensus rules ensuring the former is placed

in the ledger iff the gas cost is sufficient, and the latter otherwise. We do not construct these here, but observe that in a practical system the payment of fees will require a special status to combat denial of service attacks.

Transition Function Γ_{scs}

The multiplexing with registration, loopback, payments, and fees transition function Γ_{scs} allows addressing any pair of address and sub-transition function (a, Γ). These sub-transition functions may, return values of either $(\textsc{call}, A, M)$, or $(\textsc{return}, y)$. In the former case, a different sub-transition function is invoked, and the value it eventually returns is fed back into the original one, by re-invoking it with $(\textsc{resume}, y)$. The transition function tabulates the cost of the call, and ensures the caller sends this at the end of a valid transition.

Public state variables and initialisation values:

Variable	Description
$\Sigma := \varnothing$	Mapping from address pairs to public states
$\mathsf{spare} := 0$	Temporary book-keeping of the value to return

---

Private state variables and initialisation values:

Variable	Description
$P := \varnothing$	Mapping from address pairs to private states
$\sigma^E := \varnothing$	Temporary projected state for estimating gas costs

When receiving an input $(\textsc{token}, w)$*:*

assert $w \neq (\textsc{system-send}, \ldots)$
let $\mathcal{O}'_\sigma \leftarrow \lambda q : \mathcal{O}_\sigma(\mathsf{muxPubOracle}(\textsc{token}, q))$
let $\mathcal{O}'_\rho \leftarrow \lambda q : \mathcal{O}_\rho(\mathsf{muxPrivOracle}(\textsc{token}, q))$
return $\Gamma_{\mathsf{sp},\mathcal{O}'_\sigma,\mathcal{O}'_\rho}(w)$

When receiving an input $(\textsc{call}, \mathsf{gasPrice}, v, A = (i, \Gamma, \mathsf{desc}, \mathsf{dep}), w)$*:*

send $\textsc{init-spare}$ **to** $\mathcal{O}_\sigma$
send $\textsc{init-state}$ **to** $\mathcal{O}_\rho$
let $c \leftarrow 0$
let $\mathsf{pk} \leftarrow \Gamma_{\mathsf{scs},\mathcal{O}_\sigma,\mathcal{O}_\rho}(\textsc{token}, (\textsc{send}, A, v))$
let $c \leftarrow c + c_{\mathsf{send}}$
let $(c', y) \leftarrow \mathsf{call}_{\mathcal{O}_\sigma,\mathcal{O}_\rho}(v, \mathsf{pk}, A, w)$
let $c \leftarrow c + c'$
let $\mathsf{pk} \leftarrow \Gamma_{\mathsf{scs},\mathcal{O}_\sigma,\mathcal{O}_\rho}(\textsc{token}, (\textsc{send}, \textsc{fee-pot}, c \times \mathsf{gasPrice}))$
send $(\textsc{deinit}, \mathsf{pk}, \mathsf{gasPrice})$ **to** $\mathcal{O}_\sigma$
send $\textsc{deinit}$ **to** $\mathcal{O}_\rho$
return y

When receiving an public oracle query $\textsc{init-spare}$*:*

let $\sigma.\mathsf{spare} \leftarrow 0$

When receiving an public oracle query $(\textsc{deinit}, \mathsf{pk}, \mathsf{gasPrice})$*:*

run $\Gamma_{\mathsf{sp}}(\text{FEE-POT}, \mathsf{pk}, \sigma.\mathsf{spare} \times \mathsf{gasPrice})$
let $\sigma.\mathsf{spare} \leftarrow 0$

When receiving an public oracle query INIT-STATE*:*
let $\rho.\sigma^E \leftarrow \sigma^\pi$
When receiving an public oracle query DEINIT*:*
let $\rho.\sigma^E \leftarrow \varnothing$

Helper procedures:

function $\mathsf{subCall}_{\mathcal{O}_\sigma,\mathcal{O}_\rho}(v, A, A' = (i, \Gamma, \mathsf{desc}, \mathsf{dep}), w)$
  assert $A' \neq$ TOKEN
  let $\mathcal{O}'_\sigma \leftarrow \lambda q : \mathcal{O}_\sigma(\mathsf{muxPubOracle}(\text{TOKEN}, q))$
  let $\mathcal{O}'_\rho \leftarrow \lambda q : \mathcal{O}_\rho(\mathsf{muxPrivOracle}(\text{TOKEN}, q))$
  run $\Gamma_{\mathsf{sp},\mathcal{O}'_\sigma,\mathcal{O}'_\rho}$ (SYSTEM-SEND, A, A', v)
  let $(c, y) \leftarrow \mathsf{call}_{\mathcal{O}_\sigma,\mathcal{O}_\rho}(v, A, A', w)$
  return $(c + c_{\mathsf{send}}, y)$
function $\mathsf{call}_{\mathcal{O}_\sigma,\mathcal{O}_\rho}(v, A, A' = (\cdot, \Gamma, \cdot, \cdot), w)$
  let $\mathcal{O}'_\sigma \leftarrow \lambda q : \mathcal{O}_\sigma(\mathsf{muxPubOracle}(A', q))$
  let $\mathcal{O}'_\rho \leftarrow \lambda q : \mathcal{O}_\rho(\mathsf{muxPrivOracle}(A', q))$
  let $c \leftarrow 0; e \leftarrow 0$
  repeat
    let $(c', y) \leftarrow \$_{\mathsf{zk}}(\mathcal{O}'_\sigma, \mathcal{O}'_\rho, \Gamma, (\text{CALL}, A, v, w))$
    let $c \leftarrow c + c'$
    if $\exists v', A'', w' : y = (\text{CALL}, v', A'', w')$ **then**
      let $(c', y) \leftarrow \mathsf{subCall}_{\mathcal{O}_\sigma,\mathcal{O}_\rho}(v', A', A'', w')$
      let $c \leftarrow c + c'; w \leftarrow (\text{RESUME}, y)$
  until $\exists y' : y = (\text{RETURN}, y')$
  return (c, y')
function $\mathsf{muxPubOracle}(A, q, \sigma, \varnothing)$
  if $A \notin \sigma.\Sigma$ **then let** $\sigma.\Sigma(A) \leftarrow \varnothing$
  let $\sigma' \leftarrow \sigma.\Sigma(A)$
  let $(\sigma', y) \leftarrow q(\sigma', \varnothing)$
  if $\sigma' = \bot$ **then return** $(\bot, y)$
  else
    let $\sigma.\Sigma(A) \leftarrow \sigma'$
    return (σ, y)
function $\mathsf{muxPrivOracle}(A, q, \rho, (\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta))$
  let $(\rho.P, \sigma^o.\Sigma.\rho^o.P, \sigma^\pi.\Sigma, \rho^\pi.P) \leftarrow$
    $\mathsf{forceInitMaps}((\rho.P, \sigma^o.\Sigma, \rho^o.P, \sigma^\pi.\Sigma, \rho^\pi.P), A, \varnothing)$
  let $\rho' \leftarrow \rho.P(A)$
  let $z' \leftarrow (\sigma^o.\sigma_i, \rho^o.\rho_i, \sigma^\pi.\sigma_i, \rho^\pi.\rho_i, \eta)$
  let $(\rho', y) \leftarrow q(\rho', z')$
  if $\rho' = \bot$ **then return** $(\bot, y)$
  else
    let $\rho.P(A) \leftarrow \rho'$
    return (ρ, y)

$\mathsf{desc}_{\mathsf{scs}} = \mathsf{desc}_{\mathsf{paymux}}$
$\mathsf{dep}_{\mathsf{scs}} = \mathsf{dep}_{\mathsf{paymux}}$

J.6 Exporting Ledger Data

Real-world smart contract systems often have some means to extract limited information about the underlying consensus protocol, such as the hash of the most recent block, the address of the block's miner, or the length of the current chain. These can be useful in applications – in particular the latter, as it gives an provides an imprecise clock for use in contracts.

Clearly, these rely on tighter integration with the underlying consensus mechanism than KACHINA provides. We can still capture the core idea, by having a sub-contract which manages such chain data, and allows this to be read and set arbitrarily[6]. We can then assume that the correct usage of this sub-contract is enforced by the validation of the underlying consensus mechanism – transactions which attempt to "incorrectly" set the chain data – for any definition of "correct" will never reach the ledger.

Transition Function $\Gamma_{\mathsf{chaindata}}$

The chain data transition function $\Gamma_{\mathsf{chaindata}}$ allow arbitrary setting and reading of state. An external assumption is that the setting of state is both enforced and restricted by the underlying ledger protocol, to give it meaning – for instance each block may induce a phantom "chain-data" transaction which appears on the ledger, and sets the most recent block hash in the chain-data contracts state.

When receiving an input (SET, σ')*:*
  run $\mathcal{O}_\sigma(\lambda(\cdot,\cdot) : (\sigma', \top))$
When receiving an input GET*:*
  return $\mathcal{O}_\sigma(\lambda(\sigma,\cdot) : (\sigma, \sigma))$

The contract we present here does have a further issue: Since the loopback in our multiplexers occurs only in the main transition function, the transcripts it generates will commit to specific values for the ledger data upon transaction creation – something which is likely not reasonable. A more complex loopback design, which we do not present here, would solve this: If calling into public or private parts of other contracts were permitted from within the public and private state oracles respectively.

For both leakage descriptors and dependencies, we make use of our assumption that users cannot directly call SET.

function $\mathsf{desc}_{\mathsf{chaindata}}(t, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, w, z)$
  return "Reading the chain data"

function $\mathsf{dep}_{\mathsf{chaindata}}(X, \mathcal{T}_\rho, (\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta))$
  return ϵ

[6] This could be expanded to allow only certain types of setting – such as advancing the time, but not rewinding it.